

# Introduktion till programmering med Scala

Föreläsningar & uppgifter  
för läsning på skärm



Björn Regnell

EDAA45, Lp1-2, HT 2023  
Datavetenskap, LTH  
Lunds universitet

Kompileringsdatum: 7 november 2023  
<http://cs.lth.se/pgk>

*Editor:* Björn Regnell

*Contributors* in alphabetical order: Anders Buhl, André Philipsson Eriksson, Anna Axelsson, Anna Palmqvist Sjövall, Anton Andersson, Benjamin Lindberg, Björn Regnell, Casper Schreiter, Cecilia Lindskog, Dag Hemberg, Elliot Bräck, Elsa Cervetti Ogestad, Emelie Engström, Emil Wihlander, Erik Bjäreholt, Erik Grampp, Evelyn Beck, Fredrik Danebjer, Fritjof Bengtsson, Gustav Cedersjö, Henrik Olsson, Hussein Taher, Jakob Hök, Jakob Sinclair, Johan Ravnborg, Jonas Danebjer, Jos Rosenqvist, Maj Stenmark, Maria Kulesh, Måns Magnusson, Nicholas Boyd Isacson, Niklas Sandén, Oliver Levay, Oliver Persson, Oscar Sigurdsson, Oskar Berg, Oskar Widmark, Patrik Persson, Per Holm, Philip Sadrian, Sandra Nilsson, Sebastian Hegardt, Simon Persson, Stefan Jonsson, Theodor Lundqvist, Tim Borglund, Tom Postema, Valthor Halldorsson, Viktor Claesson, Wilhelm Wanecek, William Karlsson.

*Home:* <https://cs.lth.se/pgk>

*Repo:* <https://github.com/lunduniversity/introprog>

This compendium is on-going work.

**Contributions are welcome!**

*Contact:* [bjorn.regnell@cs.lth.se](mailto:bjorn.regnell@cs.lth.se)

You can use this work if you respect this *LICENCE*: CC BY-SA 4.0

<http://creativecommons.org/licenses/by-sa/4.0/>

Please do *not* distribute your solutions to lab assignments and projects.

Copyright © 2015-2023.

Dept. of Computer Science, LTH, Lund University. Lund. Sweden.

# Framstegsprotokoll

## Genomförda övningar

Till varje laboration hör en övning med uppgifter som utgör förberedelse inför labben. Du behöver minst behärska grunduppgifterna för att klara labben inom rimlig tid. Om du känner att du behöver öva mer på grunderna, gör då även extrauppgifterna. Om du vill fördjupa dig, gör fördjupningsuppgifterna som är på mer avancerad nivå. Kryssa för nedan vilka övningar du har gjort, så blir det lättare för din handledare att anpassa dialogen till de kunskaper du förvärvat hittills.

Övning	Grund	Extra	Fördjupning
expressions	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
programs	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
functions	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
objects	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
classes	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
patterns	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
sequences	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
matrices	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
lookup	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
inheritance	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
context	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
extra	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
example	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

## Godkända obligatoriska moment

För att bli godkänd på laborationsuppgifterna och projektuppgiften måste du lösa deluppgifterna och diskutera dina lösningar med en handledare. Denna diskussion är din möjlighet att få feedback på dina lösningar. Ta vara på den! Se till att handledaren noterar nedan när du blivit godkänd på respektive obligatorisk moment. Spara detta blad tills du fått slutbetyg i kursen.

Namn: .....

Namnteckning: .....

Lab	kompl+datum,gk+datum	Handl. underskr. + namnförtydl.
kojo	.....	.....
irritext	.....	.....
blockmole	.....	.....
blockbattle0	.....	.....
blockbattle1	.....	.....
shuffle	.....	.....
life	.....	.....
words	.....	.....
snake0	.....	.....
snake1	.....	.....
<b>Projektuppgift</b>	.....	.....
<input type="checkbox"/> bank	<i>Om egendef., ge kort beskrivning här:</i>	
<input type="checkbox"/> music		
<input type="checkbox"/> photo		
<input type="checkbox"/> egendefinerad		
<b>Muntligt prov</b>		
<input type="checkbox"/> godkänd	.....	.....

# Förord

Programmering är inte bara ett sätt att ta makten över de människoskapade system som är förutsättningen för vårt moderna samhälle och dess fortsatta digitalisering. Programmering är också ett kraftfullt verktyg för tanken. Med kunskap i programmeringens grunder kan du påbörja den livslånga läranderesan som det innebär att vara systemutvecklare och abstraktionskonstnär. Programmeringsspråk och utvecklingsverktyg kommer och går, men de grundläggande koncepten bakom *all* mjukvara består: sekvens, alternativ, repetition och abstraktion.

Detta kompendium utgör kursmaterial för en grundkurs i programmering, som syftar till att ge en solid bas för ingenjörstudenter och andra som vill utveckla system med mjukvara. Materialet omfattar en termins studier på kvartsfart och förutsätter kunskaper motsvarande gymnasienivå i svenska, matematik och engelska.

Kompendiet distribueras som öppen källkod. Det får användas fritt så länge erkännande ges och eventuella ändringar publiceras under samma licens som ursprungsmaterialet.

I kursrepot [github.com/lunduniversity/introprog](https://github.com/lunduniversity/introprog) finns instruktioner om hur du kan bidra till kursmaterialet.

Läromaterialet fokuserar på lärande genom praktiskt programmeringsarbete och innehåller övningar och laborationer som är organiserade i moduler. Varje modul har ett tema och en teoridel som bearbetas på föreläsningar.

I kursen använder vi programmeringsspråket Scala, som har enkel syntax och möjliggör flera principiellt olika sätt att programmera på, i ett och samma språk. Vi använder Scala för att illustrera grunderna i imperativ och objektorienterad programmering, tillsammans med elementär funktionsprogrammering.

Den kanske viktigaste framgångsfaktorn vid studier i programmering är att du bejakar din egen upptäckarglädje och experimentlusta. Det fantastiska med programmering är att dina egna intellektuella konstruktioner faktiskt *gör* något som just *du* har bestämt! Ta vara på det och prova dig fram genom att koda egna idéer – det är kul när det funkar, men minst lika lärorikt är felsökning, buggrättande och alla misslyckade försök som, ibland efter hårt arbete vänds till lyckade lösningar och bestående lärdomar.

Välkommen till datavetenskapens fascinerande värld och hjärtligt lycka till med dina studier!

*Lund, 7 november 2023, Björn Regnell*

# Innehåll

<b>Framstegsprotokoll</b>	<b>ii</b>
<b>Förord</b>	<b>iv</b>
<b>I Om kursen</b>	<b>1</b>
<b>-2 Kursens arkitektur</b>	<b>2</b>
-2.0.1 Veckoöversikt . . . . .	2
-2.1 Om ditt lärande . . . . .	5
-2.1.1 Vad lär du dig? . . . . .	5
-2.1.2 Progression . . . . .	5
-2.1.3 Hur lär du dig? . . . . .	5
-2.1.4 Kursmoment — varför? . . . . .	6
-2.1.5 En typisk kursvecka . . . . .	7
<b>-1 Anvisningar</b>	<b>8</b>
-1.1 Samarbetsgrupper . . . . .	8
-1.1.1 Samarbetskontrakt. . . . .	9
-1.1.2 Grupplaboration . . . . .	9
-1.1.3 Samarbetsbonus . . . . .	10
-1.2 Föreläsningar . . . . .	10
-1.3 Övningar . . . . .	11
-1.4 Resurstider . . . . .	12
-1.5 Laborationer. . . . .	13
-1.6 Kontrollskrivning . . . . .	14
-1.7 Projektuppgift . . . . .	15
-1.8 Muntligt prov . . . . .	16
-1.9 Valfri tentamen . . . . .	16
<b>0 Hur bidra till kursmaterialet?</b>	<b>17</b>
0.1 Bidrag är varmt välkomna! . . . . .	17
0.2 Instruktioner . . . . .	17
0.2.1 Vad behövs för att kunna bidra? . . . . .	17
0.2.2 Svenska eller engelska? . . . . .	17
0.3 Exempel. . . . .	18

<b>II</b>	<b>Moduler</b>	<b>21</b>
<b>1</b>	<b>Introduktion</b>	<b>22</b>
1.1	Teori . . . . .	23
1.1.1	Hur fungerar en dator? . . . . .	23
1.1.2	Vad är programmering? . . . . .	23
1.1.3	Vad är en kompilator? . . . . .	23
1.1.4	Virtuell maskin (VM) == abstrakt hårdvara . . . . .	24
1.1.5	Vad består ett program av? . . . . .	24
1.1.6	Exempel på programmeringsspråk . . . . .	24
1.1.7	Olika programmeringsparadigm. . . . .	25
1.1.8	Hello world . . . . .	25
1.1.9	Utvecklingscykeln . . . . .	25
1.1.10	Utvecklingsverktyg. . . . .	26
1.1.11	Installera verktyg på din egen dator . . . . .	26
1.1.12	Scala Command Line Interface (CLI) . . . . .	27
1.1.13	Tips och trix med scala i terminalen . . . . .	27
1.1.14	Litteraler . . . . .	27
1.1.15	Exempel på inbyggda datatyper i Scala . . . . .	28
1.1.16	Grundtyper i Scala . . . . .	28
1.1.17	Grundtypernas omfång . . . . .	28
1.1.18	Uttryck . . . . .	29
1.1.19	Variabler . . . . .	29
1.1.20	Regler för identifierare . . . . .	29
1.1.21	Att bygga strängar: konkatenering och interpolering . . . . .	30
1.1.22	Heltalsaritmetik . . . . .	30
1.1.23	Flyttalsaritmetik. . . . .	31
1.1.24	Definiera namn på uttryck . . . . .	31
1.1.25	Funktion, argument, parameter . . . . .	31
1.1.26	Färdiga matte-funktioner i paketet <code>scala.math</code> . . . . .	32
1.1.27	Logiska uttryck . . . . .	32
1.1.28	De Morgans lagar . . . . .	33
1.1.29	Alternativ med if-uttryck . . . . .	33
1.1.30	Uttryck eller sats? . . . . .	33
1.1.31	Variabeldeklaration och tilldelningssats . . . . .	34
1.1.32	Tilldelningssatser är <i>inte</i> matematisk likhet . . . . .	34
1.1.33	Förkortade tilldelningssatser . . . . .	34
1.1.34	Exempel på förkortade tilldelningssatser. . . . .	35
1.1.35	Variabler som ändrar värden kan vara knepiga . . . . .	35
1.1.36	Kontrollstrukturer: alternativ och repetition . . . . .	35
1.1.37	Scala-2-syntax för kontrollstrukturer fungerar i Scala 3 . . . . .	36
1.1.38	Repetera många satser . . . . .	36
1.1.39	Procedurer. . . . .	37

1.1.40	Problemlösning: nedbrytning i abstraktioner som sen kombineras . . . . .	37
1.1.41	Övning expressions och labb kojo . . . . .	37
1.1.42	Köa med Sigrid . . . . .	38
1.1.43	Sigrid in action. . . . .	38
1.2	Övning expressions . . . . .	39
1.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	39
1.2.2	Extrauppgifter; träna mer . . . . .	47
1.2.3	Fördjupningsuppgifter; utmaningar . . . . .	50
1.3	Laboration: kojo . . . . .	53
1.3.1	Obligatoriska uppgifter . . . . .	53
1.3.2	Kontrollfrågor . . . . .	56
1.3.3	Frivilliga extrauppgifter . . . . .	56

## 2 Program och kontrollstrukturer 62

2.1	Teori . . . . .	63
2.1.1	Vad är en datastruktur? . . . . .	63
2.1.2	Några samlingar i <code>scala.collection</code> . . . . .	63
2.1.3	Olika strukturer för att hantera data . . . . .	64
2.1.4	Vad är en vektor? . . . . .	64
2.1.5	En konceptuell bild av en vektor. . . . .	64
2.1.6	En samling strängar . . . . .	65
2.1.7	Vad är en kontrollstruktur? . . . . .	65
2.1.8	Loopa genom elementen i en vektor . . . . .	66
2.1.9	Bygg ny samling från befintlig med <code>for-yield</code> -uttryck . . . . .	66
2.1.10	Samlingen <code>Range</code> håller reda på intervall . . . . .	66
2.1.11	Loopa med <code>Range</code> . . . . .	67
2.1.12	Loopa med <code>Range</code> skapad med <code>to</code> . . . . .	67
2.1.13	Vad är en <code>Array</code> ? . . . . .	67
2.1.14	Några likheter & skillnader mellan <code>Vector</code> och <code>Array</code> . . . . .	67
2.1.15	Kompilering i terminalen . . . . .	68
2.1.16	Scala Command Line Interface (CLI) . . . . .	68
2.1.17	Ett minimalt fristående program i Scala . . . . .	69
2.1.18	Loopa genom en samling med en <code>while</code> -sats . . . . .	69
2.1.19	Strängargument till i ett program med primitiv <code>main</code> . . . . .	69
2.1.20	Typsäkra argument till i ett program med <code>@main</code> . . . . .	70
2.1.21	För kännedom: Scala- <b>skript</b> . . . . .	70
2.1.22	Vad är en algoritm? . . . . .	71
2.1.23	Algoritmexempel: N-FAKULTET . . . . .	71
2.1.24	Algoritmexempel: MIN . . . . .	71
2.1.25	Mall för funktionsdefinitioner . . . . .	72
2.1.26	Bättre många små abstraktioner som gör en sak var . . . . .	72
2.1.27	Vad är ett block? . . . . .	72
2.1.28	Namn i block blir <b>lokala</b> . . . . .	73



2.1.29	Parameter och argument . . . . .	.73
2.1.30	Procedurer. . . . .	.74
2.1.31	”Ingenting” är faktiskt någonting i Scala . . . . .	.74
2.1.32	Problemlösning: nedbrytning i abstraktioner som sen kombineras . . . . .	.74
2.1.33	Exempel på <b>funktionell nedbrytning</b> . . . . .	.75
2.1.34	Varför abstraktion? . . . . .	.75
2.1.35	Från källkod till maskinkod med JVM . . . . .	.76
2.1.36	Paket . . . . .	.76
2.1.37	Import . . . . .	.76
2.1.38	Jar-filer . . . . .	.77
2.2	Övning programs. . . . .	.78
2.2.1	Grunduppgifter . . . . .	.78
2.2.2	Extrauppgifter; träna mer . . . . .	.83
2.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.85

### 3 Funktioner och abstraktion 87

3.1	Teori . . . . .	.88
3.1.1	Vad är abstraktion? . . . . .	.88
3.1.2	Exempel på abstraktionsmekanismer inom datavetenskapen . . . . .	.88
3.1.3	Funktion: deklaration och anrop. . . . .	.88
3.1.4	Deklarera funktioner, överlagring . . . . .	.89
3.1.5	Funktioner med defaultargument . . . . .	.89
3.1.6	Funktioner med namngivna argument . . . . .	.89
3.1.7	Enhetlig access . . . . .	.90
3.1.8	Anropsstacken och objektheapen . . . . .	.90
3.1.9	Aktiveringspost . . . . .	.91
3.1.10	Vad är en stack trace? . . . . .	.91
3.1.11	Hur läsa en stack trace? . . . . .	.91
3.1.12	Lokala funktioner . . . . .	.92
3.1.13	Funktioner är äkta värden i Scala . . . . .	.92
3.1.14	Funktionsvärden kan vara argument . . . . .	.93
3.1.15	Applicera funktioner på element i samlingar med map . . . . .	.93
3.1.16	Applicera funktioner på element i samlingar med map . . . . .	.93
3.1.17	Äkta funktioner . . . . .	.94
3.1.18	Exempel på oäkta funktioner: slumptal . . . . .	.94
3.1.19	Slumptalsfrö: få samma slumptal varje gång . . . . .	.94
3.1.20	Anonyma funktioner . . . . .	.95
3.1.21	Applicera anonyma funktioner på element i samlingar . . . . .	.95
3.1.22	Platshållarsyntax för anonyma funktioner . . . . .	.96
3.1.23	Exempel på platshållarsyntax med reduceLeft. . . . .	.96
3.1.24	Predikat, med och utan namn . . . . .	.97
3.1.25	Funktionsvärde vid tom parameterlista: använd ”thunk” . . . . .	.97
3.1.26	Hur fungerar egentligen upprepa i Kojo? . . . . .	.97

3.1.27	Multipla parameterlistor . . . . .	.98
3.1.28	Värdeanrop och namnanrop . . . . .	.98
3.1.29	Klammerparenteser vid ensam parameter . . . . .	.99
3.1.30	Skapa din egen kontrollstruktur . . . . .	.99
3.1.31	Kolon vid ensam parameter . . . . .	.99
3.1.32	Stegade funktioner, "Curry-funktioner" . . . . .	.100
3.1.33	Funktion med fångad variabelrymd: <i>closure</i> . . . . .	.100
3.1.34	Rekursiva funktioner . . . . .	.100
3.1.35	Loopa med rekursion . . . . .	.101
3.1.36	Rekursiva datastrukturer . . . . .	.101
3.1.37	Kompilera om det som ändrats vid varje sparning . . . . .	.101
3.2	Övning functions . . . . .	.103
3.2.1	Grunduppgifter; förberedelse inför laboration . . . . .	.103
3.2.2	Extrauppgifter; träna mer . . . . .	.107
3.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.109
3.3	Laboration: irritext . . . . .	.112
3.3.1	Krav . . . . .	.112
3.3.2	Tips för att komma igång . . . . .	.112
3.3.3	Inspiration . . . . .	.113

## 4 Objekt och inkapsling 115

4.1	Teori . . . . .	.116
4.1.1	Vad rymmer sköldpaddan i Kojo i sitt tillstånd? . . . . .	.116
4.1.2	Vad är ett objekt? . . . . .	.116
4.1.3	Deklarera, allokerar, referera . . . . .	.116
4.1.4	Olika sätt att allokerar objekt . . . . .	.117
4.1.5	Vad är ett singelobjekt? . . . . .	.117
4.1.6	Allokering: minne reserveras med plats för data . . . . .	.118
4.1.7	Punktnotation, tillståndsförändring med tilldelning . . . . .	.118
4.1.8	Punktnotation och operatornotation . . . . .	.118
4.1.9	Namnrymd och skuggning . . . . .	.119
4.1.10	Inkapsling: att dölja interna delar . . . . .	.119
4.1.11	Idiom: Privata variabler med understreck vid "krock" . . . . .	.120
4.1.12	Principen om enhetlig access . . . . .	.120
4.1.13	Exempel: singelobjektet med förändringsbart tillstånd . . . . .	.120
4.1.14	Exempel: tillstånd, attribut . . . . .	.121
4.1.15	Tillståndsändring . . . . .	.121
4.1.16	Modul . . . . .	.121
4.1.17	Deklarera paket . . . . .	.122
4.1.18	Kompilera paket . . . . .	.122
4.1.19	Paket i REPL . . . . .	.123
4.1.20	Vad är en tupel? . . . . .	.123
4.1.21	Tupler som parametrar och returvärde . . . . .	.123

4.1.22	Ett smidigt sätt att skapa 2-tupler med metoden <code>-&gt;</code> . . . . .	.124
4.1.23	Typalias för att abstrahera typnamn. . . . .	.124
4.1.24	Lata variabler och fördröjd initialisering . . . . .	.124
4.1.25	Singelobjekt är lata . . . . .	.125
4.1.26	Vad är skillnaden mellan <code>val</code> , <code>var</code> , <code>def</code> , <code>lazy val</code> ? . . . . .	.125
4.1.27	Be kompilatorn att varna vid initialiseringsproblem . . . . .	.125
4.1.28	Be kompilatorn ge fler bra varningar . . . . .	.126
4.1.29	Programmeringsparadigm . . . . .	.126
4.1.30	Funktioner är äkta objekt i Scala . . . . .	.127
4.1.31	Fördjupning: Äkta funktionsobjekt är av funktionstyp. . . . .	.127
4.1.32	Vad är en klass? . . . . .	.127
4.1.33	Vad är en klass? . . . . .	.127
4.1.34	Använda klassen <code>Color</code> . . . . .	.128
4.1.35	Lägg till metoder i efterhand med <code>extension</code> . . . . .	.128
4.1.36	Kollektiva <code>extension</code> smetoder . . . . .	.129
4.1.37	Import av alla namn i en viss modul . . . . .	.129
4.1.38	Namnbyte vid import. . . . .	.129
4.1.39	Exkludera (gömma) namn vid import . . . . .	.130
4.1.40	Lokal import-deklaration . . . . .	.130
4.1.41	Export . . . . .	.130
4.1.42	Använda dokumentation för färdiga klasser. . . . .	.131
4.1.43	Vad är en <code>jar</code> -fil? . . . . .	.131
4.1.44	Öppen källkod på Maven Central . . . . .	.131
4.1.45	Vad är <code>classpath</code> ? . . . . .	.132
4.1.46	Färdiga grafikmetoder i klassen <code>PixelWindow</code> . . . . .	.132
4.1.47	Automatiska beroenden med Scala CLI i REPL: . . . . .	.132
4.1.48	Köra program + kodbibliotek med Scala CLI. . . . .	.133
4.1.49	Kompilera om vid varje ändring . . . . .	.133
4.2	Övning <code>objects</code> . . . . .	.134
4.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	.134
4.2.2	Extrauppgifter; träna mer . . . . .	.141
4.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.142
4.3	Laboration: <code>blockmole</code> . . . . .	.145
4.3.1	Bakgrund . . . . .	.145
4.3.2	Obligatoriska uppgifter . . . . .	.145
4.3.3	Kontrollfrågor . . . . .	.150
4.3.4	Frivilliga extrauppgifter . . . . .	.150

## 5 Klasser och datamodellering

154

5.1	Teori . . . . .	.155
5.1.1	En metafor för klass: Stämpel . . . . .	.155
5.1.2	Vad är en klass? . . . . .	.155
5.1.3	Datamodellering . . . . .	.155

5.1.4	Singelobjekt jämfört med klass . . . . .	.156
5.1.5	Förändring av objektets tillstånd . . . . .	.156
5.1.6	Bättre att initialisera med hjälp av klassparametrar . . . . .	.156
5.1.7	KlassdeklARATIONER och instansiering . . . . .	.157
5.1.8	Övning: en klass som representerar en person . . . . .	.157
5.1.9	Lösning: klassen Person . . . . .	.158
5.1.10	Skapa egen najs toString . . . . .	.158
5.1.11	Instansprivata klassparametrar . . . . .	.159
5.1.12	Case-klasser är som vanliga klasser med extra godis . . . . .	.159
5.1.13	Fördjupning: Styra synlighet med private[X] . . . . .	.160
5.1.14	Styra användningen av infix alfanumeriska operatorer . . . . .	.160
5.1.15	Övning: Klassen Complex . . . . .	.160
5.1.16	Exempel: Klassen Complex . . . . .	.161
5.1.17	Exempel: Principen om enhetlig access . . . . .	.161
5.1.18	Instansiering med direkt användning av new . . . . .	.162
5.1.19	Indirekt instansiering med fabriksmetoder . . . . .	.162
5.1.20	Hur förhindra direkt instansiering? . . . . .	.162
5.1.21	Kompanjonsobjekt med indirekt instansiering . . . . .	.163
5.1.22	Användning av kompanjonsobjekt med fabriksmetoder . . . . .	.163
5.1.23	Alternativa direktinstansieringar med default-argument . . . . .	.164
5.1.24	Alternativa sätt att instansiera med fabriksmetod . . . . .	.164
5.1.25	Medlemmar som bara behövs i en enda upplaga . . . . .	.165
5.1.26	Medlemmar i singelobjekt är statiskt allokerade . . . . .	.165
5.1.27	Attribut i kompanjonsobjekt användas för sådant som är gemensamt för alla instanser . . . . .	.165
5.1.28	Övning: en läskig mutant . . . . .	.166
5.1.29	Case-klasser . . . . .	.166
5.1.30	Exempel: oföränderliga case-klassen Point . . . . .	.167
5.1.31	Vad är en konstruktör? . . . . .	.167
5.1.32	Hjälpkonstruktörer i Scala (ovanliga) . . . . .	.167
5.1.33	Användning av hjälpkonstruktör . . . . .	.168
5.1.34	Referens saknas: null . . . . .	.168
5.1.35	Exempel: null . . . . .	.168
5.1.36	Defaultvärden under pågående konstruktion . . . . .	.169
5.1.37	Problem med initialisering av attribut vid konstruktion . . . . .	.169
5.1.38	Vilka värden har attribut medan konstruktion pågår? . . . . .	.169
5.1.39	Hur undvika initialiseringsproblem vid konstruktion? . . . . .	.170
5.1.40	Referensen this . . . . .	.170
5.1.41	Getters och setters . . . . .	.171
5.1.42	Java-exempel: Klassen JPerson . . . . .	.171
5.1.43	Motsvarande JPerson i Scala . . . . .	.172
5.1.44	Förhindra felaktiga attributvärden med setters . . . . .	.172
5.1.45	Getters och setters i Scala . . . . .	.173

5.1.46	Referenslikhet eller innehållslikhet? . . . . .	.173
5.1.47	Exempel: referenslikhet och innehållslikhet . . . . .	.173
5.1.48	Referenslikhet och egna klasser . . . . .	.174
5.1.49	Case-klasser ger innehållslikhet. . . . .	.174
5.1.50	Likhet och case-klasser . . . . .	.174
5.1.51	Sammanfattning case-klass-godis . . . . .	.175
5.1.52	Implementation saknas: ??? . . . . .	.175
5.1.53	Exempel: ofärdig kod . . . . .	.175
5.2	Övning classes . . . . .	.177
5.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	.177
5.2.2	Extrauppgifter; träna mer . . . . .	.183
5.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.187
5.3	Laboration: blockbattle0 . . . . .	.192
<b>6</b>	<b>Mönster och felhantering</b>	<b>193</b>
6.1	Teori . . . . .	.194
6.1.1	Bastypen för alla typer: Any . . . . .	.194
6.1.2	Alla typer är subtyper till Any . . . . .	.194
6.1.3	Dina egna referenstyper är subtyper till AnyRef . . . . .	.194
6.1.4	Vad är matchning? . . . . .	.195
6.1.5	Plocka isär ett objekt i sina beståndsdelar med mönster . . . . .	.195
6.1.6	Kolla om det passar med nästlade if-uttryck . . . . .	.195
6.1.7	Kolla om det passar med match-uttryck . . . . .	.196
6.1.8	Syntax för match-uttryck . . . . .	.196
6.1.9	Matchning med gard . . . . .	.197
6.1.10	Matchning med variabelmönster . . . . .	.197
6.1.11	Matchning med eller-mönster . . . . .	.197
6.1.12	Matchning med typade mönster . . . . .	.198
6.1.13	Fördjupning: Unionstyper och typen Matchable. . . . .	.198
6.1.14	Konstruktormönster med case-klasser . . . . .	.199
6.1.15	Plocka isär samlingar med djupa mönster . . . . .	.199
6.1.16	Matchning på tupler . . . . .	.200
6.1.17	Mönstermatchning och uppräknings med case-objekt. . . . .	.200
6.1.18	Mönstermatchning och förseglade typer . . . . .	.200
6.1.19	Mönstermatcha enumeration . . . . .	.201
6.1.20	Stora/små begynnelsebokstäver vid matchning . . . . .	.201
6.1.21	Stora/små begynnelsebokstäver vid matchning . . . . .	.202
6.1.22	Mönster på andra ställen än i match . . . . .	.202
6.1.23	Mönsterdelar och variabelt antal argument . . . . .	.202
6.1.24	Partiella funktioner och metoden collect . . . . .	.203
6.1.25	Fördjupning: metoden unapply . . . . .	.203
6.1.26	Hur hantera saknade värden? . . . . .	.203
6.1.27	En gemensam basstyp för ett värde som kanske saknas . . . . .	.204

6.1.28	Option för hantering av ev. saknade värden . . . . .	.204
6.1.29	Några smidiga metoder på Option. . . . .	.205
6.1.30	Några samlingsmetoder som ger en Option, övning . . . . .	.205
6.1.31	Några samlingsmetoder som ger en Option, svar . . . . .	.205
6.1.32	Vad är ett undantag (eng. <i>exception</i> )? . . . . .	.206
6.1.33	Orsaka undantag indirekt med require och assert. . . . .	.206
6.1.34	Kasta dina egna undantag med throw . . . . .	.207
6.1.35	En gemensam bastyp för något som kan misslyckas . . . . .	.207
6.1.36	Hantera undantag med Try . . . . .	.207
6.1.37	try-catch-uttryck . . . . .	.208
6.1.38	Unvik undantag om det går . . . . .	.208
6.1.39	Fördjupning: Kontrollerade undantag . . . . .	.209
6.1.40	Fördjupning: Implementera equals med match . . . . .	.209
6.1.41	Fördjupning: equals som fungerar för finala klasser . . . . .	.209
6.1.42	Fördjupning: Recept i 8 steg för arvssäker equals. . . . .	.210
6.1.43	Fördjupning: Säkrare likhetstest i Scala 3 . . . . .	.210
6.2	Övning patterns. . . . .	.212
6.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	.212
6.2.2	Fördjupningsuppgifter; utmaningar . . . . .	.217
6.3	Laboration: blockbattle1 . . . . .	.223
6.3.1	Bakgrund . . . . .	.224
6.3.2	Obligatoriska krav . . . . .	.224
6.3.3	Valbara krav – välj minst ett . . . . .	.225
6.3.4	Förebredelser inför redovisningen . . . . .	.225
6.3.5	Tips och förslag . . . . .	.225

## **7 Sekvenser och enumerationer . . . . . 228**

7.1	Teori . . . . .	.229
7.1.1	Vad är en sekvens? . . . . .	.229
7.1.2	Exempel: En sträng är en sekvens av tecken . . . . .	.229
7.1.3	Iterera över element i en sekvens . . . . .	.229
7.1.4	Lägg till i början och i slutet av en sekvens . . . . .	.230
7.1.5	Egenskaper hos några sekvenssamlingar i Scala . . . . .	.230
7.1.6	Vilken sekvenssamling ska jag välja? . . . . .	.231
7.1.7	Några konstigheter med Array . . . . .	.231
7.1.8	Oföränderlig eller förändringsbar? . . . . .	.232
7.1.9	Vad är en sekvensalgoritm? . . . . .	.232
7.1.10	Använda färdiga sekvenssamlingsmetoder . . . . .	.232
7.1.11	Några användbara samlingsmetoder vid implementation av sekvensalgoritmer. . . . .	.233
7.1.12	Uppdaterad sekvens med kraftfulla metoden patch . . . . .	.233
7.1.13	Använda for-uttryck för filtrering med hjälp av gard . . . . .	.233
7.1.14	Använda samlingsmetoden filter för filtrering . . . . .	.234



7.1.15	Vanliga sekvensproblem som funktionshuvuden . . . . .	.234
7.1.16	Implementation av sekvensproblem med for-uttryck och/eller färdiga samlingsmetoder. . . . .	.234
7.1.17	Implementation av sekvensproblem med map, filter . . . . .	.235
7.1.18	Hierarki av samlingstyper i scala.collection v2.13 . . . . .	.235
7.1.19	Lämna det öppet: använd Seq . . . . .	.235
7.1.20	Implementation med generiska funktioner . . . . .	.236
7.1.21	Använda Java-samlingar i Scala med CollectionConverters . . . . .	.236
7.1.22	Fördjupning: Skapa generisk Array . . . . .	.237
7.1.23	Repeterade parametrar blir sekvens . . . . .	.237
7.1.24	Sekvenssamling som argument till repeterade parametrar. . . . .	.238
7.1.25	Enumerationer har en ordning . . . . .	.238
7.1.26	Enumerationer kan ha parametrar och medlemmar. . . . .	.238
7.1.27	Enum kan bli fullfjädrade case-klasser. . . . .	.239
7.1.28	Enum och mönster-matchning. . . . .	.239
7.1.29	Fördelar med <b>enum</b> jämfört med uppräkningsmedeltal . . . . .	.240
7.1.30	Registrering . . . . .	.240
7.1.31	Registrering av tärningskast i Array. . . . .	.240
7.1.32	Registrering av tärningskast i Array. . . . .	.241
7.1.33	Skapa lösningar på sekvensproblem från grunden . . . . .	.241
7.1.34	Skapa ny sekvenssamling eller ändra på plats? . . . . .	.242
7.1.35	Algoritm: SEQ-COPY. . . . .	.242
7.1.36	Implementation av SEQ-COPY med while . . . . .	.242
7.1.37	Typ-alias för att abstrahera typnamn . . . . .	.243
7.1.38	Exempel: SEQ-INSERT/REMOVE-COPY . . . . .	.243
7.1.39	Pseudo-kod för SEQ-INSERT-COPY . . . . .	.244
7.1.40	Insättning/borttagning i kopia av primitiv Array . . . . .	.244
7.1.41	Exempel: PolygonWindow. . . . .	.244
7.1.42	Implementera Polygon . . . . .	.245
7.1.43	Exempel: PolygonArray, ändring på plats. . . . .	.245
7.1.44	Exempel: PolygonVector, variabel referens till oföränderlig datastruktur . . . . .	.246
7.1.45	Exempel: Polygon som oföränderlig case class. . . . .	.246
7.1.46	Att jämföra strängar lexikografiskt . . . . .	.247
7.1.47	Jämföra strängar: likhet . . . . .	.247
7.1.48	Algoritmexempel: stränglikhet, pseudokod . . . . .	.247
7.1.49	Algoritmexempel: stränglikhet, implementation . . . . .	.248
7.1.50	Jämföra strängar: "mindre än" . . . . .	.248
7.1.51	Jämföra strängar: "mindre än" . . . . .	.248
7.1.52	Jämföra strängar: "mindre än" . . . . .	.249
7.1.53	Sökning . . . . .	.249
7.1.54	Linjärsökning: hitta index för elementet x . . . . .	.250
7.1.55	Sortering . . . . .	.250
7.1.56	Det finns många olika sorteringsalgoritmer . . . . .	.250

7.1.57	Bogo sort . . . . .	.251
7.1.58	Sortera till ny vektor med insättningssortering: pseudo-kod . . . . .	.251
7.1.59	Sortera till ny vektor med insättningssortering: implementation . . . . .	.251
7.1.60	Sorter till ny samling med godtyckligt ordningspredikat . . . . .	.252
7.1.61	Insättningssortering på plats – pseudo-kod . . . . .	.252
7.1.62	Insättningssortering på plats – implementation . . . . .	.253
7.2	Övning sequences . . . . .	.254
7.2.1	Grunduppgifter; förberedelse inför laboration . . . . .	.254
7.2.2	Extrauppgifter; träna mer . . . . .	.262
7.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.265
7.3	Laboration: shuffle . . . . .	.270
7.3.1	Bakgrund . . . . .	.270
7.3.2	Obligatoriska uppgifter . . . . .	.271
7.3.3	Frivilliga extrauppgifter . . . . .	.275
7.3.4	Bilder med exempel på olika pokerhänder . . . . .	.275

## **8 Nästlade och generiska strukturer . . . . . 278**

8.1	Teori . . . . .	.279
8.1.1	Veckans labb: life . . . . .	.279
8.1.2	Vad är en matris? . . . . .	.279
8.1.3	Indexering i en matris . . . . .	.279
8.1.4	Hur skapa matriser? . . . . .	.280
8.1.5	Hur indexera i matriser? . . . . .	.280
8.1.6	Hur indexera i matriser? . . . . .	.281
8.1.7	Uppdatering av en förändringsbar nästlad struktur . . . . .	.281
8.1.8	Uppdatering av en förändringsbar nästlad struktur . . . . .	.281
8.1.9	Några olika sätt att skapa förändringsbara matriser . . . . .	.282
8.1.10	Exempel på skapande av oföränderlig nästlad struktur . . . . .	.282
8.1.11	Exempel på skapande av oföränderlig nästlad struktur . . . . .	.283
8.1.12	Uppdatering av en oföränderlig nästlad struktur . . . . .	.283
8.1.13	Uppdatering av en oföränderlig nästlad struktur . . . . .	.283
8.1.14	Iterera över nästlad struktur . . . . .	.284
8.1.15	Iterera över nästlad struktur . . . . .	.284
8.1.16	Övningsexempel: Yatzy . . . . .	.285
8.1.17	Övningsexempel: Yatzy . . . . .	.285
8.1.18	Iterera över nästlad struktur: for-sats . . . . .	.285
8.1.19	Iterera över nästlad struktur: for-sats . . . . .	.286
8.1.20	Nästlade for-uttryck . . . . .	.286
8.1.21	Nästlade for-uttryck . . . . .	.286
8.1.22	Nästlade map-uttryck . . . . .	.287
8.1.23	Nästlade map-uttryck . . . . .	.287
8.1.24	Fallgrop: likhet av array . . . . .	.287
8.1.25	Övning: Kolla likhet av array (uppfinner hjulet) . . . . .	.287



8.1.26	Använd <code>sameElements</code> för test av innehållslighet men bara på icke-nästlade arrayer . . . . .	.288
8.1.27	Om veckans övningar. . . . .	.288
8.1.28	Exempel: Icke-generisk case-klass med heltalsmatris . . . . .	.288
8.1.29	Exempel: Generisk case-klass med generell matris . . . . .	.289
8.1.30	Vad är en typparameter? . . . . .	.289
8.1.31	Exempel: Generisk funktion. . . . .	.290
8.1.32	Exempel: Generisk case-klass . . . . .	.290
8.1.33	Fallgrop: Typradering (eng. <i>type erasure</i> ). . . . .	.291
8.1.34	Testning och avlusning . . . . .	.291
8.2	Övning matrices. . . . .	.292
8.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	.292
8.2.2	Extrauppgifter; träna mer . . . . .	.296
8.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.298
8.3	Laboration: life . . . . .	.300
8.3.1	Bakgrund . . . . .	.300
8.3.2	Obligatoriska krav . . . . .	.301
8.3.3	Valbara krav – välj minst ett . . . . .	.302
8.3.4	Tips och förslag . . . . .	.303

## 9 Mängder och tabeller 305

9.1	Teori . . . . .	.306
9.1.1	Hierarki av samlingstyper i <code>scala.collection</code> v2.13 . . . . .	.306
9.1.2	Metoden <code>iterator</code> ger en "engångs-iterator" . . . . .	.306
9.1.3	Mer specifika samlingstyper i <code>scala.collection</code> . . . . .	.306
9.1.4	Några oföränderliga och förändringsbara sekvenssamlingar . . . . .	.307
9.1.5	Några användbara metoder på samlingar . . . . .	.308
9.1.6	Repetition: Vad är en sekvens? . . . . .	.308
9.1.7	En sträng är också en <code>IndexedSeq[Char]</code> . . . . .	.309
9.1.8	Konvertera mellan olika samlingstyper . . . . .	.309
9.1.9	Vad är en mängd? . . . . .	.309
9.1.10	Exempel: Oföränderlig mängd. . . . .	.310
9.1.11	Mysteriet med de försvunna elementen . . . . .	.310
9.1.12	Exempel: Förändringsbar mängd . . . . .	.311
9.1.13	Vad är en nyckel-värde-tabell?. . . . .	.311
9.1.14	Den fantastiska nyckel-värde-tabellen <code>Map</code> . . . . .	.311
9.1.15	Exempel: Oföränderlig nyckel-värde-tabell . . . . .	.312
9.1.16	Fler exempel nyckel-värde-tabell . . . . .	.312
9.1.17	Från sekvens av par till tabell. . . . .	.313
9.1.18	Övning: Implementera en <code>Multimap</code> . . . . .	.313
9.1.19	Lösning: <code>Multimap</code> . . . . .	.313
9.1.20	Speciella metoder på förändringsbara samlingar . . . . .	.314
9.1.21	Övning: Förändringsbar lokalt, returnera oföränderlig . . . . .	.314

9.1.22	Övning: Förändringsbar lokalt, returnera oföränderlig . . . . .	.314
9.1.23	Lösning: Förändringsbar lokalt, returnera oföränderlig . . . . .	.315
9.1.24	Metoden <code>sliding</code> . . . . .	.315
9.1.25	Metoderna <code>zipWithIndex</code> , <code>groupBy</code> . . . . .	.316
9.1.26	Fler användbara samlingsmetoder. . . . .	.316
9.1.27	Serialisering och deserialisering. . . . .	.317
9.1.28	Läsa text från fil och URL. . . . .	.317
9.1.29	Serialisering i modulen <code>introprog.IO</code> . . . . .	.317
9.2	Övning <code>lookup</code> . . . . .	.318
9.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	.318
9.2.2	Extrauppgifter; träna mer . . . . .	.322
9.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.323
9.3	Laboration: <code>words</code> . . . . .	.325
9.3.1	Bakgrund . . . . .	.325
9.3.2	Obligatoriska uppgifter . . . . .	.325
9.3.3	Kontrollfrågor . . . . .	.330
9.3.4	Frivilliga uppgifter . . . . .	.330

## 10 Arv och komposition 333

10.1	Teori . . . . .	.334
10.1.1	Vad är arv? . . . . .	.334
10.1.2	Varför behövs arv? . . . . .	.334
10.1.3	Klassdiagram med UML (Unified Modeling Language) . . . . .	.335
10.1.4	Exempel: Robot som bastyp för två subtyper . . . . .	.335
10.1.5	Alternativ till arv: komposition . . . . .	.335
10.1.6	Exempel på komposition i <code>snake-labben</code> . . . . .	.336
10.1.7	Behovet av gemensam bastyp . . . . .	.336
10.1.8	Skapa en gemensam bastyp. . . . .	.336
10.1.9	Skapa en gemensam bastyp med <code>trait</code> och <code>extends</code> . . . . .	.337
10.1.10	En gemensam bastyp med gemensamma delar . . . . .	.337
10.1.11	Placera gemensamma delar i bastypen. . . . .	.338
10.1.12	Scalas typhierarki . . . . .	.338
10.1.13	Implicita supertyper till dina egna klasser . . . . .	.338
10.1.14	Vad är en <code>trait</code> ? . . . . .	.339
10.1.15	Vad används en <code>trait</code> till? . . . . .	.339
10.1.16	En <code>trait</code> kan ha abstrakta medlemmar . . . . .	.340
10.1.17	En <code>trait</code> kan ha parametrar . . . . .	.340
10.1.18	Abstrakta och konkreta medlemmar . . . . .	.340
10.1.19	Undvika kodduplicering med hjälp av arv . . . . .	.341
10.1.20	Varför kan kodduplicering orsaka problem? . . . . .	.341
10.1.21	Överskuggning. . . . .	.342
10.1.22	En final medlem kan ej överskuggas . . . . .	.342
10.1.23	<code>Protected</code> ger synlighet begränsad till subtyper . . . . .	.343

10.1.24	Filnamnsregler och -konventioner . . . . .	.343
10.1.25	Klasser, arv och klassparametrar . . . . .	.343
10.1.26	Statisk och dynamisk typ . . . . .	.344
10.1.27	Inmixning . . . . .	.344
10.1.28	asInstanceOf och asInstanceOf . . . . .	.345
10.1.29	Terminologi: Polymorfism och dynamisk bindning. . . . .	.345
10.1.30	Anonym klass . . . . .	.346
10.1.31	Hur förhindra subtypning? . . . . .	.346
10.1.32	Förseglade typer med sealed . . . . .	.347
10.1.33	Öppen klass signalerar uppmuntrad subtypning . . . . .	.347
10.1.34	Trait eller abstrakt klass?. . . . .	.348
10.1.35	En trait får ej vidarebefordra parametrar . . . . .	.348
10.1.36	Medlemmar, arv och överskuggning . . . . .	.348
10.1.37	Fördjupning: Regler för överskuggning i Scala . . . . .	.349
10.1.38	Fördjupning: Överskugga var med var. . . . .	.349
10.1.39	Fördjupning: Överskugga def med var. . . . .	.350
10.1.40	Att skilja på mitt och ditt med super. . . . .	.350
10.1.41	Fördjupning: Intersektionstyp. . . . .	.351
10.1.42	Fördjupning: Typunioner med eller-operator . . . . .	.351
10.1.43	Fördjupning: Transparent trait . . . . .	.352
10.1.44	Terminologi och nyckelord vid arv . . . . .	.353
10.1.45	Vad är en algebraisk datatyp? . . . . .	.353
10.1.46	En case-klass är en produkt. . . . .	.354
10.1.47	Algebraisk datatyp, kombinerad produkt och summa . . . . .	.354
10.1.48	Algebraisk datatyp med typparameter . . . . .	.355
10.2	Övning inheritance . . . . .	.356
10.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	.356
10.2.2	Extrauppgifter; träna mer . . . . .	.361
10.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.362
10.3	Grupplaboration: snake0. . . . .	.367
10.3.1	Bakgrund . . . . .	.367
10.3.2	Obligatoriska funktionella krav . . . . .	.368
10.3.3	Obligatoriska design-krav. . . . .	.369
10.3.4	Valbara krav – varje person ska välja minst ett. . . . .	.373
10.3.5	Tips och förslag . . . . .	.373

## **11 Kontextuella abstraktioner och varians 377**

11.1	Teori . . . . .	.378
11.1.1	Repetition: Typparametrar och generiska strukturer . . . . .	.378
11.1.2	Olika sätt att begränsa typer . . . . .	.378
11.1.3	Övre och undre typgräns . . . . .	.378
11.1.4	Exempel på övre och undre typgräns. . . . .	.379
11.1.5	Vad är varians? . . . . .	.379

11.1.6	Varför behövs varians? . . . . .	.380
11.1.7	Kovarians (eng. <i>covariance</i> ) . . . . .	.380
11.1.8	Kontravarians . . . . .	.381
11.1.9	Kontravarians (eng. <i>contravariance</i> ) . . . . .	.381
11.1.10	Variansproblem – tack kompilatorn! . . . . .	.382
11.1.11	När använda vilken slags varians? . . . . .	.382
11.1.12	Typjoker (eng. <i>wildcard type</i> ) . . . . .	.383
11.1.13	Mer om varians för den nyfikne . . . . .	.383
11.1.14	Typbegränsning med egentyp (eng. <i>self type</i> ) . . . . .	.384
11.1.15	Sammanhanget är avgörande när du kodar! . . . . .	.384
11.1.16	Repetition: default-argument . . . . .	.385
11.1.17	Repetition: uppdelade parameterlistor . . . . .	.385
11.1.18	Givna värden som alternativ till default-argument . . . . .	.385
11.1.19	Går det lika bra att ha en variabel som default-värde? . . . . .	.386
11.1.20	Kontextuella abstraktioner med <i>given</i> , <i>using</i> . . . . .	.386
11.1.21	Framkalla värde med <i>summon</i> . . . . .	.387
11.1.22	Prioritetsordning vid framkallning av givna värden . . . . .	.387
11.1.23	Ad hoc polymorfism med typklasser . . . . .	.387
11.1.24	Hur få typklassen <i>Parser</i> att funka för fler typer? . . . . .	.388
11.1.25	Namnet på kontextparametrar kan utelämnas . . . . .	.388
11.1.26	Kontextgräns . . . . .	.389
11.1.27	Ännu smidigare typklass med <i>extensionsmetod</i> . . . . .	.389
11.1.28	Sortera samlingar med <i>given</i> ordning . . . . .	.389
11.1.29	Sortera samlingar med ännu smidigare <i>given</i> ordning. . . . .	.390
11.1.30	Vad är ett bra api? . . . . .	.390
11.1.31	Api-desgin med <i>Scala</i> . . . . .	.391
11.2	Övning context . . . . .	.392
11.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	.392
11.2.2	Extrauppgifter; träna mer . . . . .	.393
11.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.394
11.3	Laboration: <i>snake1</i> . . . . .	.399
11.3.1	Redovisning av grupplabb. . . . .	.399

## 12 Valfri fördjupning, Projekt 400

12.1	Projektuppgift . . . . .	.401
12.1.1	Om din avslutande projektuppgift . . . . .	.401
12.1.2	Projektuppgifter . . . . .	.401
12.1.3	Skapa dokumentation . . . . .	.401
12.1.4	Dokumentationskommentarer. . . . .	.402
12.2	Övning extra . . . . .	.403
12.2.1	Uppgifter om sökning och sortering . . . . .	.403
12.2.2	Uppgifter om trådar och jämlöpande exekvering . . . . .	.410

12.3	Projektoppgift: bank . . . . .	.419
12.3.1	Fokus . . . . .	.419
12.3.2	Bakgrund . . . . .	.419
12.3.3	Krav . . . . .	.419
12.3.4	Design . . . . .	.421
12.3.5	Tips. . . . .	.422
12.3.6	Obligatoriska uppgifter . . . . .	.423
12.3.7	Frivilliga extrauppgifter . . . . .	.423
12.3.8	Exempel på historikfil . . . . .	.424
12.3.9	Exempel på körning av programmet . . . . .	.424
12.4	Projektoppgift: music. . . . .	.427
12.4.1	Bakgrund . . . . .	.427
12.4.2	Domänmodell . . . . .	.427
12.4.3	Valfria uppgifter . . . . .	.434
12.5	Projektoppgift: photo. . . . .	.435
12.5.1	Bakgrund . . . . .	.435
12.5.2	Förberedelser . . . . .	.435
12.5.3	Matris med värden av typen Short . . . . .	.435
12.5.4	Användargränssnitt . . . . .	.436
12.5.5	Filter . . . . .	.438
12.5.6	Frivilliga extrauppgifter . . . . .	.444

## **13 Repetition** **446**

13.1	Tips. . . . .	.447
13.1.1	Repetitionsämnen på begäran från tidigare år . . . . .	.447
13.1.2	Några extra önskemål från tidigare år (i mån av tid) . . . . .	.447
13.1.3	Några tumregler/tips vid val av abstraktion . . . . .	.447
13.1.4	Tips om val av samling . . . . .	.448
13.1.5	Före tentan: . . . . .	.448
13.1.6	På tentan: . . . . .	.448
13.2	Övning examprep. . . . .	.450

## **14 MUNTligt PROV** **451**

## **III Appendix** **452**

### **A Kojo** **453**

A.1	Vad är Kojo? . . . . .	.453
A.2	Använda grafikbiblioteket i Kojo. . . . .	.454
A.3	Kojo Desktop . . . . .	.455
A.4	Kojo i Webbläsaren . . . . .	.455
A.5	Mer om Kojo . . . . .	.455

<b>B</b>	<b>Terminalfönster</b>	<b>458</b>
B.1	Vad är ett terminalfönster? . . . . .	458
B.2	Vad är en path/sökväg? . . . . .	460
B.3	Några viktiga terminalkommando . . . . .	461
<b>C</b>	<b>Editera, kompilera och exekvera</b>	<b>462</b>
C.1	Vad är en editor? . . . . .	462
C.1.1	Välj editor . . . . .	463
C.2	Vad är en kompilator? . . . . .	463
C.3	Java JDK . . . . .	465
C.3.1	Kontrollera om du har JDK installerat. . . . .	465
C.3.2	Installera JDK. . . . .	466
C.4	Scala . . . . .	466
C.4.1	Installera Scala . . . . .	466
C.4.2	Scala Read-Evaluate-Print-Loop (REPL) . . . . .	466
C.4.3	Kompilera och kör med Scala Command Line Interface . . . . .	467
<b>D</b>	<b>Fixa buggar</b>	<b>469</b>
D.1	Vad är en bugg? . . . . .	469
D.1.1	Olika sorters fel . . . . .	470
D.2	Att förebygga fel . . . . .	473
D.3	Vad är debugging? . . . . .	475
D.3.1	Hur hittas felorsaken? . . . . .	475
D.4	Åtgärda fel . . . . .	476
D.5	Använda en debugger. . . . .	476
<b>E</b>	<b>Dokumentation</b>	<b>478</b>
E.1	Vad gör ett dokumentationsverktyg? . . . . .	479
E.2	scaladoc . . . . .	479
E.2.1	Använda dokumentation från scaladoc . . . . .	479
E.2.2	Skriva dokumentationskommentarer . . . . .	484
E.2.3	Generera dokumentation . . . . .	485
<b>F</b>	<b>Byggverktyg</b>	<b>486</b>
F.1	Vad gör ett byggverktyg? . . . . .	486
F.2	Scala Command Line Interface <code>scala-cli</code> . . . . .	488
F.2.1	Grundläggande byggfunktioner i Scala CLI. . . . .	488
F.2.2	Använda optioner för att styra Scala CLI. . . . .	489
F.2.3	Generera dokumentation med Scala CLI . . . . .	490
F.2.4	Paketering av exekverbar fil med Scala CLI . . . . .	490
F.2.5	Optioner som användningsdirektiv i ”magiska” kommentarer . . . . .	490
F.3	Scala Build Tool <code>sbt</code> . . . . .	491
F.3.1	Installera <code>sbt</code> . . . . .	491
F.3.2	Använda <code>sbt</code> . . . . .	491



<b>G</b>	<b>Versionshantering och kodlagring</b>	<b>495</b>
G.1	Vad är versionshantering? . . . . .	495
G.2	Versionshanteringsverktyget Git . . . . .	495
G.2.1	Installera git. . . . .	496
G.2.2	Anpassa Git . . . . .	497
G.2.3	Använda git . . . . .	497
G.3	Kodlagringsplatser på nätet. . . . .	499
<b>H</b>	<b>Integrerad utvecklingsmiljö</b>	<b>501</b>
H.1	Vad är en integrerad utvecklingsmiljö? . . . . .	501
H.2	Visual Studio Code med tillägget Scala Metals . . . . .	502
H.2.1	Installera VS Code och Metals. . . . .	502
H.2.2	Köra program i VS Code . . . . .	502
H.2.3	Använda debuggern i VS Code . . . . .	504
H.3	JetBrains IntelliJ IDEA med Scala-plugin . . . . .	505
H.3.1	Installera IntelliJ IDEA . . . . .	507
<b>I</b>	<b>Skapa webb-appar med ScalaJS</b>	<b>508</b>
<b>J</b>	<b>Introduktion till Java</b>	<b>509</b>
J.1	Teori . . . . .	510
J.1.1	Övning scalajava och labb javatext . . . . .	510
J.1.2	”Hello world!” i Java. . . . .	510
J.1.3	Testa Java i jshell . . . . .	510
J.1.4	Grundläggande likheter och skillnader Java–Scala . . . . .	511
J.1.5	Huvudprogram i Scala och Java . . . . .	511
J.1.6	Loopa genom argumenten i ett Java-huvudprogram. . . . .	511
J.1.7	HIGHSCORE implementerad i Java . . . . .	512
J.1.8	Några saker som finns i Scala men inte i Java . . . . .	512
J.1.9	Några saker som finns i Java men inte i Scala . . . . .	513
J.1.10	Begränsningar med funktionsprogrammering i Java . . . . .	513
J.1.11	Grundtyper i Scala och primitiva typer Java . . . . .	514
J.1.12	Javas switch-sats. . . . .	514
J.1.13	Javas switch-sats utan break . . . . .	515
J.1.14	Javas switch-sats med glömd break . . . . .	516
J.1.15	Syntax för variabeldeklaration i Scala och Java. . . . .	517
J.1.16	For-sats i Scala och Java . . . . .	517
J.1.17	For-sats i Scala med indices . . . . .	518
J.1.18	For-satser och arrayer i Java . . . . .	518
J.1.19	Implementation av SEQ-COPY i Java med for-sats. . . . .	519
J.1.20	Element för element med speciell for-each-sats i Java . . . . .	519
J.1.21	Typisk utformning av Java-klass . . . . .	519
J.1.22	Statiska medlemmar i Java . . . . .	520
J.1.23	Exempel: oföränderlig klass i Scala och Java . . . . .	520

J.1.24	Exempel: Scala-klassen Complex . . . . .	.521
J.1.25	Exempel: Motsvarande Java-klassen JComplex . . . . .	.521
J.1.26	Exempel: Använda JComplex i Scala-kod. . . . .	.522
J.1.27	Exempel: Använda JComplex i Java-kod . . . . .	.523
J.1.28	Exempel: Förändringsbar klass i Scala och Java . . . . .	.523
J.1.29	Scalas "case-klass-godis" finns inte i Java . . . . .	.525
J.1.30	Repetition: Den primitiva typen Array i JVM . . . . .	.525
J.1.31	Syntax för Array i Scala och Java . . . . .	.525
J.1.32	Exempel: Polygon med primitiv array i Java . . . . .	.526
J.1.33	Polygon med primitiv array i Java: stoppa in sist och vid behov skapa mer plats . . . . .	.526
J.1.34	Polygon med primitiv array i Java: stoppa in mitt i på angiven plats . .	.527
J.1.35	Scanna filer och strängar med <code>java.util.Scanner</code> . . . . .	.527
J.1.36	Exempel: Scanner . . . . .	.528
J.1.37	Använda Java-samlingar i Scala med <code>CollectionConverters</code> . . . . .	.528
J.1.38	Generiska samlingar i Java . . . . .	.528
J.1.39	Om <code>ArrayList</code> i Java . . . . .	.529
J.1.40	Polygon med <code>ArrayList</code> i Java . . . . .	.529
J.1.41	Några viktiga operationer på <code>ArrayList&lt;E&gt;</code> . . . . .	.529
J.1.42	Övning <code>ArrayList</code> : new och add . . . . .	.530
J.1.43	For-each-sats i Java: . . . . .	.530
J.1.44	Polygon med <code>ArrayList</code> : metoderna blir enklare. . . . .	.531
J.1.45	Polygon med <code>ArrayList</code> : iterera över alla hörnpunkter i draw med indexe- ring . . . . .	.531
J.1.46	Polygon med <code>ArrayList</code> : iterera över alla hörnpunkter i draw med <code>foreach-</code> sats . . . . .	.531
J.1.47	Övning <code>ArrayList</code> : implementera metoden <code>hasVertex</code> . . . . .	.532
J.1.48	Lösning <code>ArrayList</code> : implementera metoden <code>hasVertex</code> . . . . .	.532
J.1.49	For-each-sats med array . . . . .	.532
J.1.50	Generiska klasser (t.ex. <code>ArrayList</code> ) med primitiva typer . . . . .	.533
J.1.51	Wrapper-klassen <code>Integer</code> . . . . .	.533
J.1.52	Wrapper-klasser i <code>java.lang</code> . . . . .	.534
J.1.53	Övning: primitiva versus inpackade typer . . . . .	.534
J.1.54	Exempel: Lista med heltal utan autoboxning . . . . .	.534
J.1.55	Specialregler för wrapper-klasser . . . . .	.535
J.1.56	Exempel: Lista med heltal och autoboxning. . . . .	.535
J.1.57	Fallgropar vid autoboxning . . . . .	.536
J.1.58	Referenslikhet eller innehållslikhet i Scala och Java . . . . .	.536
J.1.59	Fallgrop med samlingar: metoden <code>contains</code> kräver implementation av <code>equals</code> . . . . .	.536
J.1.60	Fullständigt recept för <code>equals</code> . . . . .	.537
J.1.61	Villkorsuttryck i Java. . . . .	.537
J.1.62	Typtest och typkonvertering . . . . .	.537



J.1.63	Regler för överskuggning i Java . . . . .	.537
J.1.64	Fånga undantag i Scala och Java . . . . .	.537
J.1.65	Gränssnittet List i Java . . . . .	.538
J.1.66	Det går inte att skapa generisk Array i Java . . . . .	.538
J.1.67	Jämföra strängar i Java . . . . .	.539
J.1.68	Jämföra strängar i Java: exempel . . . . .	.539
J.2	Övning java . . . . .	.541
J.2.1	Grunduppgifter; förberedelse inför laboration. . . . .	.541
J.3	Laboration: java . . . . .	.561
J.3.1	Krav . . . . .	.561
J.3.2	Frivilliga extrauppgifter . . . . .	.562
J.3.3	Inspiration och tips. . . . .	.562

## **K Virtuellt maskin 563**

K.1	Vad är en virtuellt maskin? . . . . .	.563
K.2	Vad innehåller kursens vm? . . . . .	.564
K.3	Installera kursens vm . . . . .	.564

## **IV Lösningar 566**

### **L Lösningar till övningarna 567**

L.1	Lösning expressions. . . . .	.568
L.1.1	Grunduppgifter; förberedelse inför laboration. . . . .	.568
L.1.2	Extrauppgifter; träna mer . . . . .	.574
L.1.3	Fördjupningsuppgifter; utmaningar . . . . .	.578
L.2	Lösning programs . . . . .	.582
L.2.1	Grunduppgifter . . . . .	.582
L.2.2	Extrauppgifter; träna mer . . . . .	.587
L.2.3	Fördjupningsuppgifter; utmaningar . . . . .	.589
L.3	Lösning functions. . . . .	.591
L.3.1	Extrauppgifter; träna mer . . . . .	.593
L.3.2	Fördjupningsuppgifter; utmaningar . . . . .	.595
L.4	Lösning objects . . . . .	.598
L.4.1	Grunduppgifter; förberedelse inför laboration. . . . .	.598
L.4.2	Extrauppgifter; träna mer . . . . .	.605
L.4.3	Fördjupningsuppgifter; utmaningar . . . . .	.606
L.5	Lösning classes . . . . .	.609
L.5.1	Grunduppgifter; förberedelse inför laboration. . . . .	.609
L.5.2	Extrauppgifter; träna mer . . . . .	.612
L.5.3	Fördjupningsuppgifter; utmaningar . . . . .	.615
L.6	Lösning patterns . . . . .	.621
L.6.1	Grunduppgifter; förberedelse inför laboration. . . . .	.621
L.6.2	Fördjupningsuppgifter; utmaningar . . . . .	.627

L.7	Lösning sequences . . . . .	.631
L.7.1	Grunduppgifter; förberedelse inför laboration. . . . .	.631
L.7.2	Extrauppgifter; träna mer . . . . .	.639
L.7.3	Fördjupningsuppgifter; utmaningar . . . . .	.642
L.8	Lösning matrices . . . . .	.645
L.8.1	Grunduppgifter; förberedelse inför laboration. . . . .	.645
L.8.2	Extrauppgifter; träna mer . . . . .	.648
L.8.3	Fördjupningsuppgifter; utmaningar . . . . .	.650
L.9	Lösning lookup . . . . .	.652
L.9.1	Grunduppgifter; förberedelse inför laboration. . . . .	.652
L.9.2	Extrauppgifter; träna mer . . . . .	.655
L.9.3	Fördjupningsuppgifter; utmaningar . . . . .	.656
L.10	Lösning inheritance. . . . .	.657
L.10.1	Grunduppgifter; förberedelse inför laboration. . . . .	.657
L.10.2	Extrauppgifter; träna mer . . . . .	.662
L.10.3	Fördjupningsuppgifter; utmaningar . . . . .	.662
L.11	Lösning context . . . . .	.665
L.11.1	Grunduppgifter; förberedelse inför laboration. . . . .	.665
L.11.2	Extrauppgifter; träna mer . . . . .	.668
L.11.3	Fördjupningsuppgifter; utmaningar . . . . .	.668
L.12	Lösning extra . . . . .	.671
L.12.1	Uppgifter om sökning och sortering . . . . .	.671
L.12.2	Uppgifter om trådar och jämlöpande exekvering . . . . .	.676
L.12.3	Extrauppgifter; träna mer . . . . .	.677
L.13	Lösning examprep . . . . .	.681
L.14	Lösning java . . . . .	.682
L.14.1	Grunduppgifter; förberedelse inför laboration. . . . .	.682

# **Del I**

## **Om kursen**

## Kapitel -2

# Kursens arkitektur

### -2.0.1 Veckoöversikt

<i>W</i>	<i>Modul</i>	<i>Övn</i>	<i>Lab</i>
W01	Introduktion	expressions	kojo
W02	Program och kontrollstrukturer	programs	–
W03	Funktioner och abstraktion	functions	irritext
W04	Objekt och inkapsling	objects	blockmole
W05	Klasser och datamodellering	classes	blockbattle0
W06	Mönster och felhantering	patterns	blockbattle1
W07	Sekvenser och enumerationer	sequences	shuffle
KS	KONTROLLSKRIVN.	–	–
W08	Nästlade och generiska strukturer	matrices	life
W09	Mängder och tabeller	lookup	words
W10	Arv och komposition	inheritance	snake0
W11	Kontextuella abstraktioner och varians	context	snake1
W12	Valfri fördjupning, Projekt	extra	Projekt0
W13	Repetition	examprep	Projekt1
W14	MUNTligt PROV	Munta	Munta
T	VALFRI TENTAMEN	–	–

Kursen består av en **modul** per läsvecka med två **föreläsningar**, en **övning** och en **laboration** (förutom några veckor som saknar labb och/eller övning eller har annan aktivitet, se veckoöversikt). Föreläsningarna ger en översikt av den teori som ingår i varje modul. Genom att göra övningarna bearbetar du teorin och föreber dig inför laborationerna. När du klarat övningen och laborationen i en modul är du redo att gå vidare till nästa. Tabellen på nästa uppslag visar begrepp som ingår i varje modul.

Kursen är uppdelad i två läsperioder. Efter första läsperioden gör du en diagnostisk **kontrollskrivning** som kontrollerar ditt kunskapsläge. Andra läsperioden avslutas med ett större **projekt**, en muntlig tentamen och en valfri skriftlig **tentamen**.

W01	<b>Introduktion</b>	sekvens, alternativ, repetition, abstraktion, editera, kompilera, exekvera, datorns delar, virtuell maskin, litteral, värde, uttryck, identifierare, variabel, typ, tilldelning, namn, val, var, def, definiera och anropa funktion, funktionshuvud, funktionskropp, procedur, inbyggda grundtyper, println, typen Unit, enhetsvärdet (), stränginterpolatorn s, aritmetik, slumptal, logiska uttryck, de Morgans lagar, if, true, false, while, for
W02	<b>Program och kontrollstrukturer</b>	huvudprogram, program-argument, indata, scala.io.StdIn.readLine, kontrollstruktur, iterera över element i samling, for-uttryck, yield, map, foreach, samling, sekvens, indexering, Array, Vector, intervall, Range, algoritm, implementation, pseudokod, algoritmexempel: SWAP, SUM, MIN-MAX, MIN-INDEX
W03	<b>Funktioner och abstraktion</b>	abstraktion, funktion, parameter, argument, returtyp, default-argument, namngivna argument, parameterlista, funktionshuvud, funktionskropp, applicera funktion på alla element i en samling, uppdelad parameterlista, skapa egen kontrollstruktur, funktionsvärde, funktionstyp, äkta funktion, stegad funktion, apply, anonyma funktioner, lambda, predikat, aktiveringspost, anropsstacken, objektheapen, stack trace, värdeandrop, namnanrop, klammerparentes och kolon vid ensam parameter, rekursion, scala.util.Random, slumptalsfrö
W04	<b>Objekt och inkapsling</b>	modul, singelobjekt, punktnotation, tillstånd, medlem, attribut, metod, paket, filstruktur, jar, dokumentation, JDK, import, selektiv import, namnbyte vid import, export, tupel, multipla returvärden, block, lokal variabel, skuggning, lokal funktion, funktioner är objekt med apply-metod, namnrymd, synlighet, privat medlem, inkapsling, getter och setter, principen om enhetlig access, överlagring av metoder, introprog.PixelWindow, initialisering, lazy val, typalias
W05	<b>Klasser och datamodellering</b>	applikationsdomän, datamodell, objektorientering, klass, instans, Any, isInstanceOf, toString, new, null, this, accessregler, private, private[this], klassparameter, primär konstruktor, fabriksmetod, alternativ konstruktor, förändringsbar, oföränderlig, case-klass, kompanjonsobjekt, referenslikhet, innehållslikhet, eq, ==
W06	<b>Mönster och felhantering</b>	mönstermatchning, match, Option, throw, try, catch, Try, unapply, sealed, flatten, flatMap, partiella funktioner, collect, wildcard-mönster, variabelbindning i mönster, sekvens-wildcard, bokstavliga mönster, implementera equals, hashCode

W07	<b>Sekvenser och enumerationer</b>	översikt av Scalas samlingsbibliotek och samlingsmetoder, klasshierarkin i scala.collection, Iterable, Seq, List, ListBuffer, ArrayBuffer, WrappedArray, sekvensalgoritm, algoritm: SEQ-COPY, in-place vs copy, algoritm: SEQ-REVERSE, registrering, algoritm: SEQ-REGISTER, linjärsökning, algoritm: LINEAR-SEARCH, tidskomplexitet, minneskomplexitet, översikt strängmetoder, StringBuilder, ordning, inbyggda sökmetoder, find, indexOf, indexWhere, inbyggda sorteringsmetoder, sorted, sortWith, sortBy, repeterade parametrar
KS	KONTROLLSKRIVN.	
W08	<b>Nästlade och generiska strukturer</b>	matris, nästlad samling, nästlad for-sats, typparameter, generisk funktion, generisk klass, fri och bunden typparameter, generiska datastrukturer, generiska samlingar i Scala
W09	<b>Mängder och tabeller</b>	innehållstest, mängd, Set, mutable.Set, nyckel-värde-tabell, Map, mutable.Map, hash code, java.util.HashMap, java.util.HashSet, persistens, serialisering, textfiler, Source.fromFile, java.nio.file
W10	<b>Arv och komposition</b>	arv, komposition, polymorfism, trait, extends, asInstanceOf, with, inmixning supertyp, subtyp, bastyp, override, Scalas typhierarki, Any, AnyRef, Object, AnyVal, Null, Nothing, topptyp, bottenotyp, referenstyper, värdetyper, accessregler vid arv, protected, final, trait, abstrakt klass
W11	<b>Kontextuella abstraktioner och varians</b>	övre- och undre typgräns, varians, kontravarians, kovarians, typjoker, egentyp, givet värde (given), kontextparameter (using), generiska extensionsmetoder, ad hoc polymorfism, kontextgräns, typklass, api, kodläsbarhet, granskningar
W12	<b>Valfri fördjupning, Projekt</b>	välj valfritt fördjupningsområde, påbörja projekt
W13	<b>Repetition</b>	träna på extensor, redovisa projekt, träna inför muntligt prov
W14	MUNTligt PROV	
T	VALFRI TENTAMEN	

## -2.1 Om ditt lärande

### -2.1.1 Vad lär du dig?

- Grundläggande principer för programmering:  
Sekvens, Alternativ, Repetition, Abstraktion (SARA)  
⇒ Inga förkunskaper i programmering krävs!
  - Implementation av algoritmer
  - Tänka i abstraktioner, dela upp problem i delproblem
  - Förståelse för flera olika angreppssätt:
    - **imperativ programmering**
    - **objektorientering**
    - **funktionsprogrammering**
  - Det moderna programmeringsspråket **Scala**
  - Utvecklingsverktyg (editor, kompilator, utvecklingsmiljö)
  - Implementera, granska, testa, felsöka
- 

### -2.1.2 Progression

Kursens koncept avancerar steg för steg:

- Kontrollstrukturer
- Funktioner
- Objekt
- Datastrukturer
- Algoritmer
- Nästlade strukturer
- Avancerade abstraktionsmekanismer
  - Komposition
  - Polymorfism
  - Kontextuella abstraktioner

Vi itererar över koncepten och fördjupar förståelsen efter hand.

---

### -2.1.3 Hur lär du dig?

- Genom praktiskt **eget arbete: Lära genom att göra!**
    - Övningar: applicera koncept på olika sätt
    - Laborationer: kombinera flera koncept till en helhet
  - Genom studier av kursens teori: **Skapa förståelse!**
  - Genom samarbete med dina kurskamrater: **Gå djupare!**
-

Kompendiet är den huvudsakliga kurslitteraturen och definierar kursinnehållet. Föreläsningar, övningar och laborationer i kompendiet är kursens primära kunskapskällor, tillsammans med de öppna resurser på nätet som kompendiet hänvisar till. Kompendiet är öppen källkod och du välkomnas varmt att bidra!

Om du gärna vill ha en eller flera mer traditionella läroböcker som bredvidläsning rekommenderas följande:

- För de som aldrig kodat, och vill läsa om kodning från grunden:
  - "Introduction to Programming and Problem-Solving Using Scala" Second Edition (2016), Mark C. Lewis, Lisa Lacher.
  - Lewis & Lacher täcker stora delar av kursen, men innehåller även en del material som ingår i senare LTH-kurser. Ordningen är ganska annorlunda, men det går bra att läsa boken i en annan ordning än den är skriven.
- För de som redan kodat en hel del i ett objektorienterat språk:
  - "Programming in Scala", Fifth Edition (2021), Martin Odersky, Lex Spoon, and Bill Venners.
  - Martin Odersky är upphovspersonen bakom Scala och denna välskrivna bok innehåller en komplett genomgång av Scala-språket med många exempel och tips. "Fifth Edition" täcker nya Scala 3. Boken riktar sig till de som redan har kunskap om något objektorienterat språk, t.ex. Java eller C#. Det finns ett bra index som gör det lätt att anpassa din läsning efter kursens upplägg. Bokens ca 800 sidor innehåller mycket material som är på en mer avancerad nivå än denna kurs, men du kommer att ha nytta av innehållet i kommande kurser.

Dessa läroböcker följer inte direkt kursens upplägg vad gäller omfång och progression och du får själv göra den nyttiga hemläxan att koppla deras innehåll till det vi går igenom i kursens olika moduler.

#### -2.1.4 Kursmoment — varför?

- **Föreläsningar**: skapa översikt, ge struktur, förklara teori, svara på frågor, motivera varför.
- **Övningar**: bearbeta teorin steg för steg, **grundövningar** för alla, **extraövningar** om du vill/behöver öva mer, **fördjupningsövningar** om du vill gå djupare; **förberedelse inför laborationerna**.
- **Laborationer**: **obligatoriska**, sätta samman teorins delar i ett större program; lösningar redovisas för handledare; gk på alla för att få tenta.
- **Resurstider**: få hjälp med övningar och laborationsförberedelser av handledare, fråga vad du vill.
- **Samarbetsgrupper**: grupplärande genom samarbete, hjälpa varandra.
- **Kontrollskrivning**: **obligatorisk**, diagnostisk, kamraträttad; kan ge samarbetsbonuspoäng till valfria tentan.
- **Individuell projektuppgift**: **obligatorisk**, du visar att du kan skapa ett större program självständigt; redovisas för handledare.
- **Muntligt prov**: **obligatoriskt**, ska klaras för godkänt på kursen; du visar att du har tillräcklig förståelse för kursens koncept för att klara nästa kurs.
- **Tentamen**: Valfri för överbetyg men alla uppmuntras att försöka; skriftlig, enda hjälpmedel: snabbreferensen <http://cs.lth.se/pgk/quickref>



### -2.1.5 En typisk kursvecka

1. Gå på **föreläsningar** på **måndag-tisdag**
2. **Jobba individuellt** med teori, övningar, labbförberedelser på **måndag-torsdag**
3. **Träffas** regelbundet i **samarbetsgruppen** och hjälp varandra att förstå mer och fördjupa lärandet, förslagsvis på återkommande tider varje vecka då alla i gruppen kan
4. Kom till **resurstiderna** och få hjälp och tips av handledare och kurskamrater på **onsdag-torsdag**
5. Genomför den obligatoriska **laborationen** på **fredag**

Se detaljerna och undantagen i schemat: [cs.lth.se/pgk/schema](https://cs.lth.se/pgk/schema)

---

# Kapitel -1

## Anvisningar

Detta kapitel innehåller anvisningar och riktlinjer för kursens olika delar. Läs noga så att du inte missar viktig information om syftet bakom kursmomenten och vad som förväntas av dig.

### -1.1 Samarbetsgrupper

Ditt lärande i allmänhet, och ditt programmeringslärande i synnerhet, fördjupas om det sker i dialog med andra. Dessutom är din samarbetsförmåga och din pedagogiska förmåga avgörande för din framgång som professionell systemutvecklare. Därför är kursdeltagarna indelade i *samarbetsgrupper* om 4-6 personer där medlemmarna samverkar för att alla i gruppen ska nå så långt som möjligt i sina studier.

För att hantera och dra nytta av skillnader i förkunskaper är samarbetsgrupperna indelade så att deltagarna har *varierande förkunskaper* baserat på en förkunskapsenkät. De som redan har provat på att programmera får då chansen att träna på sin pedagogiska förmåga som är så viktig för systemutvecklare, medan de som ännu inte kommit lika långt kan dra nytta av gruppmedlemmarnas samlade kompetens i sitt lärande. Kompetensvariationen i gruppen kommer att förändras under kursens gång, då olika individer lär sig olika snabbt i olika skeden av sitt lärande; de som till att börja med har ett försprång kanske senare får kämpa för att komma över en viss lärandetröskel.

Samarbetsgrupperna organiserar själva sitt arbete och varje grupp får finna de samarbetsformer som passar medlemmarna bäst. Här följer några erfarenhetsbaserade tips:

1. Träffas så fort som möjligt i hela gruppen och lär känna varandra. Ju snabbare ni kommer samman som grupp och får den sociala interaktionen att fungera desto bättre. Ni kommer att ha nytta av denna investering under hela terminen och kanske under resten av er studietid.
2. Kom överens om stående mötestider och mötesplatser. Det är viktigt med kontinuiteten i arbetet för att samarbetet i gruppen ska utvecklas och fördjupas. Träffas minst en gång i veckan. Ha en stående agenda, t.ex. en runda runt bordet där var och en berättar hur långt hen kommit och listar de begreppen som hen för tillfället behöver fokusera på.
3. Hjälp åt att tillsammans identifiera och diskutera era olika individuella studiebehov och studieambitioner. När man ska lära sig att programmera stöter man på olika lärandetrösklar som man kan få hjälp att ta sig över av någon som redan är förbi tröskeln. Men det gäller då för den som hjälper att först förstå exakt vad det är som är svårt, eller vilka specifika pusselbitar som saknas, för att på bästa sätt kunna

underlätta för en medstudent att ta sig över tröskeln. Det gäller att hjälpa *lagom* mycket så att var och en självständigt får chansen att skriva sin egen kod.

4. Var en schysst kamrat och agera professionellt, speciellt i situationer där gruppdeltagarna vill olika. Kommunicera på ett respektfullt sätt och sök konstruktiva kompromisser. Att utvecklas socialt är viktigt för din framtida yrkesutövning som systemutvecklare och i samarbetsgruppen kan du träna och utveckla din samarbetsförmåga.

### -1.1.1 Samarbetskontrakt

Ni ska upprätta ett samarbetskontrakt redan under första veckan och visa för en handledare. Alla gruppmedlemmarna ska skriva under kontraktet. Handledaren ska också skriva under som bekräftelse på att ni visat kontraktet.

Syftet med kontraktet är att ni ska diskutera igenom i gruppen hur ni vill arbeta och vilka regler ni tycker är rimliga. Ni bestämmer själva vad kontraktet ska innehålla. Nedan finns förslag på punkter som kan ingå i ert kontrakt. En kontraktsmall finns här: <https://github.com/lunduniversity/introprog/blob/master/study-groups/collaboration-contract.tex>

#### **Samarbetskontrakt**

Vi som skrivit under detta kontrakt lovar att göra vårt bästa för att följa samarbetsreglerna nedan, så att alla ska lära sig så mycket som möjligt.

1. Komma i tid till gruppmöten.
2. Vara väl förberedda genom självstudier inför gruppmöten.
3. Hjälpa varandra att förstå, men inte lösa uppgifter åt någon annan.
4. Ha ett respektfullt bemötande även om vi har olika åsikter.
5. Inkludera alla i gemenskapen.
6. ...

### -1.1.2 Grupplaboration

Laboration snake0 i läsvecka W10 är en grupplaboration. Följande anvisningar gäller speciellt för grupplaborationen. (Allmänna anvisningar som gäller för både de individuella laborationerna och grupplaborationer finns i avsnitt -1.5.)

1. Diskutera i din samarbetsgrupp hur ni ska dela upp koden mellan er i flera olika delar, som ni kan arbeta med var för sig. En sådan del kan vara en klass, en trait, ett objekt, ett paket, eller en funktion.
2. Varje del ska ha en **huvudansvarig** individ.
3. Arbetsfördelningen ska vara någorlunda jämnt fördelad mellan gruppmedlemmarna.
4. Den som är huvudansvarig för en viss del redovisar den delen.
5. Ni ska ta fram en gruppgemensam checklista för kodgranskning. Varje gruppmedlem ska granska minst en annan gruppmedlems kod enligt checklistan.

6. Grupplaborationen görs över **två veckor** uppdelat på två delredovisningar. Vid första redovisningen ska arbetsupplägget och pågående utveckling redovisas. Vid andra tillfället ska de färdig lösningarna presenteras av respektive huvudansvarig individ.
7. Vid första redovisningen ska du redogöra för handledaren hur ni delat upp koden och vem som är huvudansvarig för vad och vad ditt ansvar omfattar, samt hur ni jobbar praktiskt med att synkronisera er utveckling.
8. Grupplaborationen är en **extra stor uppgift** och grupparbetet behöver ledtid för att ni ska hinna koordinera er sinsemellan. Du behöver därför planera för att arbeta med något i grupplabben i stort sett varje dag under de tillgängliga veckorna, och vara redo att bidra i diskussioner.

### -1.1.3 Samarbetsbonus

Alla tjänar på att samarbeta och hjälpa varandra i lärandet. Som extra incitament för grupplärande utdelas *samarbetsbonus* baserat på resultatet från den diagnostiska kontrollskrivningen efter halva kursen (se avsnitt -1.6). Bonus ges till varje student enligt gruppmedelvärdet av kontrollskrivningspoängen och räknas ut med funktionen `collaborationBonus` nedan, där `points` är en sekvens med heltal som utgör gruppmedlemmars individuella poäng från kontrollskrivningen.

```
def collaborationBonus(points: Seq[Int]): Int =  
    (points.sum / points.size.toDouble).round.toInt
```

Samarbetsbonusen viktas så att den högsta möjliga bonusen maximalt utgör 5% av maxpoängen på tentan och adderas till det individuella tentaresultatet om du är godkänd på kursens sluttentamen. Samarbetsbonusen kan alltså påverka om du når högre betyg, men den påverkar *inte* om du får godkänt eller ej. Detta gör att alla i gruppen gynnas av att så många som möjligt lär sig på djupet inför kontrollskrivningen. Din eventuella samarbetsbonus räknas dig tillgodo endast vid det första, ordinarie tentamenstillfället.

## -1.2 Föreläsningar

En normal läsperiodsvecka börjar med två föreläsningsspass om 2 timmar vardera. Föreläsningarna ger en översikt av kursens teoretiska innehåll och går igenom innebörden av de begrepp du ska lära dig. Föreläsningarna innehåller många programmeringsexempel och föreläsaren "lajvkodar" då och då för att illustrera den kreativa problemlösningsprocess som ingår i all programmering. Föreläsningarna berör även kursens organisation och olika praktiska detaljer.

På föreläsningarna ges goda möjligheter att ställa allmänna frågor om teorin och att i plenum diskutera specifika svårigheter (individuell lärarhjälp ges på resurstider, se avsnitt -1.4, och på laborationer, se avsnitt -1.5). Även om det är många i föreläsningssalen, *tveka inte att ställa frågor* – det är säkert fler som undrar samma sak som du!

Föreläsningarna är inte obligatoriska, men det är mycket viktigt att du går dit, även om du i perioder känner att du har bra koll på all teori. På föreläsningarna får du en övergripande ämnesstruktur och en konkret programmeringsupplevelse, som du delar med dina kursare och kan diskutera i samarbetsgrupperna. Föreläsningarna ger också en prioritering av materialet och förbereder dig inför examinationen med praktiska råd och tips om hur du bör fokusera dina studier.

## -1.3 Övningar

I en normal läsperiodsvecka ingår en övning med flera uppgifter och deluppgifter. Övningarna utgör basen för dina programmeringsstudier och erbjuder en systematisk genomgång av kursteorins alla delar genom praktiska kodexempel som du genomför steg för steg vid datorn med hjälp av ett interaktivt verktyg som kallas Read-Evaluate-Print-Loop (REPL). Om du gör övningarna i REPL säkerställer du att du skaffar dig tillräcklig förståelse för alla begrepp som ingår i kursen och att du inte missar någon viktigt pusselbit.

Övningarna utgör också förberedelse inför laborationerna. Om du inte gör veckans övning är det inte troligt att du kommer att klara veckans laboration inom rimlig tid.

Dessa två punkter är speciellt viktiga när du ska lära sig att programmera:

- **Programmera!** Det räcker inte med att bara passivt läsa om programmering; du måste *aktivt* själv skriva mycket kod och genomföra egna programmeringsexperiment. Det underlättar stort om du bejakar din nyfikenhet och experimentlusta. Alla programmeringsfel som du gör och alla dina misstag, som i efterhand verkar enkla, är i själva verket oundgängliga steg på vägen och ger avgörande "Aha!"-upplevelser. Kursens övningar är grunden för denna form av lärande.
- **Ha tålamod!** Det är först när du har förmågan att aktivt kombinera *många* olika programmeringskoncept som du själv kan lösa lite större programmeringsuppgifter. Det kan vara frustrerande i början innan du når så långt att din verktygslåda med begrepp är tillräckligt stor för att du ska kunna skapa den kod du vill. Ibland krävs det extra tålamod innan allt plötsligt lossnar. Många programmeringslärare och -studenter vittnar om att "polletten plötsligt trillar ner" och allt faller på plats. Övningarna syftar till att, steg för steg, bygga din verktygslåda så att den till slut blir tillräckligt kraftfull för mer avancerad problemlösning.

Olika studenter har olika ambitionsnivå, skilda förkunskaper, varierande arbetskapa-citet, mer eller mindre välutvecklad studieteknik och olika lätt för att lära sig att programmera. För att hantera denna variation erbjuds övningsuppgifter av tre olika typer:

- **Grunduppgifter.** Varje veckas grunduppgifter täcker basteorin och hjälper dig att säkerställa att du kan gå vidare utan kunskapsluckor. Grunduppgifterna utgör även basen för laborationerna. Alla studenter bör göra alla grunduppgifter. En bra förståelse för innehållet i grunduppgifterna ger goda förutsättningar att klara godkänt betyg på sluttentamen.
- **Extrauppgifter.** Om du upplever att grunduppgifterna är svåra och du vill öva mer, eller om du vill vara säker på att du verkligen befäster dina grundkunskaper, då ska du göra extrauppgifterna. Dessa är på samma nivå som grunduppgifterna och ger extra träning.
- **Fördjupningsuppgifter.** Om du vill gå djupare och har kapacitet att lära dig ännu mer, gör då fördjupningsuppgifterna. Dessa kompletterar grunduppgifterna med mer avancerade exempel och går utöver vad som krävs för godkänt på kursen. Om du satsar på något av de högre betygen ska du göra fördjupningsuppgifterna. Vissa fördjupningsuppgifter har en stjärna i marginalen. Denna symbol visar att uppgiften är allmänbildande, men överkurs och kommer ej på tentamen.



Till varje övning finns lösningar som du hittar längst bak i detta kompendium. Titta *inte* på lösningen innan du själv först försökt lösa uppgiften. Ofta innehåller lösningarna

kommentarer och tips så glöm inte att kolla igenom veckans lösningar innan du börjar förbereda dig inför veckans laboration.

Tänk på att det ofta finns *många olika lösningar* på samma programmeringsproblem, som kan vara likvärdiga eller ha olika fördelar och nackdelar beroende på sammanhanget. Diskutera gärna olika lösningsvarianter med dina kursare och handledare – att prova många olika sätt att lösa en uppgift fördjupar ditt lärande avsevärt!

Många uppgifter lyder ”testa detta i REPL och förklara vad som händer” och svårigheten ligger ofta inte i att skapa själva koden utan att förstå hur den fungerar och *varför*. På detta sätt tränar du ditt programmeringstänkande med hjälp av en växande begreppsapparat. Syftet är ofta att illustrera ett allmängiltigt koncept och det är därför extra bra om du skapar egna övningsuppgifter på samma tema och experimenterar med nya varianter som ger dig ytterligare förståelse.

Övningsuppgifterna innehåller ofta färdiga kodsnuttar som du ska skriva in i REPL medan den kör i ett terminalfönster. REPL-kod visas i övningsuppgifterna med ljus text på mörk bakgrund, så här:

```
1 scala> val msg = "Hello world!"  
2 scala> println(msg)
```

Prompten `scala>` indikerar att REPL är igång och väntar på indata. Du ska skriva den kod som står *efter* prompten. Mer information om hur du använder REPL hittar du i appendix C.4.2.

Även om kompendiet finns tillgängligt för nedladdning, frestas *inte* att klippa ut och klistra in alla kodsnuttar i REPL. Ta dig istället den ringa tiden det tar att skriva in koden rad för rad. Medan du själv skriver hinner du tänka efter, och det egna, aktiva skrivandet främjar ditt lärande och gör det lättare att komma ihåg och förstå.

## -1.4 Resurstider

Under varje läsperiodsvecka finns ett flertal resurstider i schemat. Det finns minst en tid som passar din schemagrupp, men du får gärna gå på andra och/eller flera tider i mån av plats. Resurstiderna är schemalagda i datorsal med Linuxdatorer och i varje sal finns en handledare som är redo att svara på dina frågor.

Följande riktlinjer gäller för resurstiderna:

1. **Syfte.** Resurstiderna är primärt till för att hjälpa dig vidare om du kör fast med övningarna eller laborationsförberedelserna, men du får fråga om vad som helst som rör kursen i den mån handledaren kan svara och hinner med.
2. **Samarbete.** Hjälp gärna varandra under resurstiderna! Om någon kursare kör fast är det utvecklande och lärorikt att hjälpa till. Om schema och plats tillåter kan du gärna gå på samma resurstidstillfälle som någon medlem i din samarbetsgrupp, men ni kan också lika gärna hjälpas åt tvärs över gruppgränserna.
3. **Hänsyn.** När du hjälper andra, tänk på att prata riktigt tyst så att du inte stör andras koncentration. Tänk också på att alla behöver träna mycket själv utan att bli alltför styrda av en ”baksätessförare”. Ta inte över tangentbordet från någon annan; ge hellre välgenomtänkta tips på vägen och låt din kursare behålla kontrollen över uppgiftslösningen.

4. **Fokus.** Du ska *inte* göra och redovisa laborationen på resurstiderna; dessa ska göras och redovisas på laborationstid. Men om du varit sjuk eller ej blivit godkänd på någon enstaka laboration kan du, om handledaren så hinner, be att få redovisa din restlaboration på en resurstid.
5. **Framstegsprotokoll.** På sidan [ii](#) finns ett framstegsprotokoll för övningarna. Håll detta uppdaterat allteftersom du genomför övningarna och visa protokollet när du frågar om hjälp av handledare. Då blir det lättare för handledaren att se vilka kunskaper du förvärvat hittills och anpassa dialogen därefter.

## -1.5 Laborationer

En normal läsperiodsvecka avslutas med en lärarhandledd laboration. Medan övningar tränar teorins olika delar i många mindre uppgifter, syftar laborationerna till träning i att kombinera flera begrepp och applicera dessa tillsammans i ett större program med flera samverkande delar.

En laboration varar i 2 timmar och är schemalagd i salar med datorer som kör Linux. Följande anvisningar gäller för laborationerna:

1. **Obligatorium.** Laborationerna är obligatoriska och en viktig del av kursens examination. Godkända laborationer visar att du kan tillämpa den teori som ingår i kursen och att du har tillgodogjort dig en grundläggande förmåga att självständigt, och i grupp, utveckla större program med många delar. *Observera att samtliga laborationer måste vara godkända innan du får göra det muntliga provet och den valfria tentan!*
2. **Individuellt arbete och fusk.** Du ska lösa de individuella laborationerna *självständigt* genom eget, enskilt arbete. Du får hjälpa andra med att förstå men inte ge eller ta emot färdiga lösningar. Läs *noga* nedan om vad som är tillåtet och inte. Fusk kan medföra avstängning från universitetet och indraget studiemedel. Urkundsförfalskning kan medföra åtal i domstol.
  - (a) Det är tillåtet att under förberedelserna diskutera övergripande principer för laborationernas lösningar med andra, men var och en ska självständigt skapa en egen lösning.
  - (b) Under redovisningen ska du för handledare på begäran ingående förklara din individuella lösning och de begrepp som ingår i lärandemålen.
  - (c) Speciella anvisningar för grupplaborationer finns i avsnitt [-1.1.2](#).
  - (d) Det är *inte* tillåtet att lägga ut lösningar på nätet; det är medhjälp till fusk.
  - (e) Det är *inte* tillåtet att använda artificiell intelligens för att generera lösningar. Det är viktigt att du i denna kurs lär dig att självständigt utveckla grundläggande lösningar så att du i framtiden ska kunna granska och värdera kvaliteten på AI-genererad kod.
  - (f) Läs noga på denna webbsida om var gränsen går mellan samarbete och fusk: <http://cs.lth.se/utbildning/samarbete-eller-fusk/>
  - (g) Fusk är inte bara riskabelt och oetiskt, det undergräver dessutom dina fortsatta studier. Begreppen som du lär dig i denna kurs är en grundförutsättning för att du ska ha glädje av efterföljande kurser och ett djupinriktat lärande i denna kurs är grundläggande för hela din utbildning.



3. **Förberedelser.** Till varje laboration finns förberedelser som du ska göra *före* laborationen. Detta är helt avgörande för att du ska hinna göra laborationen inom 2 timmar. Ta hjälp av en kamrat eller en handledare under resurstiderna om det dyker upp några frågor under ditt förberedelsearbete. Innan varje laboration skall du ha:

- (a) studerat relevanta delar av kompendiet;
- (b) gjort grunduppgifterna som ingår i veckans övning, och gärna även (några) extraövningar och/eller fördjupningsövningar;
- (c) läst igenom *hela* laborationen noggrant;
- (d) löst förberedelseuppgifterna. I labbförberedelserna ska du i förekommande fall skriva delar av den kod som ingår i laborationen. Det krävs inte att allt du skrivit är helt korrekt, men du ska ha gjort ett rimligt försök. Ta hjälp om du får problem med uppgifterna, men låt inte någon annan lösa uppgiften åt dig.

Om du inte hinner med alla obligatoriska labbuppgifter, får du göra de återstående uppgifterna på egen hand och redovisa dem vid påföljande labbtillfälle eller resurstid, och förbereda dig *ännu* bättre till nästa laboration...

4. **Sjukanmälan.** Om du är sjuk vid något laborationstillfälle måste du anmäla detta till *kursansvarig* via mejl *före* laborationen. Om du varit sjuk ska du försöka göra uppgiften på egen hand och sedan redovisa den vid nästa labbtillfälle eller resurstid. Om du behöver hjälp att komma ikapp efter sjukdom, kom till en eller flera resurstider och prata med en handledare. Om du uteblir utan att ha anmält sjukdom kan kursansvarig besluta att du får vänta till nästa läsår med redovisningen, och då får du inte något slutbetyg i kursen under innevarande läsår.



5. **Skriftliga svar.** Vid några laborationsuppgifter finns en penna i marginalen. Denna symbol indikerar att du ska skriva ner och spara ett resultat som du behöver senare, och/eller som du ska visa upp för labbhandledaren vid en efterföljande kontrollpunkt eller vid den avslutande redovisningen.



6. **Kontrollpunkter.** Vid några laborationsuppgifter finns en ögonsymbol med en bock i marginalen. Detta innebär att du nått en kontrollpunkt där du ska diskutera dina resultat med en handledare. Räck upp handen och visa vad du gjort innan du fortsätter. Om det är lång väntan innan handledaren kan komma så är det ok att ändå gå vidare, men glöm inte att senare diskutera med handledaren så att ni gemensamt säkerställer att du förstått alla delresultat. Dialogen med din handledare är en viktig chans till återkoppling på din kod – ta vara på den!

## -1.6 Kontrollskrivning

Efter första halvan av kursen ska du göra en *obligatorisk kontrollskrivning*, som genomförs individuellt på papper och penna, och liknar till formen den ordinarie tentan. Kontrollskrivningen är *diagnostisk* och syftar till att hjälpa dig att avgöra ditt kunskapsläge när halva kursen återstår. Ett annat syfte är att ge träning i att lösa skrivningsuppgifter med papper och penna utan datorhjälpmedel.

Kontrollskrivningen rättas med *kamratbedömning* under själva skrivningstillfället. Du och en kurskamrat får efter att skrivningstiden är ute två andra skrivningar att poängbedöma i enlighet med en bedömningsmall. Syftet med detta är att du ska få träning i att bedöma



kod som andra skrivit och att resonera kring kodkvalitet. När rättningen är klar får du se poängsättningen av din skrivning och kan i händelse av avgörande felaktigheter överklaga bedömningen till kursansvarig.

Den diagnostiska kontrollskrivningen påverkar inte om du blir godkänd eller ej på kursen, men det samlade poängresultatet för din samarbetsgrupp ger möjlighet till *samarbetsbonus* som kan påverka ditt betyg på kursen (se avsnitt -1.1.3).

## -1.7 Projektuppgift

Efter avslutad labbserie följer en *obligatorisk projektuppgift* där du på egen hand ska skapa ett stort program med många olika samverkande delar. Det är först när mängden kod blir riktigt stor som du verkligen har nytta av de olika abstraktionsmekanismer du lärt dig under kursens gång och din felsökningsförmåga sätts på prov. Följande anvisningar gäller för projektuppgiften:

1. **Val av projektuppgift.** Du väljer själv projektuppgift. I kapitel 12 finns flera förslag att välja bland. Läs igenom alla uppgiftsalternativ innan du väljer vilken du vill göra. Du kan också i samråd med en handledare definiera en egen projektuppgift, men innan du börjar på en egendefinerad projektuppgift ska en skriftlig beskrivning av uppgiften godkännas av handledare i god tid innan redovisningstillfället. Välj uppgift efter vad du tror du klarar av och undvik både en för simpel uppgift och att ta dig vatten över huvudet.
2. Anvisningarna 1 och 2 för laborationer (se avsnitt -1.5) gäller också för projektuppgiften: den är **obligatorisk** och arbetet ska ske **individuellt**. Du får diskutera din projektuppgift på ett övergripande plan med andra och du kan be om hjälp av handledare på resurstid med enskilda detaljer om du kör fast, men lösningen ska vara *din* och du ska ha skrivit hela programmet själv.
3. **Omfattning.** Skillnaden mellan projektuppgiften och labbarna är att den ska vara *väsentligt* mer omfattande än de största laborationerna och att du färdigställer den kompletta lösningen *innan* redovisningstillfället. Du behöver därför börja i god tid, förslagsvis två veckor innan redovisningstillfället, för att säkert hinna klart. Det är viktigt att du tänker igenom omfattningen noga, i förhållande till ditt val av projektuppgift, gärna utifrån din självinsikt om vad du behöver träna på. Diskutera gärna med en handledare hur du använder projektuppgiften på bästa sätt för ditt lärande.
4. **Dokumentation.** Inför redovisningen ska du skapa automatiskt genererad dokumentation utifrån relevanta dokumentationskommentarer för minst hälften av dina publika metoder, enligt instruktioner i Appendix E.
5. **Kodlagring och versionshantering.** Projektuppgiften kan vara ett lämpligt tillfälle att träna på versionshantering med git. Det är, precis som för laborationer, *inte* tillåtet att lagra dina lösningar öppet på nätet. Om du vill träna på att använda en kodlagringsplats, t.ex. GitHub eller GitLab, var då noga med att kontrollera att repositoriet är stängt (eng. *closed repository*), så att du inte riskerar medhjälp till fusk. Användning av git och kodlagringsplats är valfritt.
6. **Redovisning.** Vid redovisningen använder du tiden med handledaren till att gå igenom din lösning och redogöra för hur din kod fungerar och diskutera för- och nackdelar med ditt angreppssätt. Du ska också beskriva framväxten av ditt program och hur du stegvis

har avlusat och förbättrat implementationen. På redovisningen ska du även gå igenom dokumentationen av din kod.

## -1.8 Muntligt prov

På schemalagd tid senast sista läsveckan i december ska du avlägga ett obligatoriskt muntligt prov för handledare. Du måste vara godkänt på alla laborationer för att få göra det muntliga provet. Syftet med provet är att kontrollera att du har godkänd förståelse för de begrepp som ingår i kursen. Du rekommenderas att förbereda dig noga inför provet, t.ex. genom att gå igenom grundläggande begrepp för varje kursmodul och repetera grundövningar och laborationer.

Provet sker som ett stickprov ur kursens innehåll. Du kommer att få några slumpvis valda frågor där du ombeds förklara några av de begrepp som ingår i kursen. Du får även uppdrag att skriva kod som liknar kursens övningar och förklara hur koden fungerar. Du kan träna på typiska frågor här: <https://cs.lth.se/pgk/muntabot/>

Om det visar sig oklart huruvida du uppnått godkänd förståelse kan du behöva komplettera ditt muntliga prov. Kontakta kursansvarig för information om omprov.

## -1.9 Valfri tentamen

Kursen avslutas med en *valfri skriftlig tentamen* med snabbreferensen<sup>1</sup> som enda tillåtna hjälpmedel. Du måste vara godkänd på obligatoriska moment för att få tentera. Tentamensuppgifterna är uppdelade i två delar, del A och del B, med följande preliminära betygsgränser:

- Del A omfattar 20% av den maximala poängsumman.
- Om du på del A erhåller färre poäng än vad som krävs för att nå upp till en bestämd ”rättningsströskel”, kan din tentamen komma att underkännas utan att del B bedöms.
- Preliminära betygsgränser:
  - För betyg 4 krävs minst 67% av maxpoängen, inklusive eventuell samarbetsbonus.
  - För betyg 5 krävs minst 83% av maxpoängen, inklusive eventuell samarbetsbonus.

---

<sup>1</sup><http://cs.lth.se/pgk/quickref>

# Kapitel 0

## Hur bidra till kursmaterialet?

### 0.1 Bidrag är varmt välkomna!

Ett av huvudsyftena med att göra detta kursmaterial fritt och öppet är att möjliggöra bidrag från alla som är intresserade. Speciellt välkommet är bidrag från studenter som vill vara delaktiga i att utveckla undervisningen.

### 0.2 Instruktioner

#### 0.2.1 Vad behövs för att kunna bidra?

Om du hittar ett problem, t.ex. ett enkelt stavfel, eller har något mer omfattande som borde förbättras, men ännu inte känner till eller har tillgång till de verktyg som beskrivs nedan och som behövs för att göra bidrag, kontakta då någon som redan bidragit till materialet, så att någon annan kan implementera ditt förslag.

Innan du själv kan implementera ändringar direkt i materialet, behöver du känna till, och ha tillgång till, ett eller flera av följande verktyg (beroende på vad ändringen gäller):

- Latex: [en.wikibooks.org/wiki/LaTeX](https://en.wikibooks.org/wiki/LaTeX)
- Scala: [en.wikipedia.org/wiki/Scala\\_%28programming\\_language%29](https://en.wikipedia.org/wiki/Scala_%28programming_language%29)
- git: [https://en.wikipedia.org/wiki/Git\\_%28software%29](https://en.wikipedia.org/wiki/Git_%28software%29)
- GitHub: [en.wikipedia.org/wiki/GitHub](https://en.wikipedia.org/wiki/GitHub)
- sbt: [en.wikipedia.org/wiki/SBT\\_%28software%29](https://en.wikipedia.org/wiki/SBT_%28software%29)

Läs mer om hur du bidrar här:

[github.com/lunduniversity/introprog#how-to-contribute](https://github.com/lunduniversity/introprog#how-to-contribute)

#### 0.2.2 Svenska eller engelska?

Vi blandar engelska och svenska enligt följande principer:

- Publika diskussioner, t.ex. i *issues* och *pull requests* på GitHub, sker på engelska. I en framtid kan delar av materialet komma att översättas till engelska och då är det bra om även icke-svenskspråkiga kan förstå vad som har hänt. Alla ändringshändelser sparas och man kan söka och gå tillbaka i historiken.
- Kompendiet finns för närvarande bara på svenska eftersom kursen initialt endast ges för svenskspråkiga studenter, men texten ska hjälpa läsaren att tillgodogöra sig motsvarande engelsk terminologi. Skriv därför motsvarande engelska begrepp (eng. *concept*) i parentes med hjälp av latex-kommandot `\Eng{concept}`.

- På övningar och föreläsningar är svenska variabelnamn ok. Svenska kan användas för att hjälpa den som håller på att lära sig att skilja på ord som vi själv hittar på och ord som finns i programmeringsspråket. Detta signalerar också att när man lär sig och experimenterar kan man hitta på tokroliga namn och använda svenska hur mycket man vill. Man lär sig genom att prova!
- Kod i labbar ska vara på engelska. Detta signalerar att när man kodar för att det ska bli något bestående, då kodar man på engelska.

## 0.3 Exempel

Som exempel på hur det går till i ett typiskt öppen-källkodsprojekt, beskrivs nedan vad som hände i ett verkligt fall: en dokumentationsuppdatering av Scala-dokumentationen efter att ett fel upptäckts. Detta exempelfall är ett typiskt scenario som illustrerar hur det kan gå till, och vad man kan behöva tänka på. Exemplet ger också länkar till och inblick i ett riktigt stort projekt med öppen källkod.

### Scenario: att göra ett bidrag vid upptäckt av problem

”Jag fick till min stora glädje denna *Pull Request* (PR) accepterad till dokumentationssajten för Scala. Man kan se mitt bidrag här:

[github.com/scala/scala.github.com/pull/517/commits/2624c305a8a6f24ea3398fe0fcbd0c72492bdd12](https://github.com/scala/scala.github.com/pull/517/commits/2624c305a8a6f24ea3398fe0fcbd0c72492bdd12)

Att börja med att bidra till dokumentation är ofta en bra väg att komma in i ett öppen-källkodsprojekt, då det är en god chans att hjälpa till utan att det behöver kräva djup kompetens om koden i *repot*<sup>1</sup>. Jag beskriver nedan vad som hände steg för steg då jag fick en riktig PR accepterad, som ett typiskt exempel på hur det ofta fungerar.

1. Jag tyckte dokumentationen för metoden `lengthCompare` på indexerbara samlingar på [scala-lang.org/documentation](https://scala-lang.org/documentation) var förvirrande. När jag provade i REPL blev det uppenbart att något var fel: antingen så var dokumentationen fel eller så funkade inte metoden som den skulle. Ojoj, kanske har jag upptäckt ett nytt fel? En chans att bidra!
2. Först sökte jag noga bland alla ärenden som ligger under fliken 'issues' på GitHub för att se om någon redan hittat detta problem. Om så vore fallet hade jag kunnat kommentera ett sådant ärende och skriva något till stöd för att den behöver fixas, eller allra helst att erbjuda mig att försöka fixa den. Men jag hittade inget ärende om detta...
3. Jag skapade därför ett nytt ärende genom att klicka på knappen *New issue* i webbgränssnittet på GitHub och här syns resultatet:  
<https://github.com/scala/scala.github.com/issues/515#>  
Jag tänkte noga på hur jag skulle formulera mig:
  - Ärendetiteln är extra viktig: den ska sammanfatta på en enda rad vad det hela rör sig om så att läsaren av rubriken förstår vad problemet handlar om.
  - Jag jobbade sedan med att skriva en tydlig och detaljerad beskrivning av problemet och angav exakt vilken version det gällde. Det är bra att klistra in exempel från Scala REPL och andra testfallskörningar med indata och utdata om relevant. Det är viktigt att problemet går att hitta och återskapa av andra, därför behövs information om vilken version det gäller och ett minimalt testfall som renodlar problemet.

---

<sup>1</sup>Ordet repo är en förkortning av repositorium, här i betydelsen en lagringsplats för kod.

- Det är bra att ställa frågor och komma med förslag för att öppna en diskussion om ärendet. Jag frågade speciellt om detta var ett dokumentationsproblem eller en bugg i koden.
  - OBS! Man ska inte öppna ett ärende innan man först kollat noga att det verkligen är något som bör åtgärdas och att det inte är en dubblett eller överlapp med andra issues: varje gång man öppnar ett ärende kommer det att generera arbete för andra även om ärendet inte ens till slut resulterade i någon åtgärd...
  - Om det är ett mer öppet, allmänt förslag, en förbättring eller en helt ny feature kan man också skapa en issue (det måste alltså inte vara en renodlad bugg). Är man osäker på om ärendet är relevant, är det bra att diskutera det i gemenskapens mejlforum först.
4. Jag fick snabbt kommentarer på mitt ärende, vilket är kännetecknande för en väl fungerande gemenskap (eng. *community*) med alerta reposkötare (eng. *maintainers*). Och när jag fick uppmuntran att bidra, så erbjöd jag mig att implementera förbättringen.
  5. Tänk på att alltid skriva alla kommentarer och svar i en saklig, kortfattad och trevlig ton!
  6. Nästa steg är att "forka" repot på GitHub genom att helt enkelt klicka på *Fork* i webbgränssnittet. Jag fick då en egen kopia av repot under min egen användare på GitHub, där jag har rättigheter att ändra.
  7. Därefter klonade jag repot till min lokala maskin med terminalkommandot `git clone https://...` (eller så kan man använda skrivbordsappen GitHub Desktop).
  8. Sedan rättade jag problemet direkt i relevant fil i en editor på min dator, i detta fallet var filen i formatet Markdown (ett lättläst textformat som man kan generera HTML från):  
[raw.githubusercontent.com/scala/scala.github.com/master/overviews/collections/seqs.md](https://raw.githubusercontent.com/scala/scala.github.com/master/overviews/collections/seqs.md)
  9. När jag fixat problemet gjorde jag `git add` på filen och sedan `git commit -m "välgenomtänkt commit msg"`  
Jag tänkte efter noga innan jag skrev första raden i commit-meddelandet så att det skulle vara både kort och kärnfullt. Men ändå glömde jag att inkludera issue-numret : (, se min kommentar till commiten, som jag tillfogade i efterhand, när jag till slut upptäckte min fadäs:  
[scala.github.com/commit/2624c305a8a6f24ea3398fe0fcbd0c72492bdd12#comments](https://scala.github.com/commit/2624c305a8a6f24ea3398fe0fcbd0c72492bdd12#comments)
  10. Efter att jag gjort `git commit` så finns ändringen ännu så länge bara lokalt på min dator. Då gäller det att "pusha" till min fork på GitHub med `git push` (eller använda *Sync*-knappen i GitHub-desktop-appen).
  11. Därefter skapade jag en PR genom att helt enkelt trycka på knappen *New pull request* på GitHub-sidan för min fork. Jag tänkte efter noga innan jag författade rubriken som beskriver denna PR. Hade denna ändring varit mer omfattande hade jag också behövt göra en detaljerad beskrivning av hur ändringen var implementerad för att underlätta granskningen av mitt förslag. Ni kan se denna (numera avslutade) PR här:  
<https://github.com/scala/scala.github.com/pull/517>
  12. När jag skapat en PR fick de som sköter repot ett automatiskt meddelande om denna nya PR och den efterföljande granskningsfasen inträdde. Den brukar sluta med att en

eller flera andra personer kommenterar PR i webbgränssnittet med 'LGTM'. LGTM = "*Looks Good To Me*" och betyder ungefär "jag har kollat på detta nu och det verkar (vad jag kan bedöma) vara utmärkt och alltså redo för *merge*". Om det inte ser bra ut så förväntas granskaren föreslå vad som behöver förbättras i en saklig och trevlig ton.

13. När PR är granskad så kan en person, som har rättigheter att ändra, "merga" in PR på huvudgrenen, som ofta kallas *master*, i det centrala repot, som ofta kallas *upstream*.
14. Avslutningsvis kan ärendet stängas av de ansvariga för repot. Denna issue är nu markerad "Closed" och syns inte längre i listan med aktiva issues.

Puh! Sen var det klart :) "

*Epilog:* Om du i framtiden får chansen att göra fler bidrag är det viktigt att först uppdatera din fork mot upstream innan du gör några nya ändringar i din lokala kopia; annars är risken att din PR innehåller föråldrad information och därmed blir en merge onödigt krånglig. Detta kan man göra genom en knapp i GitHub Desktop eller genom att följa denna beskrivning: [help.github.com/articles/syncing-a-fork/](https://help.github.com/articles/syncing-a-fork/) Det är i allmänhet den som ändrar som ansvarar för att ändringar alltid sker i samklang med den mest aktuella versionen av upstream.

# **Del II**

# **Moduler**

# Kapitel 1

## Introduktion

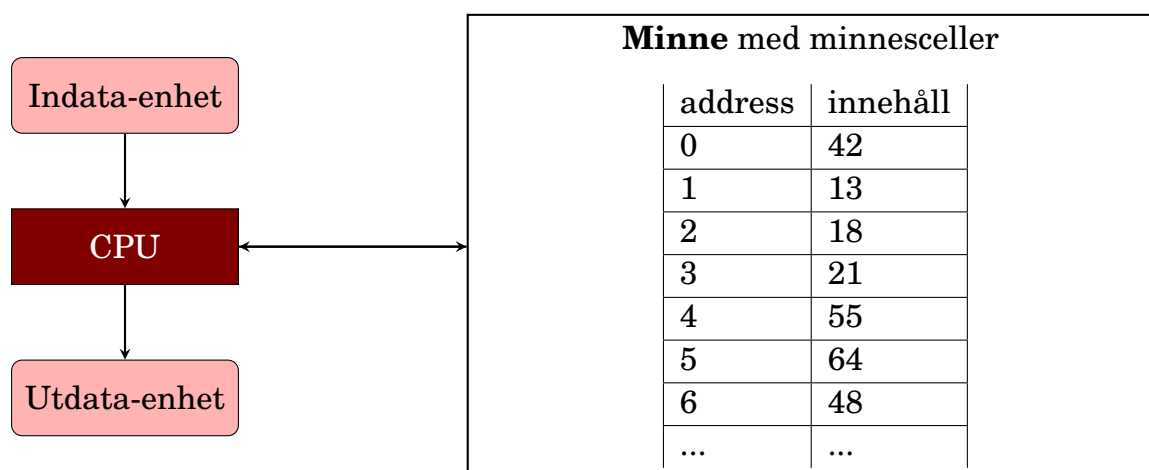
Begrepp som ingår i denna veckas studier:

- |  |  |
|--|--|
| <input type="checkbox"/> sekvens         | <input type="checkbox"/> def                           |
| <input type="checkbox"/> alternativ      | <input type="checkbox"/> definiera och anropa funktion |
| <input type="checkbox"/> repetition      | <input type="checkbox"/> funktionshuvud                |
| <input type="checkbox"/> abstraktion     | <input type="checkbox"/> funktionskropp                |
| <input type="checkbox"/> editera         | <input type="checkbox"/> procedur                      |
| <input type="checkbox"/> kompilera       | <input type="checkbox"/> inbyggda grundtyper           |
| <input type="checkbox"/> exekvera        | <input type="checkbox"/> println                       |
| <input type="checkbox"/> datorns delar   | <input type="checkbox"/> typen Unit                    |
| <input type="checkbox"/> virtuell maskin | <input type="checkbox"/> enhetsvärdet ()               |
| <input type="checkbox"/> litteral        | <input type="checkbox"/> stränginterpolatorn s         |
| <input type="checkbox"/> värde           | <input type="checkbox"/> aritmetik                     |
| <input type="checkbox"/> uttryck         | <input type="checkbox"/> slumpstal                     |
| <input type="checkbox"/> identifierare   | <input type="checkbox"/> logiska uttryck               |
| <input type="checkbox"/> variabel        | <input type="checkbox"/> de Morgans lagar              |
| <input type="checkbox"/> typ             | <input type="checkbox"/> if                            |
| <input type="checkbox"/> tilldelning     | <input type="checkbox"/> true                          |
| <input type="checkbox"/> namn            | <input type="checkbox"/> false                         |
| <input type="checkbox"/> val             | <input type="checkbox"/> while                         |
| <input type="checkbox"/> var             | <input type="checkbox"/> for                           |



## 1.1 Teori

### 1.1.1 Hur fungerar en dator?



Minnet innehåller endast **heltal** som representerar **data och instruktioner**.

### 1.1.2 Vad är programmering?

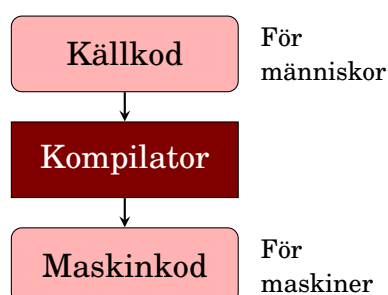
- Programmering innebär att ge instruktioner till en maskin.
- Ett **programmeringsspråk** används av människor för att skriva **källkod** som kan översättas av en **kompilator** till **maskinspråk** som i sin tur **exekveras** av en dator.

- Ada Lovelace publicerade det första programmet redan på 1800-talet ämnat för en kugghjulsdator.



- [sv.wikipedia.org/wiki/Programmering](https://sv.wikipedia.org/wiki/Programmering)
- [en.wikipedia.org/wiki/Computer\\_programming](https://en.wikipedia.org/wiki/Computer_programming)
- Ha picknick i [Ada Lovelace-parken](#) på Brunnshög!

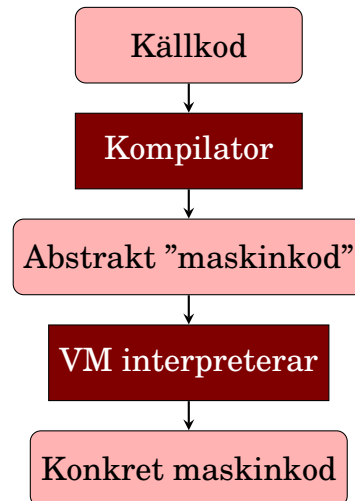
### 1.1.3 Vad är en kompilator?



Grace Hopper uppfann kompilatorn 1952.  
[en.wikipedia.org/wiki/Grace\\_Hopper](https://en.wikipedia.org/wiki/Grace_Hopper)

### 1.1.4 Virtuellt maskin (VM) == abstrakt hårdvara

- En VM är en "dator" implementerad i mjukvara som kan tolka en abstrakt "maskinkod" som **översätts under körning** till den **verkliga** maskinens konkreta maskinkod.
- Med en VM blir källkoden **plattformsoberoende** och fungerar på många olika maskiner.
- Exempel JVM:  
**Java Virtual Machine**



---

### 1.1.5 Vad består ett program av?

- Text som följer entydiga språkregler (grammatik):
  - **Syntax**: textens konkreta utseende
  - **Semantik**: textens betydelse (vad maskinen gör/beräknar)
- **Nyckelord**: ord med speciell betydelse, t.ex. **if**, **while**
- **Deklarationer**: definitioner av nya ord: **def** gurka = 42
- **Satser** är instruktioner som **gör** något: `print("hej")`
- **Uttryck** är instruktioner som beräknar ett **resultat**: `1 + 1`
- **Data** är information som behandlas: t.ex. heltalet 42
- Instruktioner ordnas i kodstrukturer: **SARA**
  - **Sekvens**: ordningen spelar roll för vad som händer
  - **Alternativ**: olika resultat beroende på uttrycks värde
  - **Repetition**: instruktioner upprepas många gånger
  - **Abstraktion**: nya byggblock skapas för att återanvändas

---

### 1.1.6 Exempel på programmeringsspråk

Det finns massor med olika språk och det kommer ständigt nya.

- Java
- C
- C++
- C#
- Python
- JavaScript
- Scala

Några topplistor:

- [Redmonk](#)
- [PYPL](#)
- [TIOBE](#)

---

### 1.1.7 Olika programmeringsparadigm

- Det finns många olika **programmeringsparadigm** (sätt att programmera på), till exempel:
  - **imperativ programmering**: programmet är uppbyggt av satser som påverkar systemets tillstånd
  - **objektorienterad programmering**: en sorts imperativ programmering där programmet består av objekt som kapslar in data och erbjuder operationer som bearbetar dessa data
  - **funktionsprogrammering**: programmet är uppbyggt av samverkande funktioner som undviker förändringar av data
  - **deklarativ programmering, logikprogrammering**: programmet är uppbyggt av logiska uttryck som beskriver olika fakta eller villkor och exekveringen utgörs av en bevisprocedur som söker efter värden som uppfyller fakta och villkor

Denna kurs behandlar de tre första.

---

### 1.1.8 Hello world

Kör rad för rad i Scala REPL (Read-Evaluate-Print-Loop):

```
> scala-cli repl
Welcome to Scala 3.3.0 (17.0.8, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> println("Hello World!")
Hello World!
```

@main framför valfri funktion anger var ett fristående program ska starta:

```
@main def hi = println("Hello world!")
```

Spara texten ovan i filen `hello.scala` och kompilera ditt program:

```
> scala-cli compile hello.scala
```

Kör ditt program:

```
> scala-cli run hello.scala
Hello World!
```

Det räcker med `scala-cli run` då koden kompileras automatiskt vid behov.

---

### 1.1.9 Utvecklingscykeln

editera; kompilera; hitta fel och förbättringar; editera; kompilera; hitta fel och förbättringar;  
editera; kompilera; hitta fel och förbättringar; editera; kompilera; hitta fel och förbättringar;  
editera; kompilera; hitta fel och förbättringar; editera; kompilera; hitta fel och förbättringar;  
...

```
upprepa(1000){  
  editera  
  kompilera  
  testa  
}
```

---

### 1.1.10 Utvecklingsverktyg

- Din verktygskunskap är mycket viktig för din produktivitet.
- Lär dig kortkommandon för vanliga handgrepp.
- Alla verktyg som behövs finns förinstallerade på **LTH:s linuxdatorer**. Om din egen burk krånglar: kör på skolans burkar så du ej fördröjs!
- Verktyg vi använder i kursen:
  - Scala **REPL**: från övn 1
  - Barnvänlig Scala-programmering med Kojo: Lab 1
  - **Texteditor** för kod, t.ex **VS code**: från övn 2
  - Kompilera och kör fristående program med **scala-cli**: från övn 2
- Andra verktyg som är bra att lära sig:  
(ingår i EDAA60 Datorer och datoranvändning)
  - Git för versionshantering
  - GitHub för kodlagring – men **inte** av lösningar till labbar!
  - Linux/Ubuntu och nyttiga terminalkommando

---

### 1.1.11 Installera verktyg på din egen dator

När du ska skriva kod i en editor, kompilera i terminalen och köra ditt program som en **fristående applikation**, så behövs:

- En editor: **VS Code** med tillägget **Scala (Metals)**
  - Körmiljön **OpenJDK**
  - Kommandoverktyg för terminalen: **scala-cli**
  - Se instruktioner här: <http://cs.lth.se/pgk/verktyg>
  - Läs mer i Appendix C.
  - Tips om du kör Windows: installera nya Windows Terminal
  - Installationshjälp:
    1. **InstallLunch**: E:2116, 12-13, w01:tis-fre, w02:mån-fre
    2. Pluggkvällar som SRD ordnar.
    3. #frågor-och-svar på vår Discord-server
    4. Fråga handledare på resurstid (i mån av tid).
-

### 1.1.12 Scala Command Line Interface (CLI)

- Utvecklingen av ett nytt kommandogränssnitt (eng. *Command Line Interface (CLI)*) för Scala startades 2022 i ett öppen-källkodsprojekt som leds av Virtuslab.
- Under 2023 ersätter **scala-cli** det gamla **scala**-kommandot.<sup>1</sup>
- Läs mer i Appendix C och F, samt här: <https://scala-cli.virtuslab.org/>
- Du kan se vad Scala CLI kan göra via hjälp-optionen:

```
> scala-cli help
```

---

### 1.1.13 Tips och trix med scala i terminalen

- Skriv `:help` i REPL så får du se vilka **kommando** som finns.
- Du kan **avsluta** REPL med `:q` eller trycka Ctrl+D.
- Ett **vertikalstreck** **visas** om du trycker ENTER mitt i en ofullständigt rad. Detta indikerar att du kan fortsätta skriva på ny rad innan tolkning sker.
- Om du vill att REPL ska vänta att tolka raden du skrivit och istället ge dig **ännu en rad**, så tryck först ner ESC-tangenten och sedan ENTER.
- Om du vill förhindra att REPL ger ny rad efter ENTER vid ofullständig rad, så skriv ett **semikolon** och tryck ENTER.
- Starta repl med punkt efter blanktecken om du vill ha tillgång till koden i alla scala-filer i **aktuell katalog** i din REPL-session:  
`scala-cli repl .`
- Kör med punkt efter blanktecken så kompileras och exekveras alla scala-filer i **aktuell katalog och** eventuella **underkataloger**:  
`scala-cli run .`

---

### 1.1.14 Litteraler

- En litteral representerar ett fixt **värde** i koden och används för att skapa **data** som programmet ska bearbeta.
- Exempel:
  - `42` heltalslitteral
  - `42.0` decimaltalslitteral
  - `'!'` teckenlitteral, omgärdas med 'enkelfnuttar'
  - `"hej"` stränglitteral, omgärdas med "dubbelfnuttar"
  - `true` litteral för sanningsvärdet "sant"
- Litteraler har en **typ** som avgör vad man kan göra med dem.

---

<sup>1</sup>I skrivande stund så har skiftet ännu inte skett. När det sker kan du ersätta alla förekomster av `scala-cli` med det kortare `scala`.

### 1.1.15 Exempel på inbyggda datatyper i Scala

- Alla värden, uttryck och variabler har en **datatyp**, t.ex.:
  - Int för heltal
  - Long för *extra* stora heltal (tar mer minne)
  - Double för decimaltal, så kallade flyttal med flytande decimalpunkt
  - String för strängar
- Kompilatorn håller reda på att uttryck kombineras på ett **typsäkert** sätt. Annars blir det **kompileringsfel**.
- Scala och Java är s.k. **statiskt typade** språk, vilket innebär att kontroll av typinformation sker vid kompilering (eng. *compile time*)<sup>2</sup>.
- Scala-kompilatorn gör **typhärledning**: man **slipper skriva typerna** om kompilatorn kan lista ut dem med hjälp av typerna hos deluttrycken.

### 1.1.16 Grundtyper i Scala

Dessa **grundtyper** (eng. *basic types*) finns inbyggda i Scala:

Svenskt namn	Engelskt namn	Grundtyper
heltalstyp	integral type	Byte, Short, Int, Long, Char
flyttalstyp	floating point number types	Float, Double
numeriska typer	numeric types	heltalstyper och flyttalstyper
strängtyp (teckensekvens)	string type	String
sanningsvärdestyp (boolesk typ)	truth value type	Boolean

### 1.1.17 Grundtypernas omfång

Grundtyp	Antal bitar	Omfång: minsta & största värde
Byte	8	$-2^7 \dots 2^7 - 1$
Short	16	$-2^{15} \dots 2^{15} - 1$
Char	16	$0 \dots 2^{16} - 1$
Int	32	$-2^{31} \dots 2^{31} - 1$
Long	64	$-2^{63} \dots 2^{63} - 1$
Float	32	$\pm 3.4028235 \cdot 10^{38}$
Double	64	$\pm 1.7976931348623157 \cdot 10^{308}$

<sup>2</sup>Andra språk, t.ex. Python och Javascript är **dynamiskt typade** och där skjuts typkontrollen upp till körningsdags (eng. *run time*)

Vilka är för- och nackdelarna med statisk vs. dynamisk typning?

Grundtypen String lagras som en *sekvens* av 16-bitars tecken av typen Char och kan vara av godtycklig längd (tills minnet tar slut).

---

### 1.1.18 Uttryck

- Ett **uttryck** består av en eller flera delar som efter **evaluering** ger ett **resultat**.
  - Delar i ett uttryck kan t.ex. vara:  
litteraler (42), operatorer (+), funktioner (sin), ...
  - Exempel:
    - Ett enkelt uttryck:  
42.0
    - Sammansatta uttryck:  
40 + 2  
(20 + 1) \* 2  
sin(0.5 \* Pi)  
"hej" + " på " + "dej"
  - När programmet tolkas sker **evaluering** av uttrycket, vilket ger ett resultat i form av ett **värde** som har en **typ**.
- 

### 1.1.19 Variabler

- En **variabel** kan tilldelas värdet av ett enkelt eller sammansatt uttryck.
- En variabel har ett **variabelnamn**, vars utformning följer språkets regler för s.k. **identifierare**.
- En ny variabel införs i en **variabeldeklaration** och då den kan ges ett värde, **initialiseras**. Namnet användas som **referens** till värdet.
- Exempel på variabeldeklarationer i Scala, notera **nyckelordet val**:

```
val a = 0.5 * Pi
val length = 42 * sin(a)
val exclamationMarks = "!!!"
val greetingSwedish = "Hej på dej" + exclamationMarks
```

- Vid exekveringen av programmet lagras variablernas värden i minnet och deras respektive värde hämtas ur minnet när de **refereras**.
  - Variabler som deklarerats med **val** kan endast tilldelas ett värde **en enda gång**, vid den initialisering som sker vid deklarationen.
- 

### 1.1.20 Regler för identifierare

- **Enkel** identifierare: t.ex. gurka2Tomat
  - Börja med bokstav
  - ...följt av bokstäver eller siffror

- Kan även innehålla understreck
- **Operator**-identifierare, t.ex. +:
  - Börjar med ett **operatortecken**, t.ex. + - \* / : ? ~ #
  - Kan följas av fler operatortecken
- En identifierare får **inte** vara ett **reserverat ord**, se [snabbreferensen](#) för alla reserverade ord i Scala.
- **Bokstavlig** identifierare: `kan innehålla allt`
  - Börjar och slutar med **backticks** ``
  - Kan innehålla vad som helst (utom backticks)
  - Kan användas för att undvika krockar med reserverade ord: `val`

### 1.1.21 Att bygga strängar: konkatenering och interpolering

- Man kan **konkatenera** strängar med operatör +  
"hej" + " på " + "dej"
- Efter en sträng kan man konkatenera vilka uttryck som helst; uttryck inom parentes evalueras först och värdet görs sen om till en sträng före konkateneringen:

```
val x = 42
val msg = "Dubbla värdet av " + x + " är " + (x * 2) + "."
```

- Man kan i Scala få hjälp av kompilatorn att övervaka bygget av strängar med **stränginterpolatorn s**:

```
val msg = s"Dubbla värdet av $x är ${x * 2}."
```

### 1.1.22 Heltalsaritmetik

- De fyra räknesätten skrivs som i matematiken (vanlig [precedens](#)):

```
1 scala> 3 + 5 * 2 - 1
2 res0: Int = 12
```

- **Parenteser** styr **evalueringsordningen**:

```
1 scala> (3 + 5) * (2 - 1)
2 val res1: Int = 8
```

- **Heltalsdivision** sker med **decimaler avkortade**:

```
1 scala> 41 / 2
2 val res2: Int = 20
```

- **Moduloräkning** med restoperatör %

```
1 scala> 41 % 2
2 val res3: Int = 1
```



### 1.1.23 Flyttalsaritmetik

- Decimaltal representeras med s.k. **flyttal** av typen Double:

```
1 scala> math.Pi
2 val res4: Double = 3.141592653589793
```

- Stora tal så som  $\pi * 10^{12}$  skrivs:

```
1 scala> math.Pi * 1E12
2 val res5: Double = 3.141592653589793E12
```

- Det finns **inte** oändligt antal decimaler vilket ger problem med **avrundningsfel**:

```
1 scala> 0.1 + 0.2
2 val res6: Double = 0.30000000000000004
3
4 scala> 1E10 + 0.0000000000000001
5 val res7: Double = 1.0E10
6
7 scala> BigDecimal("0.1") + BigDecimal("0.2") // BigDecimal funkar
8 val res8: BigDecimal = 0.3
```

Läs mer här: <https://0.30000000000000004.com>

### 1.1.24 Definiera namn på uttryck

- Med nyckelordet **def** kan man låta ett **namn** betyda samma sak som ett **uttryck**.
- Exempel:

```
def gurklängd = 42 + x
```

- Uttrycket till höger evalueras **varje** gång **anrop** sker, d.v.s. varje gång namnet används på annat ställe i koden.

```
gurklängd
```

### 1.1.25 Funktion, argument, parameter

- En **funktion** räknar ut **resultat** baserat på indata som kallas **argument**.
- Argument ges namn genom deklaration av **parametrar**.
- Exempel på deklaration av en funktion med en parameter:

```
def dubblera(x: Int) = 2 * x
```

- Parametrarnas typ **måste** beskrivas efter **kolon**.
- Kompilatorn kan härleda **returtypen**, men den kan också med fördel, för tydlighetens skull, anges **explicit**:

```
def dubblera(x: Int): Int = 2 * x
```

- Observera att namnet x blir ett "nytt fräscht" **lokalt namn** som **bara finns och syns "inuti" funktionen** och har inget med ev. andra x utanför funktionen att göra.

- Beräkningen sker först vid **anrop** av funktionen:

```
1 scala> dubblera(42)
2 res1: Int = 84
```

### 1.1.26 Färdiga matte-funktioner i paketet `scala.math`

- I paketet **scala.math** finns många användbara funktioner: t.ex. `math.random()` ger slumpstal mellan 0.0 och 0.9999999999999999

```
scala> val x = math.random()
x: Double = 0.27749191749889635

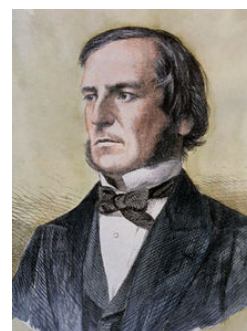
scala> val length = 42.0 * math.sin(math.Pi / 3.0)
length: Double = 36.373066958946424
```

- Studera dokumentationen här:  
<http://www.scala-lang.org/api/current/scala/math/>
- Paketet `scala.math` delegerar ofta till Java-klassen **java.lang.Math** som är dokumenterad här:  
<https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

### 1.1.27 Logiska uttryck

- Datorn kan "räkna" med sanning och falskhet:  
s.k. **boolsk algebra** efter **George Boole**
- Enkla logiska uttryck: (finns bara två stycken)

**true**  
**false**



- Sammansatta logiska uttryck med logiska operatorer:  
&& och, || eller, ! icke, == likhet, != olikhet, relationer: > < >= <=
- Exempel:

```
true && true
false || true
!false
42 == 43
42 != 43
(42 >= 43) || (1 + 1 == 2)
```

### 1.1.28 De Morgans lagar

**De Morgans lagar** beskriver vad som händer om man **negerar** ett logiskt uttryck. Kan användas för att göra **förenklingar**.

- I alla deluttryck sammanbundna med **&&** eller **||**, ändra alla **&&** till **||** och omvänt.
- Negera alla ingående deluttryck. En relation negeras genom att man byter **==** mot **!=**, **<** mot **>=**, etc.

Exempel på förenkling där de Morgans lagar används upprepat:

```
! (a < b || (a == 1 && b == 1))      ⇔
! (a < b) && ! (a == 1 && b == 1)    ⇔
! (a < b) && (! (a == 1) || ! (b == 1)) ⇔
a >= b && (a != 1 || b != 1)
```

### 1.1.29 Alternativ med if-uttryck

- Ett if-uttryck börjar med nyckelordet **if**, följt av ett logiskt uttryck (villkor) inom parentes och två grenar.

```
def slumpgrönsak = if math.random() < 0.8 then "gurka" else "tomat"
```

- Uttrycket efter **then** blir resultatet om villkoret är **true**
- Uttrycket efter **else** blir resultatet om villkoret är **false**

```
scala> slumpgrönsak
res13: String = gurka

scala> slumpgrönsak
res14: String = gurka

scala> slumpgrönsak
res15: String = tomat
```

### 1.1.30 Uttryck eller sats?

Skillnad mellan uttryck och sats:

- Ett uttryck ger ett **resultat**. Exempel: 1+1
- En sats har en **effekt**.  
Exempel: utskrift, spara på fil, tilldela variabel nytt värde.

Skriv ett **uttryck** när du är intresserad av **värdet** som beräknas.

Skriv en **sats** när du vill att något ska **göras**.

Både satser och uttryck kan i sin tur innehålla satser och uttryck i godtyckligt komplexa **nästlade strukturer** (mer om det senare).

### 1.1.31 Variabeldeklaration och tilldelningssats

- En **variabeldeklaration** medför att **plats i datorns minne** reserveras så att värden av den typ som variabeln kan referera till får plats där.
- Vid deklaration ska variabeln **initialiseras** med ett startvärde.
- En **val**-deklaration ger en variabel som efter initialisering inte kan ändras.

Dessa deklarationer...

```
var x = 42
val y = x + 1
```

... ger detta innehåll någonstans i minnet:

x	42
y	43

- Med en **tilldelningssats** ges en tidigare **var**-deklarerad variabel ett nytt värde:

```
x = 13
```

- Det gamla värdet försvinner för alltid och det nya värdet lagras istället:

x	13
y	43

Observera att y här inte påverkas av att x ändrade värde.

### 1.1.32 Tilldelningssatser är *inte* matematisk likhet

- Likhetstecknet används alltså för att **tilldela** variabler nya värden och det är **inte** samma sak som matematisk likhet. Vad händer här?

```
x = x + 1
```

- Denna syntax är ett arv från de gamla språken C, Fortran mfl.
- I **andra språk** används t.ex.

```
x := x + 1      eller      x <- x + 1
```

- Denna syntax visar kanske bättre att tilldelning är en **stegvis process**:
  1. Först beräknas **uttrycket till höger** om tilldelningstecknet.
  2. Sedan **ersätts värdet** som variabelnamnet refererar till av det beräknade uttrycket. Det gamla värdet **försvinner för alltid**.

### 1.1.33 Förkortade tilldelningssatser

- Det är vanligt att man vill tilldela en variabel ett nytt värde som beror av det gamla, så som i

```
x = x + 1
```

- Därför finns **förkortade tilldelningssatser** som gör så att man sparar några tecken och det blir tydligare (?) vad som sker (när man vant sig vid detta skrivsätt):

```
x += 1
```

- Uttrycket ovan expanderar av kompilatorn till  $x = x + 1$

---

### 1.1.34 Exempel på förkortade tilldelningssatser

```
scala> var x = 42
val x: Int = 42

scala> x *= 2

scala> x
val res0: Int = 84

scala> x /= 3

scala> x
val res1: Int = 28
```

---

### 1.1.35 Variabler som ändrar värden kan vara knepiga

- Kod som innehåller variabler som **förändras** över tid är ofta svårare att läsa och begripa.
- Många buggar beror på att variabler av misstag förändras på felaktiga och oanade sätt.
- Föränderliga värden blir speciellt svåra i kod som körs jämlöpande (parallellt).
- I kod som körs i skarpt läge med många användare (s.k. produktionskod) är därför **val** att föredra, medan **var** endast används om det **verkligen** behövs.
- Alltså: räkna hellre ut nya värden än förändra befintliga.

---

### 1.1.36 Kontrollstrukturer: alternativ och repetition

Används för att kontrollera (förändra) sekvensen och skapa **alternativa** vägar genom koden. Vägen bestäms vid körtid.

- if-sats:

```
if math.random() < 0.8 then println("gurka") else println("tomat")
```

Olika sorters **loopar** för att repetera satser. Antalet repetitioner ges vid körtid.

- **while**-sats: bra när man **inte vet hur många gånger** det kan bli.

```
while math.random() < 0.8 do println("gurka")
```

- **for**-sats: bra när man **vill ange antalet repetitioner**:

```
for i <- 1 to 10 do println(s"gurka nr $i")
```

---

### 1.1.37 Scala-2-syntax för kontrollstrukturer fungerar i Scala 3

I Scala 2 användes en gammal syntax för kontrollstrukturer som liknar mer C/C++/Java. Den är tillåten i Scala 3, men nya mer lättlästa syntaxen är att föredra.

- Scala-2-syntax för alternativ: parenteser men inget **then**

```
if (math.random() < 0.8) println("gurka") else println("tomat")
```

Scala-2-syntax för repetition:

- **while**-sats: parenteser men inget **do**

```
while (math.random() < 0.8) println("gurka")
```

- **for**-sats: parenteser men inget **do**

```
for (i <- 1 to 10) println(s"gurka nr $i")
```

- Kojo Desktop funkar ännu bara med Scala 2 och gamla syntaxen, men Kojo kan även köras med Scala 3 (se hur i kompendiet).

---

### 1.1.38 Repetera många satser

Om du vill göra flera saker i sekvens inne i en repetition så kan du skriva flera satser inom **klammer-parenteser**:

```
while math.random() < 0.8 do {  
  println("gurka")  
  println("tomat")  
}  
println("Repetitionen är klar!")
```

Du kan efter vissa nyckelord (t.ex. **do**, **then**, **else**) välja bort klammer-parenteser (eng. *optional braces*).

```
while math.random() < 0.8 do  
  println("gurka")  
  println("tomat")  
  
println("Repetitionen är klar!")
```

Då är det **indenteringen** som avgör vilka satser som ingår. Detta fungerar i Scala 3 (men inte i Scala 2).

---

### 1.1.39 Procedurer

- En **procedur** är en funktion som **gör** något intressant, men som **inte** lämnar något intressant returvärde.
- Exempel på procedur i standardbiblioteket: `println("hej")`
- Du **deklarerar egna procedurer** genom att ange **Unit** som returvärdestyp. Då returneras värdet **()** som betyder "inget".

```
scala> def hej(x: String): Unit = println(s"Hej på dej $x!")
hej: (x: String)Unit

scala> hej("Herr Gurka")
Hej på dej Herr Gurka!

scala> val x = hej("Fru Tomat")
Hej på dej Fru Tomat!
x: Unit = ()
```

- Det som **görs** kallas (sido)**effekt**. Ovan är utskriften själva effekten.
- Även funktioner kan ha sidoeffekter. De kallas då **oäkta** funktioner.

---

### 1.1.40 Problemlösning: nedbrytning i abstraktioner som sen kombineras

- En av de allra viktigaste principerna inom programmering är **funktionell nedbrytning** där **underprogram** i form av funktioner och procedurer skapas för att bli byggstenar som kombineras till mer avancerade funktioner och procedurer.
- Genom de namn som definieras skapas **återanvändbara abstraktioner** som kapslar in det funktionen gör till ett "byggblock".
- Bra "byggblock" gör det lättare att lösa svåra programmeringsproblem.
- Abstraktioner som beräknar eller gör **en enda, väldefinierad sak** är enklare att använda, jämfört med de som gör många, helt olika saker.
- Abstraktioner med **välgenomtänkta namn** är enklare att använda, jämfört med kryptiska eller missvisande namn.

---

### 1.1.41 Övning expressions och labb **kojo**

- På övningen kör du Scala REPL för att träna på SARA.  
**Läs i Appendix** och på kursens hemsida under "Verktyg" om hur du installerar och får igång Scala REPL.
- På laborationen använder du barnvänliga **Kojo** för träna på SARA, med fokus på abstraktion.
- Det finns tre olika sätt att använda Kojo:
  1. Grafikbiblioteket **kojoLib** i ett fristående Scala program med hjälp av en professionell kodeditor och kompilering och exekvering i terminalen. **Fungerar fint med nya Scala 3.**
  2. Skrivbordsappen **Kojo Desktop** med inbyggd barnvänlig editor (endast Scala 2).

3. Webbappen <http://kojo.lu.se/> som körs direkt i din webbläsare (endast Scala 2, begränsade funktioner).

Alternativ 1 rekommenderas, men om du försenas av tekniskt strul, så kom igång med 2 el. 3 så länge tills du fått hjälp.

---

### 1.1.42 Köa med Sigrid

För att köa till handledare på plats i sal i pgk: [cs.lth.se/sigrid](http://cs.lth.se/sigrid)

- Direkt när undervisningspasset **börjar**: starta en session med ditt förnamn, kurskod EDAA45 och rummets namn. Gör detta även om du inte behöver hjälp från start! Då kan **ambulanser** se antal studenter i varje rum.
  - Inget lösenord behövs.
  - Två olika köer i varje rum: **hjälpkö** och **redovisningskö**
    - Ställ dig i hjälpkö om du vill få vägledning och ställa frågor
    - Ställ dig i redovisningskö om du är klar att redovisa en labb
  - Du måste klicka på **Uppdatera** – annars händer inget!
  - OBS! **Köar inte+Uppdatera** så fort handledare anländer!
  - Om du går på extra pass i mån av plats så kan du se vilket rum som har kortast kö här: [cs.lth.se/sigrid/monitor](http://cs.lth.se/sigrid/monitor)
- 

### 1.1.43 Sigrid in action

Så här ser det ut när student står i hjälpkö efter att först ha klickat på **Hjäälp!!!** och sedan på **Uppdatera**-knappen:

## STUDENT oddput-1 i Alfa

Välj tillstånd och klicka på gröna *Uppdatera*-knappen.

Köar inte

Jobbar eller får hjälp.

Hjäälp!!!

Står i hjälpkön.

Färdiig!

Står i redovisningskön.

Loggar ut

Redovisar, lämnar rummet.

Glöm inte *Köar inte* + *Uppdatera* medan du får hjälp.  
Glöm inte *Loggar ut* + *Uppdatera* medan du redovisar.

Uppdatera

GLÖM INTE **Köar inte** + **Uppdatera** när handledare *anländer*!

---



## 1.2 Övning expressions

### Mål

- ☐ Förstå vad som händer när satser exekveras och uttryck evalueras.
- ☐ Förstå sekvens, alternativ och repetition.
- ☐ Känna till litteralerna för enkla värden, deras typer och omfång.
- ☐ Kunna deklarerar och använda variabler och tilldelning, samt kunna rita bilder av minnessituationen då variablers värden förändras.
- ☐ Förstå skillnaden mellan olika numeriska typer, kunna omvandla mellan dessa och vara medveten om noggrannhetsproblem som kan uppstå.
- ☐ Förstå booleska uttryck och värdena **true** och **false**, samt kunna förenkla booleska uttryck.
- ☐ Förstå skillnaden mellan heltalsdivision och flyttalsdivision, samt användning av rest vid heltalsdivision.
- ☐ Förstå precedensregler och användning av parenteser i uttryck.
- ☐ Kunna använda **if**-satser och **if**-uttryck.
- ☐ Kunna använda **for**-satser och **while**-satser.
- ☐ Kunna använda `math.random()` för att generera slumpstal i olika intervaller.
- ☐ Kunna beskriva skillnader och likheter mellan en procedur och en funktion.

### Förberedelser

- ☐ Studera begreppen i kapitel 1
- ☐ Du behöver en dator med Scala och Kojo, se appendix C och A.

### 1.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** *Para ihop begrepp med beskrivning.*

Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

litteral	1	A	att införa nya begrepp som förenklar kodningen
sträng	2	B	antingen sann eller falsk
sats	3	C	att översätta kod till exekverbar form
uttryck	4	D	anger ett specifikt datavärde
funktion	5	E	decimaltal med begränsad noggrannhet
procedur	6	F	en kodrad som gör något; kan särskiljas med semikolon
exekveringsfel	7	G	en sekvens av tecken
kompileringsfel	8	H	kombinerar värden och funktioner till ett nytt värde
abstrahera	9	I	beskriver vad data kan användas till
kompilera	10	J	vid anrop sker (sido)effekt; returvärdet är tomt
typ	11	K	vid anrop beräknas ett returvärde
for-sats	12	L	för att ändra en variabels värde
while-sats	13	M	kan inträffa innan exekveringen startat
tilldelning	14	N	kan inträffa medan programmet kör
flyttal	15	O	bra då antalet repetitioner är bestämt i förväg
boolesk	16	P	bra då antalet repetitioner ej är bestämt i förväg

### Uppgift 2. Utskrift i Scala REPL.

Starta Scala REPL (eng. *Read-Evaluate-Print-Loop*).

```
> scala
Welcome to Scala 3.1.2 (17.0.2, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.
scala -version.
scala>
```

- Skriv efter prompten `scala>` en sats som skriver ut en valfri (bruklig/knasig) hälsningsfras, genom anrop av proceduren `println` med något strängargument. Tryck på *Enter* så att satsen kompileras och exekveras.
- Skriv samma sats igen (eller tryck pil-upp) men "glöm bort" att skriva högerparentesen efter argumentet innan du trycker på *Enter*. Vad händer?

*Tips inför fortsättningen:* Det finns många användbara kortkommandon och andra trix för att jobba snabbt i REPL. Be gärna någon som kan dessa trix att visa dig hur man kan jobba snabbare. Läs appendix C.4.2 och prova sedan att kopiera och klistra in text. Använd piltangenterna för att bläddra i historiken, Ctrl+A för att komma till början av raden, Ctrl+K för att radera resten av raden, etc.

### Uppgift 3. Konkatenering av strängar.

- Skriv ett uttryck som konkatenerar två strängar, t.ex. "gurk" och "burk", med hjälp av operatoren `+` och studera resultatet. Vad har uttrycket för värde och typ? Vilken siffra står efter ordet `res` i variabeln som lagrar resultatet?

b) Använd resultatet från konkateneringen, t.ex. `res0` (byt ev. ut `0`:an mot siffran efter `res` i utskriften från förra evalueringen), och skriv ett uttryck med hjälp av operatoren `*` som upprepar resultatet från förra deluppgiften 42 gånger.

#### Uppgift 4. När upptäcks felet?

- Vad har uttrycket `"hej" * 3` för typ och värde? Testa i REPL.
- Byt ut `3`:an ovan mot ett så pass stort heltal så att minnet blir fullt, men inte så stort att talet inte får plats i det givna omfånget för grundtypen `Int`. Hur börjar felmeddelandet? Är detta ett körtidsfel eller ett kompileringsfel?
- Välj ett värde på argumentet efter operatoren `*` så att ett typfel genereras. Hur börjar felmeddelandet? Är detta ett körtidsfel eller ett kompileringsfel?

*Tips inför fortsättningen:* Gör gärna fel när du kodar så lär du dig mer! Träna på att tolka olika felmeddelanden och fråga någon om hjälp om du inte förstår. Kompilatorns utskrifter kan vara till stor hjälp, men är ibland kryptiska. Om du kör fast och inte kommer vidare själv så be om hjälp, *men be om tips snarare än färdiga lösningar* så att du behåller initiativet själv och tar kontroll över nästa steg i ditt lärande.

#### Uppgift 5. Litteraler och typer.

- Ta hjälp av REPL-kommandot `:type` (kan förkortas `:t`) vid behov för att para ihop nedan litteraler med rätt typ.

1	1	A	String
1L	2	B	Boolean
1.0	3	C	Boolean
1D	4	D	Unit
1F	5	E	Int
'1'	6	F	Double
"1"	7	G	Long
true	8	H	Float
false	9	I	Char
()	10	J	Double

- Vad händer om du adderar 1 till det största möjliga värdet av typen `Int`?  
*Tips:* se snabbreferensen <sup>3</sup> under rubriken "The Scala type system" avsnitt "Methods on numbers".
- Vad är skillnaden mellan typerna `Long` och `Int`?
- Vad är skillnaden mellan typerna `Double` och `Float`?

#### Uppgift 6. Matematiska funktioner. Använda dokumentation.

<sup>3</sup><http://cs.lth.se/pgk/quickref/>

a) Antag att du har ett schackbräde med 64 rutor. Tänk dig att du börjar med att lägga ett enda riskorn på första rutan och sedan lägger dubbelt så många riskorn i en ny hög för varje efterföljande ruta: 1, 2, 4, 8, ... etc. När du har gjort detta för alla rutor, hur många riskorn har du totalt lagt på schackbrädet?<sup>4</sup>

*Tips:* Du ska beräkna  $2^{64} - 1$ . Om du skriver `math.` i REPL och trycker TAB får du se inbyggda matematiska funktioner i Scalas standardbibliotek:

```
scala> math.    // Tryck TAB direkt efter punkten och betrakta listan
```

Använd funktionen `math.pow` och lämpliga argument. Om du anger `math.pow` eller `math.pow()` utan argument får du se funktionshuvudet med parameterlistan.

Om du surfar till <http://www.scala-lang.org/api/current/> och skriver `math` i sökrutan och sedan, efter att du klickat på `scala.math`, skriver `pow` i rutan längre ner, så filtreras sidan och du hittar dokumentationen av **def** `pow` som du kan klicka på och läsa mer om.

b) Definiera funktionen `omkrets` nedan i REPL. Går det bra att utelämna returtyp-annoteringen? Varför? Finns det anledning att ha den kvar?

```
def omkrets(radie: Double): Double = 2 * math.Pi * radie
```

c) Jordens (genomsnittliga) diameter (vid ekvatorn) är ca 12750 km. Skriv ett uttryck som anropar funktionen `omkrets` ovan för att beräkna hur många kilometer per dag man ungefär måste färdas om man vill åka jorden runt på 80 dagar.

### Uppgift 7. Variabler och tilldelning. Förändringsbar och oföränderlig variabel.

a) Rita en *ny* bild av datorns minne efter *varje* exekverad rad 1–6 nedan. Varje bild ska visa alla variabler som finns i minnet och deras variabelnamn, typ och värde.

```
1 scala> var a = 13
2 scala> val b = a + 1
3 scala> var c = (a + b) * 2.0
4 scala> b = 0
5 scala> a = 0
6 scala> c = c + 1
```

Efter första raden ser minnessituationen ut så här:

a: Int 13

b) Varför blir det fel på rad 4? Är det ett kompileringsfel eller exekveringsfel? Hur lyder felmeddelandet?

### Uppgift 8. Slumptal med `math.random()`.

a) Vad ger funktionen `math.random()` för resultatvärde? Vilken typ? Vad är största och minsta möjliga värde?

*Tips:* Sök här: <http://www.scala-lang.org/api/current/> och prova i REPL.

b) Deklarera den parameterlösa funktionen **def** `roll: Int = ???` som ska representera ett tärningskast och ge ett slumpmässigt heltal mellan 1 och 6. Testa funktionen genom att anropa den många gånger.

*Tips:* Använd `math.random()` och multiplicera och addera med lämpliga heltal. Omge beräkningen med parenteser och avsluta med `.toInt` för att avkorta decimaler och omvandla typen från `Double` till `Int`.

<sup>4</sup>[https://en.wikipedia.org/wiki/Wheat\\_and\\_chessboard\\_problem](https://en.wikipedia.org/wiki/Wheat_and_chessboard_problem)

**Uppgift 9.** *Repetition med **for**, **foreach** och **while**.*

a) Så här kan en **for**-sats ser ut:

```
for i <- 1 to 10 do print(s"$i, ")
```

Använd en **for**-sats för att skriva ut resultatet av 100 tärningskast med funktionen `roll` från uppgift 8.

b) Så här kan en **foreach**-sats ser ut:

```
(1 to 10).foreach(i => print(s"$i, "))
```

Använd en **foreach**-sats för att skriva ut resultatet av 100 tärningskast med funktionen `roll` från uppgift 8.

c) Så här kan en **while**-sats se ut:

```
var i = 1
while i <= 10 do { print(s"$i, "); i = i + 1 }
```

Använd en **while**-sats för att skriva ut resultatet av 100 tärningskast med funktionen `roll` från uppgift 8. Vad händer om du glömmer `i = i + 1`?

**Uppgift 10.** *Alternativ med **if**-sats och **if**-uttryck.*

a) Så här kan en **if**-sats se ut (notera dubbla likhetstecken):

```
if roll == 3 then println("TRE") else println("INTE TRE")
```

Testa ovan i REPL. Skriv sedan en **for**-sats som kastar 100 tärningar och skriver ut strängen "GRATTIS! " om det blir en sexa, annars en ledsen smiley: ": ( "

b) Så här kan ett **if**-uttryck se ut:

```
if roll < 6 then 0 else 1
```

Testa ovan i REPL. Skriv sedan en **while**-sats som kastar 100 tärningar och räknar antalet sexor. Skriv ut antalet efter **while**-satsen.

**Uppgift 11.** *Sekvens, sats och block.*

a) Vad gör dessa satser?

```
scala> def p = { print("san"); print("!"); println("hej")}
scala> p;p;p;p
```

b) Använd pil-upp för att få tillbaka raden du skrev med definitionen av proceduren `p`. Byt plats på strängarna i utskriftsanropen i proceduren `p` så att utskriften blir:

```
hejsan!
hejsan!
hejsan!
hejsan!
```

c) Hur tolkar kompilatorn klammerparenteser och semikolon? Vad är ett block?

**Uppgift 12. Heltalsdivision.** Vilket värde och vilken typ hör till vilket uttryck? Är du osäker på svaret, testa i REPL.

4 / 42	1	A	4: Int
42.0 / 2	2	B	10: Int
42 / 4	3	C	21.0: Double
42 % 4	4	D	<b>true</b> : Boolean
4 % 42	5	E	<b>false</b> : Boolean
40 % 4 == 0	6	F	0: Int
42 % 4 == 0	7	G	2: Int

**Uppgift 13. Booleska värden.** Vilket värde har dessa uttryck?

- a) **true** && **true**
- b) **false** && **true**
- c) **true** || **true**
- d) **false** || **true**
- e) **false** || **false**
- f) **true** == **true**
- g) **true** != **false**
- h) **true** > **false**
- i) **true** && (1 / 0 > 1)
- j) **false** && (1 / 0 > 1)

**Uppgift 14. Booleska variabler.** Vad skrivs ut på rad 2 och 4 nedan?

```
1 scala> var monster = false
2 scala> if monster then println("akta dig!!!")
3 scala> monster = true
4 scala> if monster then println("akta dig!!!")
```

**Uppgift 15. Turtle graphics med Kojo.** På veckans laboration ska du använda Kojo för att verifiera att du kan använda sekvens, alternativ, repetition och abstraktion. Med Kojo ska du skapa Scala-program som ritar färgglada figurer med hjälp av ett lättanvänt Scala-bibliotek för *turtle graphics*<sup>5</sup>.

Om du använder Kojo som ett grafikbibliotek (rekommenderas) och kör med `scala-cli` (se Appendix A) så kan du använda Scala 3. Men kör du Kojo Desktop eller Webb-Kojo så är det Scala 2 som gäller och även om det mesta i veckans labb fungerar lika i Scala 2 och Scala 3 så kräver Scala 2 den gamla syntaxen för kontrollstrukturer med nödvändiga parenteser runt villkorsuttryck, utan varken **do** eller **then**, och varken valfria klammerparenteser eller indenteringssyntax.

Skriv in och kör nedan program med valfri metod enligt Appendix A. Notera kopplingen mellan satsernas ordning och vad som händer i ritfönstret.

```
fram; höger
```

<sup>5</sup>[https://en.wikipedia.org/wiki/Turtle\\_graphics](https://en.wikipedia.org/wiki/Turtle_graphics)

```
fram; vänster  
färg(grön)  
fram
```

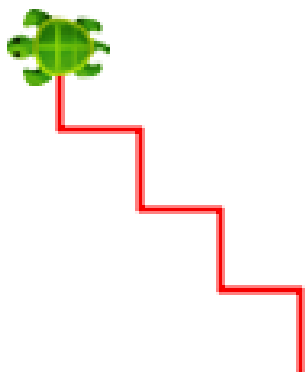
Om du kör Kojo Desktop är det bra att börjar programmet med sudda (varför det?<sup>6</sup>).

a) Skriv kod som ritar en kvadrat enligt bilden nedan.



Prova gärna olika sätt att skriva din kod *utan* att resultatet ändras: skriv satser i sekvens på flera rader eller satser i sekvens på samma rad med semikolon emellan; använd blanktecken och blanka rader i koden. Hur vill du gruppera dina satser så att de är lätta för en människa att läsa?

b) Rita en trappa enligt bilden nedan.



c) Rita valfri bild på valfri bakgrund med hjälp av några av procedurerna i tabellen nedan. Du kan till exempel rita en rosa triangel med lila konturer mot svart bakgrund. Försök att underlätta läsbarheten av din kod med hjälp av lämpliga radbrytningar och gruppering av satser.

---

<sup>6</sup>När du trycker på playknappen i Kojo Desktop så nollställs varken canvas i ritfönstret eller paddans tillstånd. Genom att börja dina Kojo Desktop-program med sudda så startar du exekveringen i exakt samma utgångsläge: en tom canvas där paddan pekar uppåt, pennan är nere och pennans färg är röd.



<code>fram(100)</code>	Paddan går framåt 100 steg (25 om argument saknas).
<code>färg(rosa)</code>	Sätter pennans färg till rosa.
<code>fill(lila)</code>	Sätter ifyllnadsfärgen till lila.
<code>fill(genomskinlig)</code>	Gör så att paddan <i>inte</i> fyller i något när den ritar.
<code>bredd(20)</code>	Gör så att pennan får bredden 20.
<code>bakgrund(svart)</code>	Bakgrundsfärgen blir svart.
<code>pennaNer</code>	Sätter ner paddans penna så att den ritar när den går.
<code>pennaUpp</code>	Sänker paddans penna så att den <i>inte</i> ritar när den går.
<code>höger(45)</code>	Paddan vrider sig 45 grader åt höger.
<code>vänster(45)</code>	Paddan vrider sig 45 grader åt vänster.
<code>hoppa</code>	Paddan hoppar 25 steg utan att rita.
<code>hoppa(100)</code>	Paddan hoppar 100 steg utan att rita.
<code>hoppaTill(100, 200)</code>	Paddan hoppar till läget (100, 200) utan att rita.
<code>gåTill(100, 200)</code>	Paddan vrider sig och går till läget (100, 200).
<code>öster</code>	Paddan vrider sig så att nosen pekar åt höger.
<code>väster</code>	Paddan vrider sig så att nosen pekar åt vänster.
<code>norr</code>	Paddan vrider sig så att nosen pekar uppåt.
<code>söder</code>	Paddan vrider sig så att nosen pekar neråt.
<code>sättVinkel(90)</code>	Paddan vrider nosen till vinkeln 90 grader.

*Tips inför fortsättningen:* Ha både REPL och en editor igång samtidigt. Då kan du undersöka hur olika kodfragment fungerar i REPL, medan du *stegvis* skapar allt större program i editorn. Detta sätt att jobba har du stor nytta av under resten av kursen. Oavsett vilka andra verktyg du kör är det användbart att ha REPL igång i ett eget fönster som hjälp i den kreativa processen, medan du jagar buggar och medan du lär dig nya koncept. Så fort du undrar hur något fungerar i Scala: **fram med REPL och testa!**

## 1.2.2 Extrauppgifter; träna mer

**Uppgift 16.** *Typ och värde.* Vilket värde och vilken typ hör till vilket uttryck? Är du osäker på svaret, testa i REPL.

1.0 + 18	1	A	1.042E42: Double
(41 + 1).toDouble	2	B	65: Int
1.042e42 + 1	3	C	113: Int
12E6.toLong	4	D	48: Int
32.toChar.toString	5	E	" ": String
'A'.toInt	6	F	0: Int
0.toInt	7	G	'*': Char
'0'.toInt	8	H	19.0: Double
'9'.toInt	9	I	12000000: Long
'A' + '0'	10	J	'q': Char
('A' + '0').toChar	11	K	42.0: Double
"*!%#".charAt(0)	12	L	57: Int

**Uppgift 17.** *Satser och uttryck.*

- Vad är det för skillnad på en sats och ett uttryck?
- Ge exempel på satser som inte är uttryck?
- Förklara vad som händer för varje evaluerad rad:

```

1 scala> def värdeSaknas = ()
2 scala> värdeSaknas
3 scala> värdeSaknas.toString
4 scala> println(värdeSaknas)
5 scala> println(println("hej"))

```

- Vilken typ har litteralen ()?
- Vilken returtyp har println?

**Uppgift 18.** *Procedur med parameter.* En procedur är en funktion som orsakar en effekt, till exempel en utskrift eller en variabeltilldelning, men som inte returnerar något intressant resultatvärde.<sup>7</sup>

- Deklarera en förändringsbar variabel highscore som initieras till 0.
- Deklarera en procedur updateHighscore som tar en parameter points och tilldelar highscore ett nytt värde om points är större än highscore och skriver ut strängen "REKORD!". Om inte points är större än highscore ska strängen "GE INTE UPP!" skrivas ut. Testa proceduren i REPL.
- Gör en ny variant av updateHighscore, som *inte* är en procedur utan i stället är en funktion som ger en sträng för senare utskrift. Testa funktionen i REPL.

<sup>7</sup>I Scala är procedurer funktioner som returnerar det *tomma värdet*, vilket skrivs () och är av typen Unit. I Java och flera andra språk finns inget tomt värde och man har en specialsyntax för procedurer som använder nyckelordet void.

**Uppgift 19.** *Flyttalsaritmetik.*

- Vilket är det minsta positiva värdet av typen Double?
- Vad är värdet av detta uttryck? Varför blir det så?

```
1 scala> Double.MaxValue + Double.MinPositiveValue == Double.MaxValue
```

**Uppgift 20.** *if-sats.* För varje rad nedan, beskriv vad som skrivs ut.

```
1 scala> if !true then println("sant") else println("falskt")
2 scala> if !false then println("sant") else println("falskt")
3 scala> def singlaSlant = if math.random() < 0.5 then "krona" else "klave"
4 scala> for i <- 1 to 5 do print(s"$i:$singlaSlant ")
```

**Uppgift 21.** Deklarera följande variabler med nedan initialvärden:

```
scala> var grönsak = "gurka"
scala> var frukt = "banan"
```

Ange för varje rad nedan vad uttrycket har för värde och typ:

```
scala> if grönsak == "tomat" then "gott" else "inte gott"
scala> if frukt == "banan" then "gott" else "inte gott"
scala> if true then grönsak else 42
scala> if false then grönsak else 42
```

**Uppgift 22.** *Modulo-operatorn % och Booleska värden.*

- Deklarera en funktion **def** `isEven(n: Int): Boolean = ???` som ger **true** om talet `n` är jämnt, annars **false**.
- Deklarera en funktion **def** `isOdd(n: Int): Boolean = ???` som ger **false** om talet `n` är jämnt, annars **true**.

**Uppgift 23.** *Skillnader mellan var, val, def.*

- Evaluera varje rad en i taget i tur och ordning i Scala REPL. För varje rad nedan: förklara för vad som händer och notera värde och ev fel.

```
1 scala> var x = 30
2 scala> x + 1
3 scala> x = x + 1
4 scala> x == x + 1
5 scala> val y = 20
6 scala> y = y + 1
7 scala> var z = { println("hej z!"); math.random() }
8 scala> def w = { println("hej w!"); math.random() }
9 scala> z
10 scala> z
11 scala> z = z + 1
12 scala> w
13 scala> w
14 scala> w = w + 1
```

- Vad är det för skillnad på **var**, **val** och **def**?

**Uppgift 24.** *Skillnaden mellan if och while.* Vad blir resultatet av rad 3 och 4?

```
1 scala> def lotto1 = if math.random() > 0.5 then print("vinst :) ")
2 scala> def lotto2 = while math.random() > 0.5 do print("vinst :) ")
3 scala> lotto1
4 scala> lotto2
```

### 1.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 25.** *Logik och De Morgans Lagar.* Förenkla följande uttryck. Antag att poäng och highscore är heltalsvariabler medan klar är av typen Boolean.

- `poäng > 100 && poäng > 1000`
- `poäng > 100 || poäng > 1000`
- `!(poäng > highscore)`
- `!(poäng > 0 && poäng < highscore)`
- `!(poäng < 0 || poäng > highscore)`
- `klar == true`
- `klar == false`

**Uppgift 26.** *Stränginterpolatorn s.* Med ett `s` framför en stränglitteral får man hjälp av kompilatorn att, på ett typsäkert sätt, infoga variabelvärden i en sträng. Variablernas namn ska föregås med ett dollartecken, t.ex. `s"Hej $namn"`. Om man vill evaluera ett uttryck placeras detta inom klammer direkt efter dollartecknet, t.ex. `s"Dubbla längden: ${namn.size * 2}"`

- Vad skrivs ut nedan?

```
1 scala> val f = "Kim"
2 scala> val e = "Finkodare"
3 scala> println(s"Namnet '$f $e' har ${f.size + e.size} bokstäver.")
```

- Skapa följande utskrifter med hjälp av stränginterpolatorn `s` och variablerna `f` och `e` i föregående deluppgift.

```
1 Kim har 3 bokstäver.
2 Finkodare har 9 bokstäver.
```

**Uppgift 27.** *Tilldelningsoperatorer.* Man kan förkorta en tilldelningssats som förändrar en variabel, t.ex. `x = x + 1`, genom att använda så kallade tilldelningsoperatorer och skriva `x += 1` som betyder samma sak. Rita en ny bild av datorns minne efter varje rad nedan. Bilderna ska visa variablers namn, typ och värde.

```
1 scala> var a = 40
2 scala> var b = a + 40
3 scala> a += 10
4 scala> b -= 10
5 scala> a *= 2
6 scala> b /= 2
```

**Uppgift 28.** *Stora tal.* Om vi vill beräkna  $2^{64} - 1$  som ett exakt heltal<sup>8</sup> blir det större än `Int.MaxValue`, så vi kan tyvärr inte använda snabba `Int`. Till vår räddning: `BigInt`

- Läs om `BigInt` och `BigDecimal` på <http://www.scala-lang.org/api/current/>. Notera vad de kan användas till.
- Du skapar ett `BigInt`-heltal med `BigInt(2)` och kan anropa funktionen `pow` på en `BigInt` med punktnotation. Beräkna  $2^{64} - 1$  som ett exakt heltal.
- Vilka nackdelar finns med `BigInt` och `BigDecimal`?

<sup>8</sup>[https://en.wikipedia.org/wiki/Wheat\\_and\\_chessboard\\_problem](https://en.wikipedia.org/wiki/Wheat_and_chessboard_problem)

**Uppgift 29.** *Precedensregler* Evalueringsordningen kan styras med parenteser. Vilket värde och vilken typ har följande uttryck?

- a)  $23 + 2 * 2 + (23 + 2) * 2$
- b)  $((-(2 - 42)) / (1 + 1 + 1))$
- c)  $((-(2 - 42)) / (-1)) / (1 + 1 + 1)$

**Uppgift 30.** *Dokumentation av paket i Java och Scala.*

a) Genom att trycka på tab tangenten kan man se vad som finns i olika paket. Vad heter konstanten  $\pi$  i `java.lang.Math` (notera stort M) respektive `scala.math`?

```
1 scala> java.lang.Math.    //tryck TAB efter punkten
2 scala> scala.math.       //tryck TAB efter punkten
```

b) Jämför dokumentationen för klassen `java.lang.Math` här:

<https://docs.oracle.com/javase/8/docs/api/>

med dokumentationen för paketet `scala.math` här:

<http://www.scala-lang.org/api>

Ge exempel på vad man kan göra på webbsidan med Scala-dokumentationen som man *inte* kan göra i motsvarande webbsida Java-dokumentation.

c) Vad gör metoden `hypot`? Vad är det som är bra med att använda `hypot` i stället för att själv implementera beräkningen med hjälp av kvadratroter, multiplikation och addition?

**Uppgift 31.** *Noggrannhet och undantag i aritmetiska uttryck.* Vad blir resultatet av uttrycken nedan? Notera undantag (eng. *exceptions*) och noggrannhetsproblem.

- a) `Int.MaxValue + 1`
- b) `1 / 0`
- c) `1E8 + 1E-8`
- d) `1E9 + 1E-9`
- e) `math.pow(math.hypot(3,6), 2)`
- ★ f) `1.0 / 0`
- ★ g) `(1.0 / 0).toInt`
- ★ h) `math.sqrt(-1)`
- ★ i) `math.sqrt(Double.NaN)`
- j) `throw new Exception("PANG!!!")`

★ **Uppgift 32.** *Modulo-räkning med negativa tal.* Läs om modulatoräkning här:

[en.wikipedia.org/wiki/Modulo\\_operation](http://en.wikipedia.org/wiki/Modulo_operation)

och undersök hur det blir med olika tecken (positivt resp. negativt) på modulatoräkning med `dividend%divisor` i Scala.

★ **Uppgift 33.** *Bokstavliga identifierare.* Läs om identifierare i Scala och speciellt *literal identifiers* här: <http://www.artima.com/pinsled/functional-objects.html#6.10>.

a) Förklara vad som händer nedan:

```
scala> val `bokstavlig val` = 42
scala> println(`bokstavlig val`)
```

b) Scala och Java har olika uppsättningar med reserverade ord. På vilket sätt kan ”back-ticks” vara användbart med anledning av detta?

★ **Uppgift 34.** *java.lang.Integer, hexadecimala litteraler, BigDecimal.*

a) Sök upp dokumentationen för `java.lang.Integer`.

Använd metoderna `toBinaryString` och `toHexString` för att fylla i tabellen nedan.

decimalt heltal	binärt värde	hexadecimalt värde
33		
42		
64		

b) Hur anger man det hexadecimala heltalsvärdet `10c` (motsvarar 268 decimalt) som en litteral i Scala?

c) Vad blir `0x10` upphöjt till  $c$  = ljusets hastighet i *m/s*? *Tips: Använd BigDecimal.*

★ **Uppgift 35.** *Strängformatering.* Läs om f-interpolatorn här:

<http://docs.scala-lang.org/overviews/core/string-interpolation.html>

Hur kan du använda f-interpolatorn för att göra följande utskrift i REPL? Ändra rad 2 vid ??? så att flyttalet `g` avrundas till tre decimaler innan utskrift sker.

```
1 scala> val g = 2 / 3.0
2 scala> val str = f"Jättegurkan är $g??? meter lång"
3 scala> println(str)
4 Jättegurkan är 0.667 meter lång
```

**Uppgift 36.** *Multiplikationsvarning.* Sök upp dokumentationen för `java.lang.Math.multiplyExact` och läs om vad den metoden gör.

a) Vad händer här?

```
scala> Math.multiplyExact(1, 2)
scala> Int.MaxValue * 2
scala> Math.multiplyExact(Int.MaxValue, 2)
```

b) Varför kan man vilja använda `java.lang.Math.multiplyExact` i stället för ”vanlig” multiplikation?

★ **Uppgift 37.** *Extra operatorer för exakt multiplikation.* Kim Kodmagiker tycker att `Math.multiplyExact` är för krångligt att skriva och utökar därför typen `Int` med en extra operator:

```
extension (i: Int) def *!(j: Int) = Math.multiplyExact(i, j)
```

a) Klistra in koden ovan i REPL och prova den extra operatorn.

b) Hjälp Kim Kodmagiker att lägga till fler operatorer på värden av typen `Int`, som gör att det även går att använda `Math.subtractExact` och `Math.addExact` smidigt.

c) Testa ett sammansatt uttryck som använder alla extrametoder på `Int`. Tycker du det blev mer lättläst eller mer kryptiskt med de nya operatorerna?

## 1.3 Laboration: kojo

### Mål

- ☐ Kunna tillämpa och kombinera principerna sekvens, alternativ, repetition, och abstraktion i skapandet av egna program om minst 20 rader kod.
- ☐ Kunna förklara vad ett program gör i termer av sekvens, alternativ, repetition, och abstraktion.
- ☐ Kunna formatera egna program så att de blir lätta att läsa och förstå.
- ☐ Kunna förklara vad en variabel är och kunna deklarerat oföränderliga och förändringsbara variabler, samt göra tilldelningar.
- ☐ Kunna genomföra upprepade varv i cykeln *editera-exekvera-felsöka/förbättra* för att stegvis bygga upp allt mer utvecklade program.

### Förberedelser

- ☐ Repetera veckans föreläsningsmaterial.
- ☐ Gör övning *expressions* i avsnitt 1.2
- ☐ Läs om Kojo i appendix A. Kojo Desktop är förinstallerat på LTH:s datorer; om du vill installera Kojo Desktop på din egen dator, följ instruktionerna i A.2. Du kan också köra Kojo i din webbläsare här: <http://kojo.lu.se/>
- ☐ Läs igenom hela laborationen nedan. Fundera på möjliga lösningar till de uppgifter som är markerade med en penna i marginalen.
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).

### 1.3.1 Obligatoriska uppgifter

Om det förekommer en penna i marginalen ska du anteckna något inför redovisningen.

**Uppgift 1.** *Sekvens och repetition.* Rita en kvadrat med hjälp av `upprepa(n){ ??? }` där du ersätter `n` med antalet repetitioner och `???` med de satser som ska repeteras.

**Uppgift 2.** *Variabel och repetition.*

a) Funktionen `System.currentTimeMillis` ingår i Javas standardbibliotek och ger ett heltal av typen `Long` med det nuvarande antalet millisekunder sedan midnatt den första januari 1970. Med Kojo-proceduren `sakta(0)` blir det ingen fördröjning när paddan ritar och utritningen sker så snabbt som möjligt. Prova nedan program och förklara vad som händer.


```
sakta(0)
val n = 800 * 4
val t1 = System.currentTimeMillis
upprepa(n){ upprepa(4){ fram; höger } }
val t2 = System.currentTimeMillis
println(s"$n kvadratvarv tog ${t2 - t1} millisekunder")
```

Om du kör Kojo Desktop är det bra att börja programmet med sudda. (Varför?)




b) Anteckna ungefär hur många kvadratvarv per sekund som paddan kan rita när den är som snabbast. Kör flera gånger eftersom den virtuella maskinen behöver "värmas upp" för att maskinkoden ska optimeras. Vissa körningar kan gå långsammare om skräpsamlaren behöver lägga tid på att frigöra minne.





-  c) Vad har variablerna i koden ovan för namn? Vad har variablerna för värden?
- d) Rita en kvadrat igen, men nu med hjälp av en **while**-sats och en loopvariabel.

```
sakta(100)
var i = 0
while (???) { fram; höger; i = ??? }
```

-  e) Vad är det för skillnad på variabler som deklareras med **val** respektive **var**?
- f) Rita en kvadrat igen, men nu med hjälp av en **for**-sats. Skriv ut värdet på den lokala variabeln *i* i varje loop-runda.

```
for (i <- 1 to ???) { ??? }
```

-  g) Går det att tilldela variabeln *i* ett nytt värde i loopen?
-  h) Går det att referera till namnet *i* utanför loopen?
- i) Rita en kvadrat igen, men nu med hjälp av **foreach**. Skriv ut loopvariabelns värde i varje runda.

```
(1 to ???).foreach{ i => ??? }
```

### Uppgift 3. Abstraktion.

- a) Använd en repetition för att abstrahera nedan sekvens, så att programmet blir kortare:

```
fram; höger; hoppa; fram; vänster; hoppa; fram; höger;
hoppa; fram; vänster; hoppa; fram; höger; hoppa; fram;
vänster; hoppa; fram; höger; hoppa; fram; vänster; hoppa;
fram; höger; hoppa; fram; vänster; hoppa
```

- b) Definiera en egen procedur som heter **kvadrat** med hjälp av nyckelordet **def** som vid anrop ritar en kvadrat med hjälp av en **for**-loop.

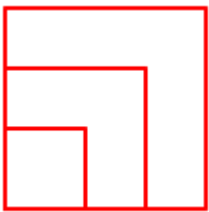
```
def kvadrat = for (???) {???
```

- c) Anropa din abstraktion efter att den deklarerats och efter att du exekverat:  
sakta(100)
- d) Anropa din abstraktion inuti en **for**-loop så att paddan ritar en stapel som är 10 kvadrater hög enligt bilden nedan.
- e) Studera hur anrop av proceduren **kvadrat** påverkar exekveringssekvensen av dina satser genom att göra lämpliga utskrifter så att du kan se när olika delar av koden exekveras. Vid vilka punkter i programmet sker ett "hopp" i sekvensen i stället för att efterföljande sats exekveras? Använd lämpligt argument till **sakta** för att du ska hinna studera exekveringen.
- f) Rita samma bild med 10 staplade kvadrater (se bild 1.1 på sidan 55), men nu *utan* att använda abstraktionen **kvadrat** – använd i stället en nästlad repetition (alltså en upprepning inuti en upprepning). Vilket av de två sätten (med och utan abstraktionen **kvadrat**) är lättast att läsa?
- g) Generalisera din abstraktion **kvadrat** genom att ge den en parameter sida: Double som anger kvadratens storlek. Rita flera kvadrater i likhet med bild 1.2 på sidan 55).



```
def kvadrat = for (???) {???}
for (???) {???}
```

**Figur 1.1:** En kvadratstapel.



**Figur 1.2:** Olika stora kvadrater.

#### Uppgift 4. Alternativ.

a) Kör programmet nedan. Förklara vad som händer.

```
sakta(5000)

def move(key: Int): Unit = {
  println("key: " + key)
  if (key == 87) fram(10)
  else if (key == 83) fram(-10)
}

move(87); move('W'); move('W')
move(83); move('S'); move('S'); move('S')
```

b) Kör programmet nedan. Notera `activateCanvas()` för att du ska slippa klicka i ritfönstret innan du kan styra paddan. Anropet `onKeyPress(move)` gör så att `move` kommer att anropas då en tangent trycks ned. Lägg till kod i `move` som gör att tangenten A ger en vridning moturs med 5 grader medan tangenten D ger en vridning medurs 5 grader. Med `onKeyPress` bestämmer man vilken procedur som ska köras vid tangenttryck.

```
sakta(0); activateCanvas()

def move(key: Int): Unit = {
  println("key: " + key)
  if (key == 'W') fram(10)
  else if (key == 'S') fram(-10)
```

```
}
onKeyPress(move)
```

### 1.3.2 Kontrollfrågor

✓ Repetera teorin för denna vecka och var beredd på att kunna svara på dessa frågor när det blir din tur att redovisa vad du gjort under laborationen:

1. Vad innebär sekventiell exekvering av satser?
2. Vad är skillnaden mellan en sats och ett uttryck?
3. Vad är skillnaden mellan en procedur och en funktion?
4. Spelar ordningen mellan argument någon roll vid anrop av en funktion med flera parametrar?
5. Vad är en variabel? Ge exempel på deklaration, initialisering och tilldelning av variabler, samt användning av variabler i uttryck.
6. Vad är ett logiskt uttryck? Ge exempel på användning av logiska uttryck.
7. Vad är abstraktion? Ge exempel på användning av abstraktion.
8. Vad är nyttan med abstraktion?
9. Hur deklareras och initialiseras en variabel vars värde är förändringsbart?
10. Hur deklareras och initialiseras en variabel vars värde är oföränderligt?
11. Är det ett körtidsfel eller kompileringsfel att tilldela en oföränderlig variabel ett nytt värde?
12. Ange vilken av **for** och **while** som är lämpligast i dessa fall:
  - A. Summera de hundra första heltalen.
  - B. Räkna antal tecken i en sträng innan första blanktecken.
  - C. Dra 100 slumpstal mellan 1 och 6 och summera de tal som är mindre än 3.
  - D. Summera de första heltalen från 1 och uppåt tills summan är minst 100.

### 1.3.3 Frivilliga extrauppgifter

Gör i mån intresse och träningsbehov nedan uppgifter i valfri ordning.

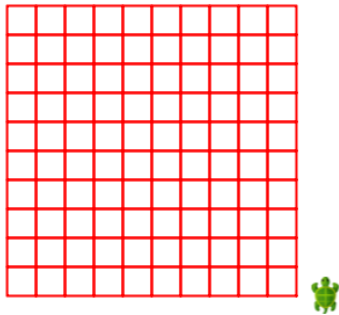
**Uppgift 5.** *Abstraktion och generalisering.*

- a) Skapa en abstraktion **def** stapel = ??? som använder din abstraktion kvadrat.
- b) Du ska nu *generalisera* din procedur så att den inte bara kan rita exakt 10 kvadrater i en stapel. Ge proceduren stapel en parameter n som styr hur många kvadrater som ritas.

```
def kvadrat = ???
def stapel(n: Int) = ???

sakta(100)
stapel(42)
```

c) Rita nedan bild med hjälp av abstraktionen `stapel`. Det är totalt 100 kvadrater och varje kvadrat har sidan 25. *Tips:* Med ett negativt argument till proceduren hoppa kan du få sköldpaddan att hoppa baklänges utan att rita, t.ex. `hoppa(-10*25)`



- d) Generalisera dina abstraktioner `kvadrat` och `stapel` så att man kan påverka storleken på kvadraterna som ritas ut.
- e) Skapa en abstraktion `rutnät` med lämpliga parametrar som gör att man kan rita rutnät med olika stora kvadrater och olika många kvadrater i både x- och y-led.
- f) Generalisera dina abstraktioner `kvadrat` och `stapel` så att man kan påverka fyllfärgen och pennfärgen för kvadraterna som ritas ut.

### Uppgift 6. Växling med booleska värden.

a) Bygg vidare på programmet i uppgift 4 och lägg till nedan kod i början av programmet. Lägg även till kod som gör så att om man trycker på tangenten G så sätts rutnätet omväxlande på och av. Observera att det är exakt *en* procedur som anropas vid `onKeyPress`.

```
var isGridOn = false

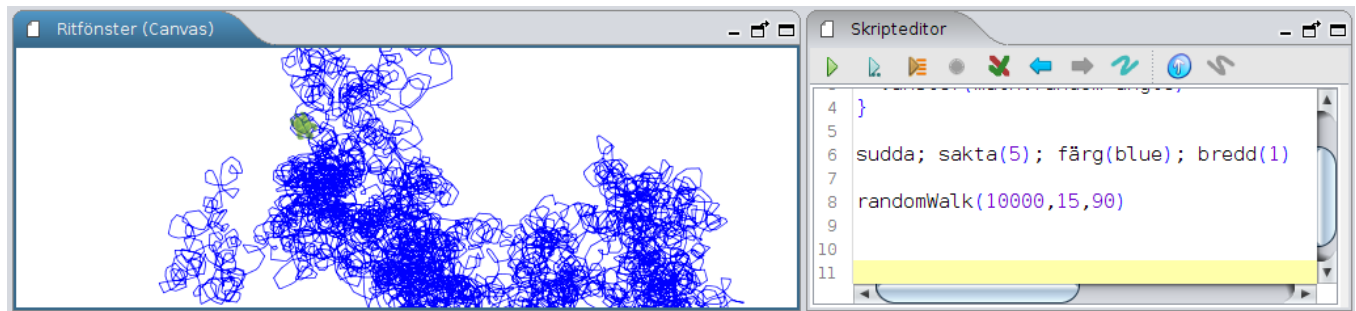
def toggleGrid =
  if (isGridOn) {
    gridOff
    isGridOn = false
  } else {
    gridOn
    isGridOn = true
  }
```

b) Gör så att när man trycker på tangenten X så sätter man omväxlande på och av koordinataxlarna. Använd en variabel `isAxesOn` och definiera en abstraktion `toggleAxes` som anropar `axesOn` och `axesOff` på liknande sätt som i föregående uppgift.

### Uppgift 7. Repetition. Skriv en procedur `randomWalk` med detta huvud:

```
def randomWalk(n: Int, maxStep: Int, maxAngle: Int): Unit
```

som gör så att paddan tar `n` steg av slumpmässig längd mellan 0 och `maxStep`, samt efter varje steg vrider sig åt vänster en slumpmässig vinkel mellan 0 och `maxAngle`. Anropa din procedur med olika argument och undersök hur dess värden påverkar bildens utseende. *Tips:* Uttrycket `math.random() * 100` ger ett tal från 0 till (nästan) 100. Du kan styra hur långsamt paddan ritar genom anrop av `sakta(???)` (prova dig fram till något lämpligt heltalsargument i stället för `???`).







### Uppgift 8. Variabler, namngivning och formatering.

a) Klistra in nedan konstigt formaterade program *exakt* som det står med blanktecken, indragningar och radbrytningar. Kör programmet och förklara vad som händer.

```
// Ett konstigt formaterat program med en del konstiga namn.
```

```
def gurka(x: Double,
y: Double, namn: String,
typ: String,
värde:String) = {
val tomat = 15
val h = 30
hoppaTill(x,y)
norr
skriv(namn+": "+typ)
hoppaTill(x+tomat*(namn.size+typ.size),y)
skriv(värde); söder; fram(h); vänster
fram(tomat * värde.size); vänster
fram(h); vänster
fram(tomat * värde.size); vänster }
sudda; färg(svart); val s = 130
val h = 40
var x = 42; gurka(10, s-h*0, "x","Int", x.toString)
var y = x; gurka(10, s-h*1, "y","Int", y.toString)
x = x + 1; gurka(10, s-h*2, "x","Int", x.toString)
gurka(10, s-h*3, "y","Int", y.toString); osynlig
```

-  b) Skriv ner namnet på alla variabler som förekommer i programmet.
-  c) Vilka av dessa variabler är lokala?
-  d) Vilka av dessa variabler kan förändras efter initialisering?
-  e) Föreslå tre förändringar av programmet ovan (till exempel namnbyten) som gör att det blir lättare att läsa och förstå.
- f) Gör sök-ersätt av gurka till ett bättre namn. *Tips:* undersök contextmenyn i editorn i Kojo genom att högerklicka. Använd kortkommandot för Sök/Ersätt.
- g) Gör automatisk formatering av koden med hjälp av lämpligt kortkommando. Notera skillnaderna. Vilka autoformateringar gör programmet lättare att läsa? Vilka manuella formateringar tycker du bör göras för att öka läsbarheten? Ge funktionen gurka ett bättre namn. Diskutera läsbarheten med en handledare.

### Uppgift 9. Tidmätning. Hur snabb är din dator?

a) Skriv in koden nedan i Kojos editor och kör upprepade gånger med den gröna play-knappen. Tar det lika lång tid varje gång? Varför?

```
object timer {
  def now: Long = System.currentTimeMillis
  var saved: Long = now
  def elapsedMillis: Long = now - saved
  def elapsedSeconds: Double = elapsedMillis / 1000.0
  def reset: Unit = { saved = now }
}

// HUVUDPROGRAM:
timer.reset
var i = 0L
while (i < 1e8.toLong) { i += 1 }
val t = timer.elapsedSeconds
println("Räknade till " + i + " på " + t + " sekunder.")
```

b) Ändra i loopen i uppgift a) så att den räknar till 4.4 miljarder. Hur lång tid tar det för din dator att räkna så långt?<sup>9</sup>

c) Om du kör på en Linux-maskin: Kör nedan Linux-kommando upprepade gånger i ett terminalfönster. Med hur många MHz kör din dators klocka för tillfället? Hur förhåller sig klockfrekvensen till antalet rundor i while-loopen i föregående uppgift? (Det kan hända att din dator kan variera centralprocessorns klockfrekvens. Prova både medan du kör tidmätningen i Kojo och då din dator "vilar". Vad är det för poäng med att en processor kan variera sin klockfrekvens?)

```
> lscpu | grep MHz
```

d) Ändra i koden i uppgift a) så att **while**-loopen bara kör 5 gånger.

e) Lägg till koden nedan i ditt program och försök ta reda på ungefär hur långt din dator hinner räkna till på en sekund för Long- respektive Int-variabler. Använd den gröna play-knappen.

```
def timeLong(n: Long): Double = {
  timer.reset
  var i = 0L
  while (i < n) { i += 1 }
  timer.elapsedSeconds
}

def timeInt(n: Int): Double = {
  timer.reset
  var i = 0
  while (i < n) { i += 1 }
  timer.elapsedSeconds
}

def show(msg: String, sec: Double): Unit = {
  print(msg + ": ")
  println(sec + " seconds")
}
```

<sup>9</sup>Det går att göra ungefär en heltalsaddition per klockcykel per kärna. Den första elektroniska datorn [Eniac](#) hade en klockfrekvens motsvarande 5 kHz. Den dator på vilken denna övningsuppgift skapades hade en i7-4790K turboklockad upp till 4.4 GHz.

```
def report(n: Long): Unit = {
  show("Long " + n, timeLong(n))
  if (n <= Int.MaxValue) show("Int  " + n, timeInt(n.toInt))
}

// HUVUDPROGRAM, mätningar:

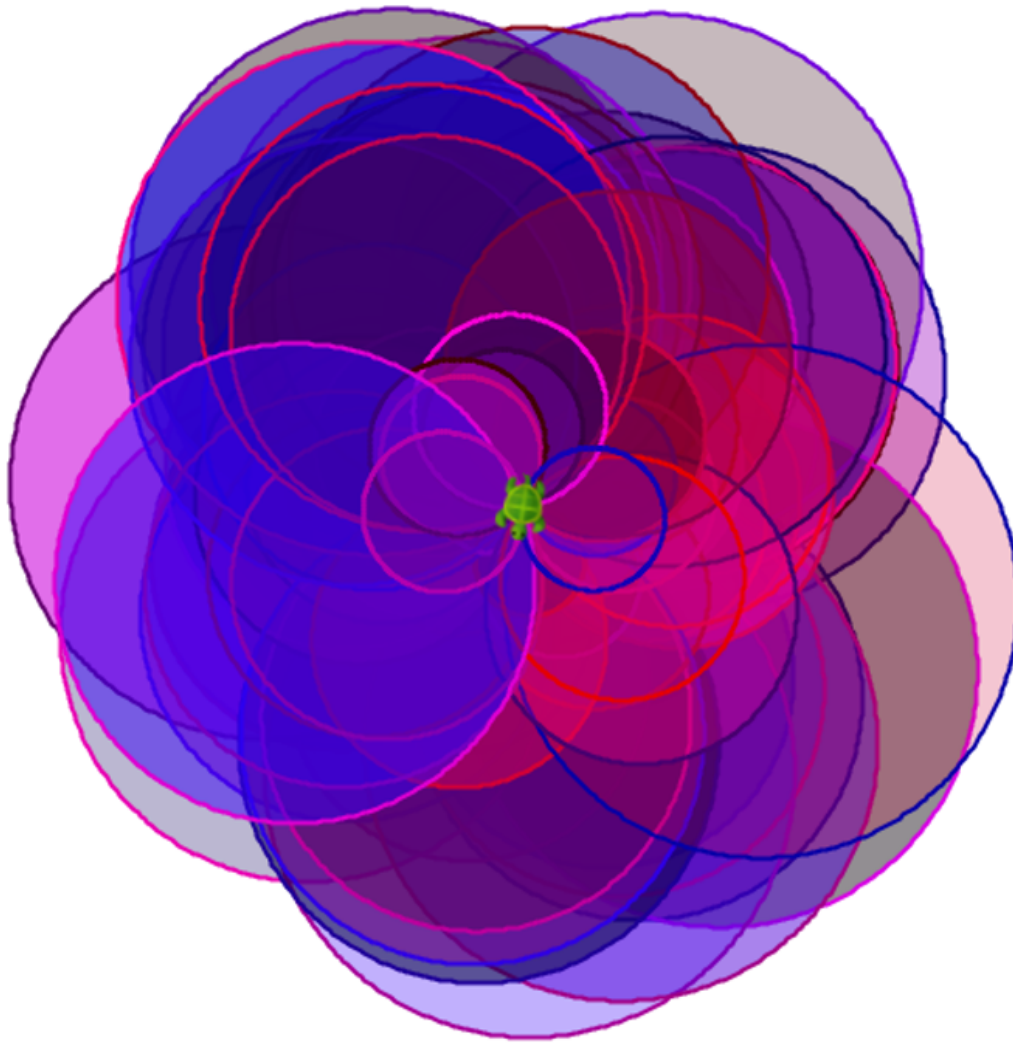
report(Int.MaxValue)
for (i <- 1 to 10) report(4.26e9.toLong)
```

f) Hur mycket snabbare går det att räkna med Int-variabler jämfört med Long-variabler? Diskutera gärna svaret med en handledare.

**Uppgift 10.** Lek med färg i Kojo. Sök på internet efter dokumentationen för klassen `java.awt.Color` och studera vilka heltalsparametrar den sista konstruktorn i listan med konstruktorer tar för att skapa sRGB-färger. Om du högerklickar i editorn i Kojo och väljer "Välj färg..." får du fram färgväljaren och med den kan du välja fördefinierade färger eller blanda egna färger. När du har valt färg får du se vilka parametrar till `java.awt.Color` som skapar färgen. Testa detta i REPL:

```
1 scala> val c = new java.awt.Color(124,10,78,100)
2 c: java.awt.Color = java.awt.Color[r=124,g=10,b=78]
3
4 scala> c. // tryck på TAB
5 asInstanceOf  getColorComponents  getRGBComponents
6 brighter      getColorSpace      getRed
7 createContext  getComponents  getTransparency
8 darker        getGreen            instanceof
9 getAlpha       getRGB              toString
10 getBlue        getRGBColorComponents
11
12 scala> c.getAlpha
13 res3: Int = 100
```

Skriv ett program som ritar många figurer med olika färger, till exempel cirklar som nedan. Om du använder alfakanalen blir färgerna genomskinliga.



**Uppgift 11.** Ladda ner "Uppdrag med Kojo" från [lth.se/programmera/uppdrag](http://lth.se/programmera/uppdrag) och gör några uppgifter som du tycker verkar intressanta.

**Uppgift 12.** Om du vill jobba med att hjälpa skolbarn att lära sig programmera med Kojo, kontakta <http://www.vattenhallen.lth.se> och anmäl ditt intresse att vara handledare.



# Kapitel 2

## Program och kontrollstrukturer

Begrepp som ingår i denna veckas studier:

- ☐ huvudprogram
- ☐ program-argument
- ☐ indata
- ☐ `scala.io.StdIn.readLine`
- ☐ kontrollstruktur
- ☐ iterera över element i samling
- ☐ for-uttryck
- ☐ yield
- ☐ map
- ☐ foreach
- ☐ samling
- ☐ sekvens
- ☐ indexering
- ☐ Array
- ☐ Vector
- ☐ intervall
- ☐ Range
- ☐ algoritm
- ☐ implementation
- ☐ pseudokod
- ☐ algoritmexempel: SWAP
- ☐ SUM
- ☐ MIN-MAX
- ☐ MIN-INDEX

## 2.1 Teori

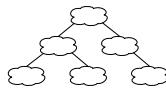
Ett program innehåller satser och uttryck. En **kontrollstruktur**, t.ex. **while**, styr i vilken **ordning** satser och uttryck exekveras. Data kan placeras i en **datastruktur**, t.ex. en Vector, så att man senare kan komma åt data igen.

### 2.1.1 Vad är en datastruktur?

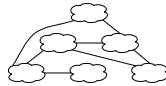
- En **datastruktur** är en struktur för organisering av data som...
  - kan innehålla **många** element,
  - kan **refereras** till som en **helhet**, och
  - ger möjlighet att **komma åt enskilda element**.
- En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.
- Exempel på olika samlingar där elementen är organiserade på olika vis:

**Sekvens** 

**Träd**



**Graf**



Mer om sekvenser & träd i [EDAA01 pfk](#). Mer om träd, grafer i [Diskreta strukturer](#).

### 2.1.2 Några samlingar i `scala.collection`

- En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.
- En **sekvens** (eng. *sequence*) är en samling där alla element är ordnade.
- Exempel på **färdiga samlingar** i Scalas standardbibliotek där elementen är organiserade internt på **olika** vis så att samlingen får olika egenskaper som passar **olika användningsområden**:
  - `scala.collection.immutable.Vector`, sekvens med snabb access **överallt**.
  - `scala.collection.immutable.List`, sekvens med snabb access **i början**.
  - `scala.collection.immutable.Set`, `scala.collection.mutable.Set`, mängd med unika element; ej i sekvens men snabb innehållstest.
  - `scala.collection.immutable.Map`, `scala.collection.mutable.Map`, mängd med par av nyckel & tillhörande värde, snabb access via nyckel.
  - `scala.collection.mutable.ArrayBuffer`, förändringsbar sekvens kan ändra storlek.
  - `scala.Array`, förändringsbar sekvens som **inte** kan ändra storlek. Alla element är lagrade efter varandra i minnet: snabbast access av alla samlingar, men har speciella begränsningar.

### 2.1.3 Olika strukturer för att hantera data

- **Tupel** (eng. *tuple*)
  - samla flera datavärden t.ex. (1, "hej", **true**) i element **\_1**, **\_2**, **\_3**
  - elementen kan vara av **olika** typ
- **Enumeration** (även kallad *uppräknings*) (eng. *enumeration*)
  - Namnge uppräknade värden t.ex. **enum** Color { **case** Red, Black }
  - Värdena har ordningsnummer och är alla av **samma** typ (här Color)
- **Klass** (eng. *class*)
  - samlar data i **attribut** med (väl valda!) namn
  - attributen kan vara av **olika** typ
  - definierar även **metoder** som använder attributen (kallas även **operationer** på data)
- **Färdig samling**
  - speciella klasser som samlar data i element av **samma** typ
  - exempel: `scala.collection.immutable.Vector`
  - har ofta *många* färdiga **bra-att-ha-metoder**, se snabbreferensen <http://cs.lth.se/pgk/quickref>
- **Egenimplementerade samlingar**
  - → fördjupningskurs

### 2.1.4 Vad är en vektor?

En **vektor**<sup>1</sup> (eng. *vector*) är en **sekvens** som är **snabb** att **indexera** i. Åtkomst av element i en sekvens som t.ex. heter `xs` sker i Scala med `xs.apply(platsnummer)`:

```

1 scala> val heltal = Vector(42, 13, -1, 0, 1)
2 val heltal: scala.collection.immutable.Vector[Int] = Vector(42, 13, -1, 0, 1)
3
4 scala> heltal.apply(0) // platsnummer räknas från noll
5 val res0: Int = 42
6
7 scala> heltal(1) // man kan i Scala skippa .apply före (
8 val res1: Int = 13
9
10 scala> heltal(5) // ger körtidsfel då sjätte platsen inte finns
11 java.lang.IndexOutOfBoundsException: 5
12 at scala.collection.immutable.Vector.checkRangeConvert(Vector.scala:132)

```

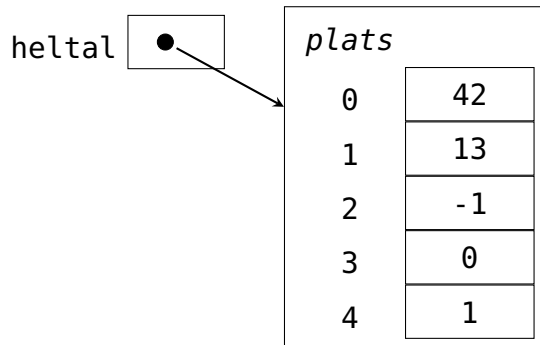
Utelämnar du `.apply` så skapar kompilatorn automatiskt ett anrop av `apply`.

### 2.1.5 En konceptuell bild av en vektor

<sup>1</sup>Vektor kallas ibland på svenska även **fält**, men det skapar stor förvirring eftersom det engelska ordet *field* ofta används för *attribut* (förklaras senare).

```
scala> val heltal = Vector(42, 13, -1, 0, 1)

scala> heltal(0)
val res0: Int = 42
```



### 2.1.6 En samling strängar

- En vektor kan lagra **många** värden av samma typ.
- Elementen kan vara till exempel heltal eller strängar.
- Eller faktiskt vad som helst. (En s.k. *generisk* samling.)

```
1 scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")
2 grönsaker: scala.collection.immutable.Vector[String] =
3   Vector(gurka, tomat, paprika, selleri)
4
5 scala> val g = grönsaker(1)
6 val g: String = tomat
7
8 scala> val xs = Vector(42, "gurka", true, 42.0)
9 val xs: Vector[Matchable] = Vector(42, gurka, true, 42.0)
```

Notera typen `Matchable` som betyder **"nästan vilken typ som helst"** (Mer om `Matchable` senare.)

### 2.1.7 Vad är en kontrollstruktur?

- En **kontrollstruktur** påverkar i vilken ordning (sekvens) satser exekveras och uttryck evalueras.

Exempel på **inbyggda** kontrollstrukturer:

**for-do**-sats

**while-do**-sats

**for-*yield***-uttryck

- I Scala kan man definiera **egna** kontrollstrukturer.

Exempel: upprepa som du använt i Kojo

```
upprepa(4){fram; höger}
```

## 2.1.8 Loopa genom elementen i en vektor

En **for-do-sats** som skriver ut alla element i en vektor:

```
1 scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")
2
3 scala> for g <- grönsaker do println(g)
4 gurka
5 tomat
6 paprika
7 selleri
```

**for ... do ...** gör så att följande händer:

- Plocka ut **varje element** ur samlingen.
- **Namnet** före pilen (här g) **refererar** till ett **nytt** värde för varje runda i loopen.
- Detta namn motsvarar en **lokal val**-variabel.

## 2.1.9 Bygg ny samling från befintlig med for-yield-uttryck

Ett **for-yield-uttryck** som **skapar en ny samling**.

```
for g <- grönsaker yield s"god $g"
```

```
1 scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")
2
3 scala> val åsikter = for (g <- grönsaker) yield s"god $g"
4 val åsikter: Vector[String] =
5   Vector(god gurka, god tomat, god paprika, god selleri)
```

## 2.1.10 Samlingen Range håller reda på intervall

- Med en `Range(start, slut)` kan du skapa ett **intervall**:  
från och med start till (men inte med) slut

```
scala> Range(0, 42)
val res0: Range =
  Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
    15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
    29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41)
```

- Men alla värden däremellan skapas inte förrän de behövs:

```
1 scala> val jättestortIntervall = Range(0, Int.MaxValue)
2 val jättestortIntervall: Range = Range(0, 1, 2, 3, 4, 5, ...
3
4 scala> jättestortIntervall.end
5 val res1: Int = 2147483647
6
7 scala> jättestortIntervall.toVector
8 java.lang.OutOfMemoryError: GC overhead limit exceeded
```

### 2.1.11 Loopa med Range

Range används i for-loopar för att hålla reda på antalet rundor.

```
scala> for i <- Range(0, 6) do print(s" gurka $i")
gurka 0 gurka 1 gurka 2 gurka 3 gurka 4 gurka 5
```

Du kan skapa en Range med until efter ett heltal:

```
scala> 1 until 7
val res1: Range =
  Range(1, 2, 3, 4, 5, 6)

scala> for i <- 1 until 7 do print(s" tomat $i")
tomat 1 tomat 2 tomat 3 tomat 4 tomat 5 tomat 6
```

### 2.1.12 Loopa med Range skapad med to

Med to efter ett heltal får du en Range till och **med** sista:

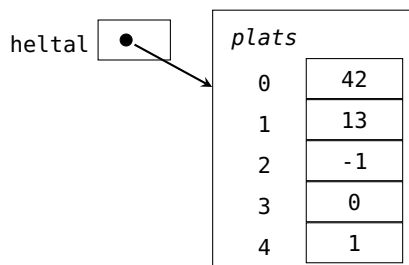
```
scala> 1 to 6
res2: Range.Inclusive =
  Range(1, 2, 3, 4, 5, 6)

scala> for i <- 1 to 6 do print(" gurka " + i)
gurka 1 gurka 2 gurka 3 gurka 4 gurka 5 gurka 6
```

### 2.1.13 Vad är en Array?

- En **Array** liknar en Vector men har en särställning i JVM:
  - Lagras som en sekvens i minnet på efterföljande adresser.
  - **Fördel**: snabbaste samlingen för element-access i JVM.
  - Men det finns en hel del **nackdelar** som vi ska se senare.

```
scala> val heltal = Array(42, 13, -1, 0 , 1)
```



### 2.1.14 Några likheter & skillnader mellan Vector och Array

```
scala> val xs = Vector(1,2,3)
```

```
scala> val xs = Array(1,2,3)
```

### Några likheter mellan Vector och Array

- Båda är samlingar som kan innehålla många element.
- Med båda kan man snabbt accessa vilket element som helst: `xs(2)`

Några viktiga skillnader:

#### Vector

- Är **oföränderlig**: du kan lita på att elementreferenserna aldrig någonsin kommer att ändras.
- Är **snabb på att skapa en delvis förändrad kopia**, t.ex. tillägg/borttagning/uppdatering mitt i sekvensen.

#### Array

- Är **föränderlig**: `xs(2) = 42`
- Är **snabb** om man bara vill läsa eller skriva på befintliga platser.
- Är **långsam** om man vill lägga till eller ta bort element mitt i sekvensen.
- Kan **ej** ändra storlek.

---

### 2.1.15 Kompilering i terminalen

När du ska skriva kod i en editor, kompilera i terminalen och köra ditt program som en **fristående applikation**, så behövs:

- En editor: **VS Code** med tillägget **Scala (Metals)**
- Körmiljön **OpenJDK**
- Kommandoverktyg för terminalen: **scala-cli**
- Se instruktioner här: <http://cs.lth.se/pgk/verktyg>
- Läs mer i Appendix C.
- Tips om du kör Windows: installera nya Windows Terminal

Be om hjälp i #frågor-och-svar-textkanalen på vår Discord-server eller fråga handledare på resurstid.

---

### 2.1.16 Scala Command Line Interface (CLI)

- Utvecklingen av ett nytt kommandogränssnitt (eng. *Command Line Interface (CLI)*) för Scala startades 2022 i ett öppen-källkodsprojekt som leds av Virtuslab.
- Under 2023 ersätter **scala-cli** det gamla **scala**-kommandot.<sup>2</sup>
- Läs mer i Appendix C och F, samt här: <https://scala-cli.virtuslab.org/>
- Du kan se vad Scala CLI kan göra via hjälp-optionen:

```
> scala-cli help
```

---

<sup>2</sup>I skrivande stund så har skiftet ännu inte skett. När det sker kan du ersätta alla förekomster av `scala-cli` med det kortare `scala`.

### 2.1.17 Ett minimalt fristående program i Scala

Spara nedan Scala-kod i filen `hej.scala`:

```
@main def run = println("Hej Scala!")
```

Spara i filen `hej.scala`, kompilera och kör i terminalen:

```
1 > cat hej.scala
2 @main def run = println("Hej Scala!")
3
4 > scala-cli run hej.scala
5 Hej Scala!
```

Innan körning kompileras dina kodfiler. Du kan se maskinkoden här:

```
1 > ls .scala-build/*/classes/main
2 'hej$package.class' 'hej$package$.class' 'hej$package.tasty' run.class run.tasty
```

---

### 2.1.18 Loopa genom en samling med en `while`-sats

```
scala> val xs = Vector("Hej", "på", "dej", "!!!")
val xs: Vector[String] =
  Vector(Hej, på, dej, !!!)

scala> xs.size
val res0: Int = 4

scala> var i = 0
val i: Int = 0

scala> while i < xs.size do { println(xs(i)); i = i + 1 }
Hej
på
dej
!!!
```

---

### 2.1.19 Strängargument till i ett program med primitiv `main`

Skriv och spara nedan kod i filen `helloargs1.scala`

```
> code helloargs1.scala
```

```
object HelloScalaArgs:
  def main(args: Array[String]): Unit = // en primitiv main-metod utan @main
    var i = 0
    while i < args.size do
      println(args(i))
      i = i + 1
```

En primitiv `main`-metod har ej `@main` och måste vara i ett objekt.  
Kompilera och kör med programargument efter --



```
1 > scala-cli run helloargs1.scala -- morot gurka tomat
2 morot
3 gurka
4 tomat
```

### 2.1.20 Typsäkra argument till i ett program med @main

Skriv och spara nedan kod i filen `helloargs2.scala`

```
> code helloargs2.scala
```

```
@main def hej(heltal: Int, resten: String*): Unit = // notera * efter String
  for i <- 0 until heltal do println(resten(i))
```

Med `@main` behövs inget objekt.

Kompilera och kör med programargument efter --

```
1 > scala-cli run helloargs2.scala -- 2 morot gurka tomat
2 morot
3 gurka
4 > scala-cli run helloargs2.scala -- aj morot gurka tomat
5 Illegal command line: java.lang.NumberFormatException: For input string: "aj"
```

Med `@main` genereras automatiskt en primitiv main som kollar att argumenten har rätt typ.

### 2.1.21 För kännedom: Scala-skript

- Scala-kod kan köras som ett **skript**.<sup>3</sup>
- Ett skript finns i en enda fristående fil med ändelsen `.sc`
- Skript behöver inget huvudprogram.
- Skript har automatiskt alla programargument i strängsekvensen `args`

```
// spara detta i filen 'myscript.sc'
println("Hej alla mina argument:")
for a <- args do println(s"Hej: $a")
```

```
> scala-cli run myscript.sc -- ett två tre
Hej alla mina argument:
Hej: ett
Hej: två
Hej: tre
```

---

<sup>3</sup>En nackdel med Scala-skript är att de ej kan inkludera skript i andra kodfiler.

### 2.1.22 Vad är en algoritm?

En **algoritm** är en sekvens av instruktioner som beskriver hur man löser ett problem.

#### Exempel:

- baka en kaka
- räkna ut din pensionsprognos
- köra bil
- kolla om highscore i ett spel

### 2.1.23 Algoritmexempel: N-FAKULTET

**Indata** : heltalet  $n$

**Utdata**: produkten av de första  $n$  positiva heltalen

```

1
2  $prod \leftarrow 1$ 
3  $i \leftarrow 2$ 
4 while  $i \leq n$  do
5   |  $prod \leftarrow prod * i$ 
6   |  $i \leftarrow i + 1$ 
7 end
8  $prod$ 
```

- Vad händer om  $n$  är noll?
- Vad händer om  $n$  är ett?
- Vad händer om  $n$  är två?
- Vad händer om  $n$  är tre?

### 2.1.24 Algoritmexempel: MIN

**Indata** : Array  $args$  med strängar som alla innehåller heltal

**Utdata**: minsta heltalet

```

1
2  $min \leftarrow$  det största heltalet som kan uppkomma
3  $n \leftarrow$  antalet heltal
4  $i \leftarrow 0$ 
5 while  $i < n$  do
6   |  $x \leftarrow args(i).toInt$ 
7   | if  $(x < min)$  then
8     |  $min \leftarrow x$ 
9   | end
10  |  $i \leftarrow i + 1$ 
11 end
12  $min$ 
```

**Testa med indata:** args = Array("2", "42", "1", "2")

En program delas ofta upp i många olika **funktioner**. En funktion kan ha parametrar och ge ett returvärde. Om du delar upp ditt program i många enkla funktioner med bra namn, så blir ditt program lättare att läsa och begripa. Om en vältestad och buggfri funktion användas på flera ställen, så kan risken för buggar minskas.

### 2.1.25 Mall för funktionsdefinitioner

**def** funktionsnamn(parameterdeklarationer): returtyp = uttryck

**Exempel:**

```
def öka(i: Int): Int = i + 1
```

Returtypen kan härledas av kompilatorn:

```
def öka(i: Int) = i + 1
```

Men för att få hjälp av kompilatorn är det bra att ange returtyp!

Om flera parametrar använd kommatecken. Om flera satser använd indentering (och eventuell valfria klammerparenteser).

```
def isHighscore(points: Int, high: Int): Boolean = {
  val highscore: Boolean = points > high
  if highscore then println(":)") else println(":(")
  highscore
}
```

Ovan funktion har **sidoeffekten** att skriva ut en smiley.

### 2.1.26 Bättre många små abstraktioner som gör en sak var

```
def isHighscore(points: Int, high: Int): Boolean = points > high

def printSmiley(isHappy: Boolean): Unit =
  if isHappy then println(":)") else print(":(")
```

```
printSmiley(isHighscore(113,99))
```

- Denna bättre isHighscore är nu en **äkta funktion** som alltid ger samma svar för samma inparametrar och **saknar sidoeffekter**; dessa funktioner är ofta lättare att förstå.
- Funktioner som ger ett booleskt värde kallas för **predikat**.

### 2.1.27 Vad är ett block?

- Ett block **kapslar in** flera satser/uttryck och ser "utifrån" ut som en enda sats/uttryck.
- Ett block skapas med hjälp av klammerparenteser ("krullparenteser")  
{ uttryck1; uttryck2; ... uttryckN }

- I Scala (till skillnad från många andra språk) har ett block ett **värde** och är alltså ett **uttryck**.
- Värdet ges av **sista uttrycket** i blocket.

```
scala> val x = { println(1 + 1); println(2 + 2); 3 + 3 }
2
4
x: Int = 6
```

### 2.1.28 Namn i block blir lokala

Synlighetsregler:

1. Identifierare deklarerade inuti ett block blir **lokala**.
2. Lokala namn **överskuggar** namn i yttre block om samma.
3. Namn syns i nästlade underblock.

```
1 scala> def a = { val lokaltNamn = 42; println(lokaltNamn) }
2 scala> a
3 42
4
5 scala> println(lokaltNamn)
6 1 |println(lokaltNamn)
7   |      ^^^^^^^^^^^
8   |      Not found: lokaltNamn
9
10 scala> def b = { val x = 42; { val x = 76; println(x) }; println(x) }
11 scala> def c = { val x = 42; { val b = x + 1; println(b) } }
12 scala> b // vad händer?
13 scala> c // vad händer?
```

### 2.1.29 Parameter och argument

Skilj på parameter och argument!

- En **parameter** är det deklarerade namnet som används **lokalt** i en funktion för att referera till...
- **argumentet** som är värdet som skickas med **vid anrop** och binds till det lokala parameternamnet.

```
scala> val ettArgument = 42

scala> def öka(minParameter: Int) = minParameter + 1

scala> öka(ettArgument)
```

Speciell syntax: anrop med s.k. **namngivet argument**

```
scala> öka(minParameter = ettArgument)
```

Namngivna argument kan ges i valfri ordning; då riskerar man inte fel ordning.

### 2.1.30 Procedurer

- En **procedur** är en funktion som **gör** något intressant, men som **inte** lämnar något intressant returvärde.
- Exempel på befintlig procedur: `println("hej")`
- Du **deklarerar egna procedurer** genom att ange **Unit** som returvärdestyp. Då ges värdet **()** som betyder "inget".

```
scala> def hej(x: String): Unit = println(s"Hej på dej $x!")

scala> hej("Herr Gurka")
Hej på dej Herr Gurka!

scala> val x = hej("Fru Tomat")
Hej på dej Fru Tomat!

scala> :type x
Unit

scala> println(x)    // vad händer?
```

- Det som **görs** kallas (sido)**effekt**. Ovan är utskriften själva effekten.
- Funktioner kan också ha sidoeffekter. De kallas då **oäkta** funktioner.

---

### 2.1.31 "Ingenting" är faktiskt någonting i Scala

- I många språk (Java, C, C++, ...) är funktioner som saknar värden speciella. Java m.fl. har speciell syntax för procedurer med nyckelordet **void**, men **inte** Scala.
- I Scala är procedurer inte specialfall; de är vanliga funktioner som returnerar ett värde som **representerar** ingenting, nämligen **()** som är av typen **Unit**.
- På så sätt blir procedurer inget undantag utan följer vanlig syntax och semantik precis som för alla andra funktioner.
- Detta är typiskt för Scala: generalisera koncepten och vi slipper besvärliga undantag! (Men vi måste förstå generaliseringen...) [https://en.wikipedia.org/wiki/Void\\_type](https://en.wikipedia.org/wiki/Void_type) [https://en.wikipedia.org/wiki/Unit\\_type](https://en.wikipedia.org/wiki/Unit_type)

---

### 2.1.32 Problemlösning: nedbrytning i abstraktioner som sen kombineras

- En av de allra viktigaste principerna inom programmering är **funktionell nedbrytning** där **underprogram** i form av funktioner och procedurer skapas för att bli byggstenar som kombineras till mer avancerade funktioner och procedurer.
- Genom de namn som definieras skapas **återanvändbara abstraktioner** som kapslar in det funktionen gör.
- Problemet blir med bra byggblock lättare att lösa.
- Abstraktioner som beräknar eller gör **en enda, väldefinierad sak** är enklare att använda, jämfört med de som gör många, helt olika saker.

- Abstraktioner med **välgenomtänkta namn** är enklare att använda, jämfört med kryptiska eller missvisande namn.

---

### 2.1.33 Exempel på funktionell nedbrytning

Kojo-labben gav exempel på **funktionell nedbrytning** där ett antal abstraktioner skapas och återanvänds.

```
// skapa abstraktioner som bygger på varandra

def kvadrat = upprepa(4){fram; höger}

def stapel = {
  upprepa(10){kvadrat; hoppa}
  hoppa(-10*25)
}

def rutnät = upprepa(10){stapel; höger; fram; vänster}

// huvudprogram

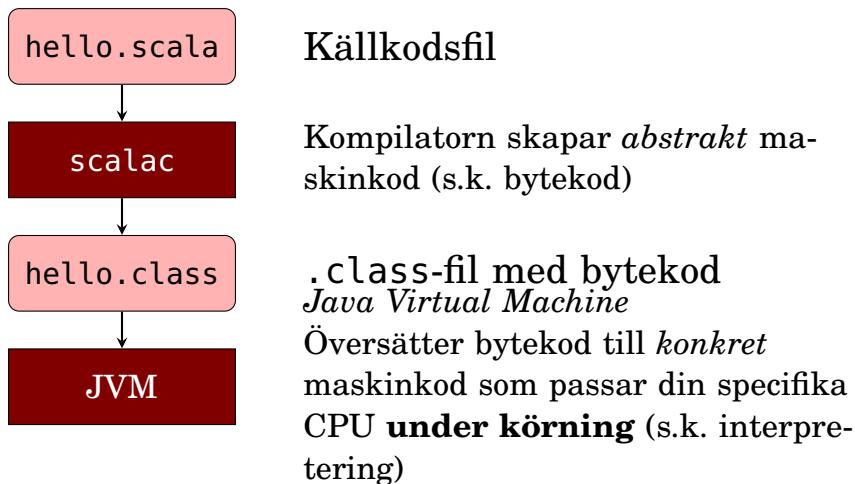
sudda; sakta(200)
rutnät
```

---

### 2.1.34 Varför abstraktion?

- Stora program behöver delas upp annars blir det mycket svårt att förstå och bygga vidare på programmet.
- Vi behöver kunna välja namn på saker i koden *lokalt*, utan att det krockar med samma namn i andra delar av koden.
- Abstraktioner hjälper till att hantera och kapsla in komplexa delar så att de blir enklare att använda om och om igen.
- Exempel på **abstraktionsmekanismer** i Scala:
  - **Klasser** är "byggblock" med kod som används för att skapa **objekt**, innehållande delar som hör ihop.  
Nyckelord: **class** och **object**
  - **Metoder** är funktioner som finns i klasser/objekt och används för att lösa specifika uppgifter. Nyckelord: **def**
  - **Paket** används för att organisera kodfiler i en hierarkisk katalogstruktur och skapa namnrymder.  
Nyckelord: **package**

### 2.1.35 Från källkod till maskinkod med JVM



### 2.1.36 Paket

```
package greeting

@main def run = println("Hello world!")
```

- Paket (eng. *package*) ger struktur åt koden och skapar namnrymder.
  - Paket kan vara **nästlade**: ofta finns paket i paket i paket.
  - Paket är speciellt bra om man har mycket kod i många kodfiler.
  - Kompilatorn placerar maskinkoden i kataloger enligt paketstrukturen.<sup>4</sup>
- Är du nyfiken, kolla underkataloger i `.scala-build`:

```
ls -R .scala-build
```

### 2.1.37 Import

Med hjälp av punktnotation kommer man åt innehåll i ett paket.

```
val age = scala.io.StdIn.readLine("Ange din ålder:")
```

En **import**-sats...

```
import scala.io.StdIn.readLine
```

...gör så att kompilatorn "ser" namnet, och man slipper skriva hela sökvägen till namnet:

```
val age = readLine("Ange din ålder:")
```

Man säger att det importerade namnet hamnar *in scope*.

<sup>4</sup>Katalogstrukturen för källkoden *måste* i många andra språk, t.ex. Java, *exakt motsvara paketstrukturen*, men detta är inte nödvändigt i Scala – alla Scala-kodfiler kan ligga i samma katalog på toppnivå eller i underkatalog med valfritt namn, oavsett hur din kod använder **package**.

### 2.1.38 Jar-filer

- jar-filer liknar zip-filer och används för att sammanföra många kompillerade kodfiler i **en komprimerad fil** för enkel distribution och körning.
- Du använder jar-filer med optionen `--jar`

```
scala-cli run . --jar introprog.jar
```

- Du kan skapa egna jar-filer med `scala-cli package`

```
scala-cli package . --library --output myapp.jar  
scala-cli run --jar myapp.jar
```

Läs mer om jar-filer i Appendix F.

---



## 2.2 Övning programs

### Mål

- ☐ Kunna skapa, kompilera och köra en enkel applikation i terminalen.
- ☐ Kunna skapa samlingarna Range, Array och Vector med heltal och strängar.
- ☐ Kunna indexera i en indexerbar samling, t.ex. Array och Vector.
- ☐ Kunna anropa operationerna size, mkString, sum, min, max på samlingar som innehåller heltal.
- ☐ Känna till skillnader och likheter mellan samlingarna Range, Array och Vector.
- ☐ Förstå skillnaden mellan en while-sats och ett for-uttryck.
- ☐ Kunna skapa samlingar med heltalsvärden som resultat av enkla for-uttryck.
- ☐ Förstå skillnaden mellan en algoritm i pseudo-kod och dess implementation.
- ☐ Kunna implementera algoritmerna SUM, MIN, MAX med en indexerbar samling och en while-sats.

### Förberedelser

- ☐ Studera begreppen i kapitel 2
- ☐ Bekanta dig med grundläggande terminalkommandon, se appendix B.
- ☐ Bekanta dig med VS Code, se appendix C.
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).

### 2.2.1 Grunduppgifter

**Uppgift 1.** *Para ihop begrepp med beskrivning.*

Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

kompilera	1	A	kan överföras via parametern args till main
skript	2	B	många olika element i en helhet; elementvis åtkomst
objekt	3	C	datastruktur med element av samma typ
@main	4	D	en specifik realisering av en algoritm
programargument	5	E	en oföränderlig, indexerbar sekvenssamling
datastruktur	6	F	datastruktur med element i en viss ordning
samling	7	G	samlar variabler och funktioner
sekvenssamling	8	H	applicerar en funktion på varje element i en samling
Array	9	I	maskinkod skapas ur en eller flera källkodsfiler
Vector	10	J	ensam kodfil, huvudprogram behövs ej
Range	11	K	en förändringsbar, indexerbar sekvenssamling
yield	12	L	där exekveringen av kompilerat program startar
map	13	M	stegvis beskrivning av en lösning på ett problem
algoritm	14	N	en samling som representerar ett intervall av heltal
implementation	15	O	används i for-uttryck för att skapa ny samling

**Uppgift 2.** *Använda terminalen.* Läs om terminalen i appendix B.

- a) Vilka tre kommando ska du köra för att 1) skapa en katalog med namnet hello och 2) navigera till katalogen och 3) visa namnet på ut aktuell katalog? Öppna ett terminalfönster och kör dessa tre kommando.
- b) Vilka två kommando ska du köra för att 1) navigera tillbaka ”upp” ett steg i filträdet och 2) lista alla filer och kataloger på denna plats? Kör dessa två kommando i terminalen.

**Uppgift 3.** *Skapa och köra ett Scala-skript.*

- a) Skapa en fil med namn `sum.sc` i katalogen `hello` som du skapade i föregående uppgift med hjälp av en editor, t.ex. VS code.

```
> cd hello
> code sum.sc
```

Filen ska innehålla följande rader:

```
val n = 1000
val summa = (1 to n).sum
println(s"Summan av de $n första talen är: $summa")
```

Spara filen och kör kommandot `scala-cli run sum.sc` i terminalen:

```
> scala-cli run sum.sc
```

Vad blir summan av de 1000 första talen?

- b) Ändra i filen `sum.sc` så att högerparentesen på sista raden saknas. Spara filen (Ctrl+S) och kör skriptfilen igen i terminalen (pil-upp). Hur lyder felmeddelandet? Är det ett körtidsfel eller ett kompileringsfel?
- c) Ändra i `sum.sc` så att det i stället för 1000 står `args(0).toInt` efter `val n =` och spara och kör om ditt program med argumentet 5001 så här:

```
1 > scala-cli run sum.sc -- 5001
```

Vad blir summan av de 5001 första talen?

- d) Vad blir det för felmeddelande om du glömmer att ge skriptet ett argument? Är det ett körtidsfel eller ett kompileringsfel?

**Uppgift 4.** *Scala-applikation med @main.* Skapa med hjälp av en editor en fil med namn `hello.scala`.

```
> code hello.scala
```

Skriv nedan kod i filen:

```
@main def run(): Unit = {
  val message = "Hello world!"
  println(message)
}
```

- a) Kompilera med `scala-cli compile hello.scala`. Vad heter filerna som kompilatorn skapar? Leta efter filer som slutar med `.class` i mapparna som ligger under mappen som börjar med `project...`

```
> scala-cli compile hello.scala
> ls .scala-build/project*/classes/main/
```

b) Hur ska du ändra i din kod så att kompilatorn ger följande felmeddelande:  
Syntax Error: '}' expected, but eof found?

c) I Scala är klammerparenteser valfria (eng. *optional braces*) och koden struktureras istället i sammanhängande block med hjälp av indenteringar<sup>5</sup>. Det går bra att byta mellan stilarna i samma fil om du tycker detta gör koden mer lättläst.

Ovan kod kan skrivas:

```
@main def run(): Unit =
  val message = "Hello world!"
  println(message)
```

Vad händer om du tar bort indenteringen på den sista raden?

d) Vad betyder @main-annoteringen?

**Uppgift 5.** *Skapa och använda samlingar.* I Scalas standardbibliotek finns många olika samlingar som går att använda på ett enhetligt sätt (med vissa undantag för Array). Para ihop uttrycken som skapar eller använder samlingar med förklaringarna, så att alla kopplingar blir korrekta (minst en förklaring passar med mer än ett uttryck, men det finns bara en lösning där alla kopplingar blir parvis korrekta):

<code>val xs = Vector(2)</code>	1	A	ny samling med en nolla tillagd på slutet
<code>val ys = Array.fill(9)(0)</code>	2	B	ny samling, elementen omgjorda till heltal
<code>Vector.fill(9&gt;(' '))</code>	3	C	ny referens till förändringsbar sekvens
<code>xs(0)</code>	4	D	ny samling, elementen omgjorda till strängar
<code>xs.apply(0)</code>	5	E	förkortad skrivning av <code>apply(0)</code>
<code>xs :+ 0</code>	6	F	indexering, ger första elementet
<code>0 +: xs</code>	7	G	ny sträng med komma mellan elementen
<code>ys.mkString</code>	8	H	ny samling med en nolla tillagd i början
<code>ys.mkString(",")</code>	9	I	ny referens till sekvens av längd 1
<code>xs.map(_.toString)</code>	10	J	ny oföränderlig sekvens med blanktecken
<code>xs.map(_.toInt)</code>	11	K	ny sträng med alla element intill varandra

Träna med dina egna varianter i REPL tills du lärt dig använda uttryck som ovan utantill. Då har du lättare att komma igång med kommande laborationer.

**Uppgift 6.** *Jämför Array och Vector.* Para ihop varje samlingstyp med den beskrivning som passar bäst:

a) Vad gäller angående föränderlighet (eng. *mutability*)?

<sup>5</sup>Valfria klammerparenteser och signifikant indentering kom med nya Scala 3. I gamla Scala 2 var klammerparenteser nödvändiga om flera satser ska kombineras och indenteringen påverkade inte betydelsen.

Vector	1	A	förändringsbar
Array	2	B	oföränderlig

b) Vad gäller vid tillägg av element i början (eng. *prepend*) och slutet (eng. *append*), eller förändring av delsekvens på godtycklig plats (eng. *to patch*, även på svenska: *att patcha*)?

Vector	1	A	långsam vid ändring av storlek (kopiering av rubbet krävs)
Array	2	B	varianter med fler/andra element skapas snabbt ur befintlig

c) Vad gäller vid likhetstest (eng. *equality test*).

Vector	1	A	<code>xs == ys</code> är <b>true</b> om alla element lika
Array	2	B	olikt andra Scala-samlingar kollar <code>==</code> ej innehållslikhet

**Uppgift 7.** *Räkna ut summa, min och max i args.* Skriv ett program som skriver ut summa, min och max för en sekvens av heltal i args. Du kan förutsätta att programmet bara körs med heltal som programparametrar. *Tips:* Med uttrycken `args.sum` och `args.min` och `args.max` ges summan, minsta resp. största värde.

Exempel på körning i terminalen:

```
1 > code sum-min-max.scala
2 > scala-cli run sum-min-max.scala -- 1 2 42 3 4
3 52 1 42
```

Vad blir det för felmeddelande om du ger argumentet hej när ett heltal förväntas?

**Uppgift 8.** *Algoritmen: SWAP.* Det är vanligt när man arbetar med förändringsbara datastrukturer att man kan behöva byta plats mellan element och då behövs algoritmen SWAP, som här illustreras genom platsbyte mellan värden:

*Problem:* Byta plats på två variablers värden.

*Lösningssidé:* Använd temporär variabel för mellanlagring.

a) Skriv med *pseudo-kod* (steg för steg på vanlig svenska) algoritmen SWAP nedan.

*Indata:* två heltalsvariabler *x* och *y*

**???**

*Utdata:* variablerna *x* och *y* vars värden har bytt plats.

b) Implementerar algoritmen SWAP. Ersätt ??? nedan med kod som byter plats på värdena i variablerna *x* och *y*:

```
1 scala> var x = 42; var y = 43
2 scala> ???
3 scala> println(s"x är $x, y är $y")
4 x är 43, y är 42
```

**Uppgift 9.** *Indexering och tilldelning i Array med SWAP.* Skriv ett program som byter plats på första och sista elementet i parametern args. Bytet ska bara ske om det är minst två element i args. Oavsett om förändring skedde eller ej ska args sedan skrivas

ut med blanktecken mellan argumenten. *Tips:* Du kan komma åt sista elementet med `args(args.length - 1)`

Exempel på körning i terminalen:

```
1 > code swap-args.scala
2 > scala-cli run swap-args.scala -- hej alla barn
3 barn alla hej
```

**Uppgift 10.** *for-uttryck och map-uttryck.* Variabeln `xs` nedan refererar till samlingen `Vector(1, 2, 3)`. Para ihop uttrycken till vänster med rätt värde till höger.

<code>for x &lt;- xs yield x * 2</code>	1	A	<code>Vector(2, 4, 6)</code>
<code>for i &lt;- xs.indices yield i</code>	2	B	<code>Vector(1, 2)</code>
<code>xs.map(x =&gt; x + 1)</code>	3	C	<code>Vector(1, 2, 3)</code>
<code>for i &lt;- 0 to 1 yield xs(i)</code>	4	D	<code>Vector(2, 3, 4)</code>
<code>(1 to 3).map(i =&gt; i)</code>	5	E	<code>Vector(0, 1, 2)</code>
<code>(1 until 3).map(i =&gt; xs(i))</code>	6	F	<code>Vector(2, 3)</code>

Träna med dina egna varianter i REPL tills du lärt dig använda uttryck som ovan utantill. Då har du lättare att komma igång med kommande laborationer.

**Uppgift 11.** *Algoritm: SUMBUG* . Nedan återfinns pseudo-koden för SUMBUG.

```
Indata : heltalet  $n$ 
Utdata: summan av de positiva heltalen 1 till och med  $n$ 
1  $sum \leftarrow 0$ 
2  $i \leftarrow 1$ 
3 while  $i \leq n$  do
4    $sum \leftarrow sum + 1$ 
5 end
6  $sum$ 
```

a) Kör algoritmen steg för steg med penna och papper, där du skriver upp hur värdena för respektive variabel ändras. Det finns två buggar i algoritmen. Vilka? Rätta buggarna och testa igen genom att "köra" algoritmen med penna på papper och kontrollera så att algoritmen fungerar för  $n = 0$ ,  $n = 1$ , och  $n = 5$ . Vad händer om  $n = -1$ ?

b) Skapa med hjälp av en editor filen `sumn.scala`. Implementera algoritmen SUM enligt den rättade pseudokoden och placera implementationen i en `@main`-annoterad metod med namnet `sumn`. Du kan skapa indata  $n$  till algoritmen med denna deklaration i början av din metod:

```
val n = args(0).toInt
```

eller direkt ha  $n$  som parameter till metoden.

Vad ger applikationen för utskrift om du kör den med argumentet 8888?

```
scala-cli sumn.scala -- 8888
```

Kontrollera att din implementation räknar rätt genom att jämföra svaret med detta uttrycks värde, evaluerat i Scala REPL:

```
scala> (1 to 8888).sum
```

## 2.2.2 Extrauppgifter; träna mer

**Uppgift 12.** *Algoritm: MAXBUG* . Nedan återfinns pseudo-koden för MAXBUG.

```

Indata : Array args med strängar som alla innehåller heltal
Utdata : största heltalet
1 max ← det minsta heltalet som kan uppkomma
2 n ← antalet heltal
3 i ← 0
4 while i < n do
5   | x ← args(i).toInt
6   | if (x > max) then
7   |   | max ← x
8   | end
9 end
10 max

```

- Kör med penna och papper. Det finns en bugg i algoritmen ovan. Vilken? Rätta buggen.
- Implementera algoritmen MAX (utan bugg) som en Scala-applikation. Tips:
  - Det minsta Int-värdet som någonsin kan uppkomma: `Int.MinValue`
  - Antalet element i *args* ges av: `args.length` eller `args.size`

```

1 > code maxn.scala
2 > scala-cli maxn.scala -- 7 42 1 -5 9
3 42

```

- Skriv om algoritmen så att variabeln *max* initialiseras med det första talet i sekvensen.
- Implementera den nya algoritmvarianten från uppgift c och prova programmet. Se till att programmet fungerar även om *args* är tom.

**Uppgift 13.** *Algoritm MIN-INDEX*. Implementera algoritmen MIN-INDEX som söker index för minsta heltalet i en sekvens. Pseudokod för algoritmen MIN-INDEX:

```

Indata : Sekvens xs med n st heltal.
Utdata : Index för det minsta talet eller -1 om xs är tom.
1 minPos ← 0
2 i ← 1
3 while i < n do
4   | if xs(i) < xs(minPos) then
5   |   | minPos ← i
6   | end
7   | i ← i + 1
8 end
9 if n > 0 then
10 | minPos
11 else
12 | -1
13 end

```

- Prova algoritmen med penna och papper på sekvensen (1,2,-1,4) och rita minnessituationen efter varje runda i loopen. Vad blir skillnaden i exekveringsförloppet om loopvariabeln

*i* initialiserats till 0 i stället för 1?

b) Implementera algoritmen MIN-INDEX i ett Scala-program med nedan funktion:

```
def indexOfMin(xs: Array[Int]): Int = ???
```

- Låt programmet ha en main-funktion som ur args skapar en ny array med heltal som skickas till `indexOfMin` och sedan gör en utskrift av resultatet.
- Testa för olika fall:
  - tom sekvenser
  - sekvens med endast ett tal
  - lång sekvens med det minsta talet först, någonstans mitt i, samt sist.

**Uppgift 14.** *Datastrukturen Range.* Evaluera nedan uttryck i Scala REPL. Vad har respektive uttryck för värde och typ?

- `Range(1, 10)`
- `Range(1, 10).inclusive`
- `Range(0, 50, 5)`
- `Range(0, 50, 5).size`
- `Range(0, 50, 5).inclusive`
- `Range(0, 50, 5).inclusive.size`
- `0.until(10)`
- `0 until (10)`
- `0 until 10`
- `0.to(10)`
- `0 to 10`
- `0.until(50).by(5)`
- `0 to 50 by 5`
- `(0 to 50 by 5).size`
- `(1 to 1000).sum`

### 2.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 15.** *Sten-Sax-Påse-spel.* Bygg vidare på koden nedan och gör ett Sten-Sax-Påse-spel<sup>6</sup>. Koden fungerar som den ska, förutom funktionen `winner` som fuskar till datorns fördel. Lägg även till en `main`-funktion så att programmet kan kompileras och köras i terminalen. Spelet blir roligare om du räknar antalet vinster och förluster. Du kan också göra så att datorn inte väljer med jämn fördelning.

```
object Game:
  val choices = Vector("Sten", "Påse", "Sax")

  def printChoices(): Unit =
    for i <- 1 to choices.size do println(s"$i: ${choices(i - 1)}")

  def userChoice(): Int =
    printChoices()
    scala.io.StdIn.readLine("Vad väljer du? [1|2|3]<ENTER>:").toInt - 1

  def computerChoice(): Int = (math.random() * 3).toInt

  /** Ska returnera "Du", "Datorn", eller "Ingen" */
  def winner(user: Int, computer: Int): String = "Datorn"

  def play(): Unit =
    val u = userChoice()
    val c = computerChoice()
    println(s"Du valde ${choices(u)}")
    println(s"Datorn valde ${choices(c)}")
    val w = winner(u, c)
    println(s"$w är vinnare!")
    if w == "Ingen" then play()
```

★ **Uppgift 16.** *Jämför exekveringstiden för storleksförändring mellan Array och Vector.* Klistra in nedan kod i REPL:

```
def time(block: => Unit): Double =
  val t = System.nanoTime
  block
  (System.nanoTime - t) / 1e6 // ger millisekunder
```

a) Skriv kod som gör detta i tur och ordning:

1. deklarerar en `val` `as` som är en `Array` fylld med en miljon heltalsnollor,
2. deklarerar en `val` `vs` som är en `Vector` fylld med en miljon heltalsnollor,
3. kör `time(as :+ 0)` 10 gånger och räknar ut medelvärdet av tidmätningarna,
4. kör `time(vs :+ 0)` 10 gånger och räknar ut medelvärdet av tidmätningarna.

b) Vilken av `Array` och `Vector` är snabbast vid tillägg av element? Varför är det så?

<sup>6</sup>[https://sv.wikipedia.org/wiki/Sten,\\_sax,\\_påse](https://sv.wikipedia.org/wiki/Sten,_sax,_påse)



- ★ **Uppgift 17.** *Minnesåtgång för Range.* Datastrukturen `Range` håller reda på start- och slutvärde, samt stegstorleken för en uppräknings, men alla talen i uppräkningsen genereras inte förrän på begäran. En `Int` tar 4 bytes i minnet. Ungefär hur mycket plats i minnet tar de objekt som variablerna (a) `intervall` respektive (b) `sekvns` refererar till nedan?

```
1 scala> val intervall = (1 to Int.MaxValue by 2)
2 scala> val sekvns = intervall.toArray
```

*Tips:* Använd uttrycket `BigInt(Int.MaxValue) * 2` i dina beräkningar.

- ★ **Uppgift 18.** *Undersök den genererade byte-koden.* Kompilatorn genererar byte-kod, uttalas "bajtkod" (eng. *byte code*), som den virtuella maskinen tolkar och översätter till maskinkod medan programmet kör.

Skapa en fil `plusxy.scala` med:

```
@main def plusxy(x: Int, y: Int) = x + y
```

Kompilera programmet med

```
scala-cli compile plusxy.scala
```

Navigera med `cd .scala-build/` och vidare ner med `ls` och `cd` så djupt du kan komma i katalogstrukturen tills du befinner dig i katalogen `main`. Notera vilka filer kompilatorn har skapat med `ls`. Med kommandot `javap -v 'plusxy$package$.class'` kan du undersöka byte-koden direkt i terminalen.

```
1 javap -v 'plusxy$package$.class'
```

a) Leta upp raden `public int plusxy(int, int);` och studera koden efter `Code:` och försök gissa vilken instruktion som utför själva additionen.

b) Vad händer om vi lägger till en parameter?

Skapa en ny fil `plusxyz.scala`:

```
@main def plusxyz(x: Int, y: Int, z: Int) = x + y + z
```

Kompilera och studera därefter byte-koden med `javap -v 'plusxyz$package$.class'`. Vad skiljer byte-koden mellan `plusxy` och `plusxyz`?

c) Läs om byte-kod här: [en.wikipedia.org/wiki/Java\\_bytecode](https://en.wikipedia.org/wiki/Java_bytecode). Vad betyder den inledande bokstaven i additionsinstruktionen?

## Kapitel 3

# Funktioner och abstraktion

Begrepp som ingår i denna veckas studier:

- ☐ abstraktion
- ☐ funktion
- ☐ parameter
- ☐ argument
- ☐ returtyp
- ☐ default-argument
- ☐ namngivna argument
- ☐ parameterlista
- ☐ funktionshuvud
- ☐ funktionskropp
- ☐ applicera funktion på alla element i en samling
- ☐ uppdelad parameterlista
- ☐ skapa egen kontrollstruktur
- ☐ funktionsvärde
- ☐ funktionstyp
- ☐ äkta funktion
- ☐ stegad funktion
- ☐ apply
- ☐ anonyma funktioner
- ☐ lambda
- ☐ predikat
- ☐ aktiveringspost
- ☐ anropsstacken
- ☐ objektheapen
- ☐ stack trace
- ☐ värdeandrop
- ☐ namnanrop
- ☐ klammerparentes och kolon vid ensam parameter
- ☐ rekursion
- ☐ scala.util.Random
- ☐ slumpvalsfrö

## 3.1 Teori

### 3.1.1 Vad är abstraktion?

- **Abstraktion** innebär att skapa en förenklad **modell** ur konkreta detaljer
- Vi "hittar på" nya **begrepp** som ger oss återanvändbara "byggblock" för våra tankar och vår kommunikation
- Vi får ett abstrakt **namn** som kan användas i stället för en massa **konkreta detaljer**
- Skilj på abstraktionens **namn** (begrepp, koncept), dess **användning** (anrop) och dess detaljerade **beskrivning** (definition, implementation)
- **Funktioner** (som du redan känner från matematiken) är en av våra **viktigaste** abstraktionsmekanismer

<https://sv.wikipedia.org/wiki/Abstraktion> <https://en.wikipedia.org/wiki/Abstraction>

---

### 3.1.2 Exempel på abstraktionsmekanismer inom datavetenskapen

Vi kommer att behandla flera olika, alltmer **kraftfulla** abstraktionsmekanismer i denna kurs:

- Funktioner
- Objekt
- Klasser
- Arv
- Generiska strukturer
- Kontextuella abstraktioner

Dessa abstraktionsmekanismer blir **extra kraftfulla** om de **kombineras**!

---

### 3.1.3 Funktion: deklaration och anrop

**def** funktionsnamn(parameterdeklarationer): returtyp = uttryck

- En funktion har ett **huvud** och efter = kommer dess **kropp**.
- En **namngiven** funktion **deklareras** med nyckelordet **def**
- En funktion kan ha **parametrar** som deklareras i huvudet.
- **Kroppen** ska vara ett **uttryck** (ev. ett block med flera uttryck).
- **Parametrar** binds till **argument** vid **anrop**.
- Uttrycket i funktionens kropp **evalueras** vid **varje anrop**.
- Värdet av uttrycket blir funktionen **returvärde**.

Exempel:

```
def öka(a: Int, b: Int): Int = a + b
```

```
scala> öka(42, 1)
val res0: Int = 43
```

---

### 3.1.4 Deklarera funktioner, överlagring

- En parameter, och sedan två parametrar:

```
1 scala> def öka(a: Int): Int = a + 1
2
3 scala> def öka(a: Int, b: Int): Int = a + b
4
5 scala> öka(1)
6 val res0: Int = 2
7
8 scala> öka(1,1)
9 val res1: Int = 2
```

- Båda funktionerna ovan kan finnas samtidigt! Trots att de har **samma namn** är de **olika funktioner**; kompilatorn kan skilja dem åt med hjälp av de **olika parameter-listorna**.
- Detta kallas **överlagring** (eng. *overloading*) av funktioner.
- Överlagring ger **flexibilitet i användningen**; vi slipper hitta på nytt namn så som öka2 vid 2 parametrar.

---

### 3.1.5 Funktioner med defaultargument

- Vi kan ofta åstadkomma samma flexibilitet som vid överlagring, men med **en enda** funktion, om vi i stället använder **defaultargument**:

```
scala> def inc(a: Int, b: Int = 1) = a + b

scala> inc(42, 2)
val res0: Int = 44

scala> inc(42, 1)
val res1: Int = 43

scala> inc(42)
val res2: Int = 43
```

- Om ett argument utelämnas och parametern deklarerats med defaultargument så appliceras detta. Kompilatorn fyller alltså i argumentet åt oss, om det är entydigt vilken parameter som avses.

---

### 3.1.6 Funktioner med namngivna argument

- Genom att använda **namngivna argument** behöver man inte hålla reda på ordningen på parametrarna, bara man känner till parameternamnen.
- Namngivna argument går fint att **kombinera** med defaultargument.

```
scala> def namn(förnamn: String,
               efternamn: String,
               förnamnFörst: Boolean = true,
               ledtext: String = "Namn:"): String =
```

```

    if förnamnFörst then s"$ledtext $förnamn $efternamn"
    else s"$ledtext $efternamn, $förnamn"

scala> namn(ledtext = "Name:", efternamn = "Coder", förnamn = "Kim")
val res0: String = Name: Kim Coder

```

### 3.1.7 Enhetlig access

- Om en funktion **deklarerar med** tom parameterlista () så ska den **anropas med** tom parameterlista.

```

scala> def tomParameterlista() = 42

scala> tomParameterlista()
val res1: Int = 42

scala> tomParameterlista
1 |tomParameterlista
  |^^^^^^^^^^^^^^^^^^
  |method tomParameterlista must be called with () argument

```

- En parameterlös funktion deklarerad **utan** () ska anropas **utan** () .

```

scala> def ingenParameterlista = 42
scala> ingenParameterlista()
1 |ingenParameterlista()
  |^^^^^^^^^^^^^^^^^^
  |method ingenParameterlista does not take parameters

```

- Deklaration utan () möjliggör **enhetlig access**: implementationen kan ändras från **val** till **def** eller tvärtom, **utan** att **användandet** påverkas.

### 3.1.8 Anropsstacken och objektheapen

Minnet som innehåller ett programs data är uppdelat i två delar:

- Anropsstacken:**
  - På anropsstacken läggs en **aktiveringspost** (eng. *stack frame*<sup>1</sup>, *activation record*) för varje funktionsanrop med plats för **parametrar** och **lokala variabler**.
  - Aktiveringsposten **raderas** när **returvärdet** har levererats.
  - Stacken **växer** vid **nästlade funktionsanrop**, då en funktion i sin tur anropar en annan funktion.
- Objektheapen:** I objektheapen<sup>2,3</sup> sparas alla objekt (data) som allokeras under körning. Heapen städas då och då av **skräpsamlaren** (eng. *garbage collector*), och minne som inte används längre frigörs.

<sup>1</sup>[en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)

<sup>2</sup>[en.wikipedia.org/wiki/Memory\\_management](https://en.wikipedia.org/wiki/Memory_management)

<sup>3</sup>Ej att förväxlas med datastrukturen heap [sv.wikipedia.org/wiki/Heap](https://sv.wikipedia.org/wiki/Heap)

### 3.1.9 Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
scala> def h(x: Int, y: Int) = { val z = x + y; println(z) }

scala> def g(a: Int, b: Int) = { val x = 1; h(x + 1, a + b) }

scala> def f() = { val n = 5; g(n, 2 * n) }

scala> f()
```

#### Stacken

variabel	värde	Aktiveringspost för anrop av...
n	5	f
a	5	g
b	10	
x	1	
x	2	h
y	15	
z	17	

### 3.1.10 Vad är en stack trace?

När du letar buggar vid körtidsfel har du nytta av att **noga studera utskriften av anropsstacken** (eng. *stack trace*):

```
1 // Program i filen bmi.scala
2
3 @main
4 def bmi(heightCm: Int, weightKg: Int) =
5   safeDiv(weightKg, heightCm * heightCm)
6
7 def safeDiv(numerator: Int, denominator: Int): (Int, String) =
8   if denominator == 0 then (numerator / denominator, "") // ser du buggen?
9   else (0, "division by zero")
```

```
1 > scala-cli run bmi.scala -- 0 42
2 Exception in thread "main" java.lang.ArithmeticException: / by zero
3 // HÄR KOMMER STACK TRACE pga körtidsfel - se nästa bild
```

### 3.1.11 Hur läsa en stack trace?

```
1 Exception in thread "main" java.lang.ArithmeticException: / by zero
2   at bmi$package$.safeDiv(bmi.scala:8)
3   at bmi$package$.bmi(bmi.scala:5)
4   at bmi.main(bmi.scala:3)
```

- En **stack trace** skrivs ut efter en krasch p.g.a. körtidsfel.
- Körtidsfel känns igen med ordet **Exception**.
- Först kommer en beskrivning av felet som orsakat kraschen, här:  
java.lang.ArithmeticException: \ by zero
- Därefter visas anropsstacken.
- För varje funktionsanrop anges: **klass.metod(kodfil:radnummer)**
- Main-funktioner läggs i ett singelobjekt i ett speciellt paket
- Singelobjekt i Scala kodas som en Java-klass med dollar-tecken efter namnet, eftersom det inte finns singelobjekt i JVM.

### 3.1.12 Lokala funktioner

Med lokala funktioner kan delproblem lösas med nästlade abstraktioner.

```
def gissaTalet(max: Int, min: Int = 1): Unit =
  def gissat = io.StdIn.readLine(s"Gissa talet mellan $min och $max: ").toInt

  val hemlis = (math.random() * (max - min) + min).toInt

  def skrivLedtrådOmEjRätt(gissning: Int): Unit =
    if gissning > hemlis then println(s"$gissning är för stort :(")
    else if gissning < hemlis then println(s"$gissning är för litet :(")

  def inteRätt(gissning: Int): Boolean =
    skrivLedtrådOmEjRätt(gissning)
    gissning != hemlis

  def loop: Int = { var i = 1; while inteRätt(gissat) do i += 1; i }

  println(s"Du hittade talet $hemlis på $loop gissningar :)")
```

Lokala, nästlade funktionsdeklarationer är tyvärr inte tillåtna i många andra språk, t.ex. Java.<sup>4</sup>

### 3.1.13 Funktioner är äkta värden i Scala

- En funktion är ett **äkta värde**.
- Vi kan till exempel tilldela en variabel ett **funktionsvärde**.
- Med hjälp enbart funktionsnamnet får vi funktionen som ett **värde** (inga argument appliceras än):

```
scala> def add(a: Int, b: Int) = a + b

scala> val f = add
val f: (Int, Int) => Int = Lambda$7210/0x00000000841e4e040@1ce2db23

scala> f(21, 21)
val res0: Int = 42
```

<sup>4</sup>[stackoverflow.com/questions/5388584/does-java-support-inner-local-sub-methods](https://stackoverflow.com/questions/5388584/does-java-support-inner-local-sub-methods)

- Ett funktionsvärde har en **typ** precis som alla värden:  
`f: (Int, Int) => Int`
- Ett funktionsvärde har till skillnad från en funktionsdeklaration inget namn (variabeln `f` har ett namn men inte själva funktionen). Den kallas därför en **anonym** funktion eller **lambda** (mer om detta snart).

---

### 3.1.14 Funktionsvärden kan vara argument

En funktion kan ha en annan funktion som parameter:

```
1 scala> def tvåGångar(x: Int, f: Int => Int) = f(f(x))
2
3 scala> def öka(x: Int) = x + 1
4
5 scala> def minska(x: Int) = x - 1
6
7 scala> tvåGångar(42, öka)
8 val res1: Int = 44
9
10 scala> tvåGångar(42, minska)
11 val res1: Int = 40
```

---

### 3.1.15 Applicera funktioner på element i samlingar med `map`

```
def öka(x: Int) = x + 1

def minska(x: Int) = x - 1

val xs = Vector(1, 2, 3)
```

Metoden **map** fungerar på alla Scala-samlingar och tar **en funktion som argument** och applicerar denna funktion på alla element och **skapar en ny samling** med resultaten:

```
1 scala> xs.map(öka)
2 val res0: ??? // vad blir resultatet?
3
4 scala> xs.map(minska)
5 val res1: ??? // vad blir resultatet?
```

En funktion som har funktionsvärden som indata (eller utdata) kallas en **högre ordningens funktion** (eng. *higher-order function*).

---

### 3.1.16 Applicera funktioner på element i samlingar med `map`

```
def öka(x: Int) = x + 1

def minska(x: Int) = x - 1

val xs = Vector(1, 2, 3)
```



Metoden **map** fungerar på alla Scala-samlingar och tar **en funktion som argument** och applicerar denna funktion på alla element och **skapar en ny samling** med resultaten:

```
1 scala> xs.map(öka)
2 val res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
3
4 scala> xs.map(minska)
5 val res1: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
```

En funktion som har funktionsvärden som indata kallas en **högre ordningens funktion** (eng. *higher-order function*).

---

### 3.1.17 Äkta funktioner

- En **äkt**a (eng. *pure*) funktion är en funktion som ger ett resultat som **enbart** beror av dess argument. Alltså som funktioner i matematiken.
- En äkta (matematisk) funktion är **referentiellt transparent** (eng. *referentially transparent*), vilket innebär att varje anrop kan bytas ut mot funktionskroppen där parameterna ersatts med motsvarande argument.
- En äkta funktion har **inga sidoeffekter**, t.ex. utskrift, skriva/läsa filer, eller uppdateringar av variabler **synliga utanför** funktionen.
- Exempel:

```
def add(x: Int, y: Int): Int = x + y           // äkta funktion
def rnd(n: Int): Int = (math.random() * n).toInt // oäkta funktion
```

- Uttrycket `add(41, 1)` kan ersättas med `41 + 1` som i sin tur kan ersättas med `42` utan att det påverkar resultatet. Resultatet av `add(41, 1)` blir **samma varje gång** funktionen appliceras med dessa argument
  - Uttrycket `rnd(42)` kan **inte** bytas ut mot ett specifikt uttryck som säkert ger samma resultat varje gång. Alltså: *ej referentiell transparens*.
- 

### 3.1.18 Exempel på oäkta funktioner: slump

- Funktioner vars värden på något sätt beror av slumpen är **inte** äkta funktioner.
  - Även om samma argument ges vid upprepad applicering, så kan ju resultatet bli olika.
  - Studera dokumentationen för `scala.util.Random` här: <https://www.scala-lang.org/api/current/scala/util/Random.html>
  - Du har nytta av funktionen `Random.nextInt` och slumpfrasfrö (eng. *random seed*) i veckans uppgifter.
- 

### 3.1.19 Slumpralsfrö: få samma slump tal varje gång

- Om man använder slump tal kan det vara svårt att leta buggar, eftersom det blir **olika varje gång** man kör programmet och buggen kanske bara uppstår ibland.
- Med klassen `scala.util.Random` kan man skapa **pseudo**-slumpralssekvenser.

- Om man ger ett s.k. **frö** (eng. *seed*), av heltalstyp, som argument till konstruktorn när man skapar en instans av klassen `scala.util.Random`, får man samma ”slumpmässiga” sekvens **varje gång** man kör programmet.

```
val seed = 42
val rnd = util.Random(seed) // skapa ny slumpgenerator med frö 42
val r = rnd.nextInt(6) // ger slumptal mellan 0 till och med 5
```

- Om man **inte** ger ett **frö** så sätts fröet till ”*a value very likely to be distinct from any other invocation of this constructor*”. Då vet vi inte vilket fröet blir och det blir olika varje gång man kör programmet.

```
val rnd = util.Random() // OLIKA frö varje körning
val r = rnd.nextInt(6) // ger slumptal mellan 0 till och med 5
```

### 3.1.20 Anonyma funktioner

- Man behöver inte ge funktioner namn. De kan i stället skapas med hjälp av **funktionslitteraler**.<sup>5</sup>
- En funktionslitteral har ...
  1. en parameterlista (utan funktionsnamn, utan returtyp),
  2. sedan den reserverade teckenkombinationen `=>`
  3. och sedan ett uttryck (eller ett block).
- Exempel:

```
(x: Int, y: Int) => x + y // vilken typ?
```

- Om kompilatorn kan gissa typerna från sammanhanget så behöver typerna inte anges i själva funktionslitteralen:

```
val f: (Int, Int) => Int = (x, y) => x + y
```

### 3.1.21 Applicera anonyma funktioner på element i samlingar

Anonym funktion skapad med funktionslitteral direkt i anropet:

```
1 scala> val xs = Vector(1, 2, 3)
2
3 scala> xs.map((x: Int) => x + 1)
4 res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

Eftersom kompilatorn här kan härleda typerna så behövs de inte:

```
1 scala> xs.map(x => x + 1)
2 res1: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

<sup>5</sup>Även kallat ”lambda-värde” eller bara ”lambda” efter den s.k. lambdakalkylen. [en.wikipedia.org/wiki/Anonymous\\_function](https://en.wikipedia.org/wiki/Anonymous_function)

Om man bara använder parametern en enda gång i funktionen så kan man byta ut parameternamnet mot ett understreck.

```
1 scala> xs.map(_ + 1)
2 res2: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

### 3.1.22 Platshållarsyntax för anonyma funktioner

Understreck i funktionslitteraler kallas **platshållare** (eng. *placeholder*) och medger ett förkortat skrivsätt **om** den parameter som understrecket representerar används **endast en gång**.

```
_ + 1
```

Ovan expanderar av kompilatorn till följande funktionslitteral (där namnet på parametern är godtyckligt):

```
x => x + 1
```

Det kan förekomma flera understreck; det första avser första parametern, det andra avser andra parametern etc.

```
_ + _
```

... expanderas till:

```
(x, y) => x + y
```

### 3.1.23 Exempel på platshållarsyntax med reduceLeft

Metoden `reduceLeft` applicerar en funktion på de två första elementen i en sekvens och tar sedan resultatet som första argument och nästa element som andra argument och upprepar detta genom hela samlingen.

```
1 scala> def summa(x: Int, y: Int) = x + y
2
3 scala> val xs = Vector(1, 2, 3, 4, 5)
4
5 scala> xs.reduceLeft(summa)
6 res20: Int = 15
7
8 scala> xs.reduceLeft((x, y) => x + y)
9 res21: Int = 15
10
11 scala> xs.reduceLeft(_ + _)
12 res22: Int = 15
13
14 scala> xs.reduceLeft(_ * _)
15 res23: Int = 120
```

### 3.1.24 Predikat, med och utan namn

- En funktion som har Boolean som returtyp kallas för ett **predikat**.
- Exempel:

```
def isTooLong(name: String): Boolean = name.length > 10

def isTall(heightInMeters: Double, limit: Double = 1.78): Boolean =
  heightInMeters > limit
```

- Predikat ges ofta ett namn som börjar på is eller has så att man lätt kan se att det är ett predikat när man läser kod som anropar funktionen.
- Många av samlingsmetoderna i Scalas standardbibliotek tar predikat som funktionsargument. Exempel med predikat som anonym funktion:

```
scala> val parts = Vector(3, 1, 0, 5).partition(_ > 1)
val parts: (Vector[Int], Vector[Int]) =
  (Vector(3, 5), Vector(1, 0))
```

- Studera snabbreferensen och försök hitta samlingsmetoder som tar predikat som funktionsargument. <http://cs.lth.se/pgk/quickref>  
I anropsexempel med predikat-argument används bokstaven p.

### 3.1.25 Funktionsvärde vid tom parameterlista: använd "thunk"

- Om du vill ha funktionen som ett värde så skriv bara namnet och inte parameterlistan (samma exempel som tidigare):

```
scala> def add(a: Int, b: Int) = a + b

scala> val f = add      // inget anrop sker
val f: (Int, Int) => Int = Lambda7210/0x0000000841e4e040@1ce2db23
```

- Vid **tom parameterlista** behövs anonym funktion som **fördröjer anrop**:

```
scala> def a() = 42
def a(): Int

scala> val b = a
1 |val b = a
  |      ^
  |      method a must be called with () argument

scala> val b = () => a() // anonym funktion, fördröjd evaluering
val b: () => Int = Lambda7214/0x0000000841e50440@565d794
```

- Notera typen: `() => Int` Ett sådant funktionsvärde kallas **thunk**  
<https://en.wikipedia.org/wiki/Thunk>

### 3.1.26 Hur fungerar egentligen upprepa i Kojo?

```
upprepa(10) {
    println("hej")
}
```

Vi ska nu se hur vi, genom att kombinera ett antal koncept, kan skapa egna kontroll-strukturer likt upprepa ovan:

- klammerparentes vid ensam parameter
- multipla parameterlistor
- namnanrop (födröjd evaluering)

### 3.1.27 Multipla parameterlistor

Vi har tidigare sett att man kan ha mer än en parameter:

```
scala> def add(a: Int, b: Int) = a + b

scala> add(21, 21)
res0: Int = 42
```

Man kan även ha **mer än en parameterlista**:

```
scala> def add(a: Int)(b: Int) = a + b

scala> add(21)(21)
res1: Int = 42
```

(eng. *multiple parameter lists*)

[docs.scala-lang.org/style/declarations.html#multiple-parameter-lists](https://docs.scala-lang.org/style/declarations.html#multiple-parameter-lists)

### 3.1.28 Värdeanrop och namnanrop

Det vi sett hittills är **värdeanrop**: argumentet evalueras **först** innan dess **värde** *sedan* appliceras:

[illegible]

Men man kan med => före parametertypen åstadkomma **namnanrop**: argumentet **"klistras in"** i stället för **namnet** och evalueras **varje gång** (kallas även **fördröjd evaluering**):

[illegible]

**Kluring:** Varför skrivs "hej" ut en extra gång i början? ledtråd: 1 to n

### 3.1.29 Klammerparenteser vid ensam parameter

Så här har vi sett nyss att man man göra:

```
1 scala> def twice(action: => Unit): Unit = { action; action }
2
3 scala> twice( { print("hej"); print("san ") } )
4 hejsan hejsan
```

Det ser rätt klyddigt ut med { ( och ) } eller vad tycker du? Men... För alla funktioner  $f$  gäller att:

det är helt ok att byta ut vanliga parenteser:

$f(\text{uttryck})$

mot krullparenteser:

$f\{\text{uttryck}\}$

**om** parameterlistan har **exakt en** parameter.

Man kan alltså skippa det yttre parentesparet för bättre läsbarhet:

```
scala> twice { print("hej"); print("san ") }
```

### 3.1.30 Skapa din egen kontrollstruktur

- Genom att **kombinera multipla parameterlistor** med **namnanrop** med **klammerparentes vid ensam parameter** kan vi skapa vår egen kontrollstruktur: upprepa

```
upprepa(42){
  if math.random() < 0.5 then print(" gurka")
  else print(" tomat")
}
```

Hur då? Till exempel så här:

```
def upprepa(n: Int)(block: => Unit) = for i <- 0 until n do block
```

```
gurka gurka gurka tomat tomat gurka gurka gurka tomat tomat tomat tomat
```

### 3.1.31 Kolon vid ensam parameter

Du kan från Scala 3.3 i stället för klammerparentes vid ensam parameter använda kolon för att få färre ”krullisar” (eng. *fewer braces*).

```
upprepa(42):
  if math.random() < 0.5
  then print(" gurka")
  else print(" tomat")
```

Denna förenklade syntax föregicks av långa diskussioner innan den till slut accepterades.<sup>6</sup>

<sup>6</sup>Den nyfikne kan läsa förslaget före omröstning här:  
<https://docs.scala-lang.org/sips/fewer-braces.html>

### 3.1.32 Stegade funktioner, "Curry-funktioner"

Om en funktion har multipla parameterlistor kan man skapa **stegade funktioner**, även kallat **partiellt applicerade** funktioner (eng. *partially applied functions*) eller **"Curry"-funktioner**.

```
scala> def add(x: Int)(y: Int) = x + y

scala> val öka = add(1)
val öka: Int => Int = Lambda7339/0x00000000841eb7040@19c8add7

scala> Vector(1,2,3).map(öka)
val res0: Vector[Int] = Vector(2, 3, 4)

scala> Vector(1,2,3).map(add(2))
val res1: Vector[Int] = Vector(3, 4, 5)
```

### 3.1.33 Funktion med fångad variabelrymd: *closure*

```
def f(x: Int): Int => Int =
  val a = 42 + x
  def g(y: Int): Int = y + a
  g
```

Funktionen g **fångar** den lokala variabeln a i ett **funktionsobjekt**.

```
scala> val funkis = f(1)
val funkis: Int => Int = Lambda7356/0x00000000841ed2840@1bda26bc

scala> funkis(2)
val res0: Int = 45
```

Ett funktionsobjekt med "fångade" variabler kallas **closure**.  
(Mer om funktioner som objekt senare.)

### 3.1.34 Rekursiva funktioner

- Funktioner som **anropar sig själv** kallas **rekursiva**.

```
scala> def fakultet(n: Int): Int =
  if n < 2 then 1 else n * fakultet(n - 1)

scala> fakultet(5)
val res0: Int = 120
```

- För varje nytt anrop läggs en ny aktiveringspost på stacken.
- I aktiveringsposten sparas varje returvärde som gör att  $5 * (4 * (3 * (2 * 1)))$  kan beräknas.
- Rekursionen avbryts när man når **basfallet**, här  $n < 2$
- En rekursiv funktion **måste** ha en returtyp.

### 3.1.35 Loopa med rekursion

```
def gissaTalet(max: Int, min: Int = 1): Unit =
  def gissat =
    io.StdIn.readLine(s"Gissa talet mellan [$min, $max]: ").toInt

  val hemlis = (math.random() * (max - min) + min).toInt

  def skrivLedtrådOmEjRätt(gissning: Int): Unit =
    if gissning > hemlis then println(s"$gissning är för stort :(")
    else if (gissning < hemlis) println(s"$gissning är för litet :(")

  def ärRätt(gissning: Int): Boolean =
    skrivLedtrådOmEjRätt(gissning)
    gissning == hemlis

  def loop(n: Int = 1): Int = if ärRätt(gissat) then n else loop(n + 1)

  println(s"Du hittade talet $hemlis på ${loop()} gissningar :)")
```

### 3.1.36 Rekursiva datastrukturer

- Datastrukturena Lista och Träd är exempel på datastrukturer som passar bra ihop med rekursion.
- Båda dessa datastrukturer kan beskrivas rekursivt:
  - En lista består av ett huvud och en lista, som i sin tur består av ett huvud och en lista, som i sin tur...
  - Ett träd består av grenar till träd som i sin tur består av grenar till träd som i sin tur, ...
- Dessa datastrukturer bearbetas med fördel med rekursiva algoritmer.
- I denna kursen ingår rekursion endast "för kännedom":  
du ska veta vad det är och kunna skapa en enkel rekursiv funktion, t.ex. fakultetsberäkning. Du kommer jobba mer med rekursion och rekursiva datastrukturer i fortsättningskursen.

### 3.1.37 Kompilera om det som ändrats vid varje sparning

- Den kreativa programmeringsprocessen innehåller många korta cykler av koda, ändra, testa.
- Det blir **många omkompileringar** och då vill man gärna slippa skriva samma kommando om och om igen.
- Vid **varje liten ändring** vill man **kompilera om** det som ändrats och se om det fortfarande kompilerar utan fel.



- Då kan du använda:  
`scala-cli compile . --watch`  
Ändringar bevakas och kompileras om direkt.
-

## 3.2 Övning functions

### Mål

- ☐ Kunna skapa och använda funktioner med en eller flera parametrar, default-argument, och namngivna argument.
- ☐ Kunna förklara nästlade funktionsanrop med aktiveringsposter på stacken.
- ☐ Kunna förklara skillnaden mellan äkta och "oäkta" funktioner.
- ☐ Kunna applicera en funktion på alla element i en samling.
- ☐ Kunna använda funktioner som äkta värden.
- ☐ Kunna skapa och använda anonyma funktioner (ä.k. lambda-funktioner).
- ☐ Känna till att funktioner kan ha uppdelad parameterlista.
- ☐ Känna till att det går att partiellt applicera argument på funktioner med uppdelad parameterlista för att skapa s.k. stegade funktioner (ä.k. curry-funktioner).
- ☐ Känna till rekursion och kunna beskriva vad som kännetecknar en rekursiv funktion.
- ☐ Känna till att det går att skapa egna kontrollstrukturer med hjälp av namnanrop.
- ☐ Känna till skillnaden mellan värdeanrop och namnanrop.
- ☐ Kunna tolka en stack trace.

### Förberedelser

- ☐ Studera begreppen i kapitel 3

### 3.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** *Para ihop begrepp med beskrivning.* Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

funktionshuvud	1	A	gör att argument kan utelämnas
funktionskropp	2	B	funktion utan namn; kallas även lambda
parameterlista	3	C	ger alltid samma resultat om samma argument
block	4	D	har parameterlista och eventuellt en returtyp
namngivna argument	5	E	fördröjd evaluering av argument
defaultargument	6	F	ger återupprepningsbar sekvens av pseudoslumtals
värdeanrop	7	G	koden som exekveras vid funktionsanrop
namnanrop	8	H	lista anropskedja vid körtidsfel
äkta funktion	9	I	beskriver namn och typ på parametrar
predikat	10	J	en funktion som ger ett booleskt värde
slumtalsfrö	11	K	en funktion som anropar sig själv
anonym funktion	12	L	argumentet evalueras innan anrop
rekursiv funktion	13	M	kan ha lokala namn; sista raden ger värdet
stack trace	14	N	gör att argument kan ges i valfri ordning

**Uppgift 2.** *Definiera och anropa funktioner.* En funktion med en parameter definieras med följande syntax i Scala:

**def** *namn*(parameter: Typ = defaultArgument): Returtyp = returvärde

- Definiera funktionen öka som har en heltalsparameter x och vars returvärde är argumentet plus 1. Defaultargument ska vara 1. Ange returtypen explicit.
- Vad har uttrycket öka(öka(öka(öka())))) för värde?
- Definiera funktionen minska som har en heltalsparameter x och vars returvärde är argumentet minus 1. Defaultargument ska vara 1. Ange returtypen explicit.
- Vad är värdet av uttrycket öka(minska(öka(öka(minska(minska())))))
- Vad är det för skillnad mellan parameter och argument?

**Uppgift 3.** *Textspelet AliensOnEarth.* Ladda ner spelet nedan <sup>7</sup> och studera koden.

```

1 object AliensOnEarth:
2   def readChoice(msg: String, options: Vector[String]): String =
3     options.indices.foreach(i => println(s"$i: ${options(i)}"))
4     val selected = scala.io.StdIn.readLine(msg).toInt
5     options(selected)
6
7   def isAnswerYes(msg: String): Boolean =
8     scala.io.StdIn.readLine(s"$msg (Y/n)").toLowerCase.startsWith("y")
9
10  def randomChoice(options: Vector[String]): String =
11    val selected = scala.util.Random.nextInt(options.size)
12    options(selected)
13
14  def playGame(alien: String, maxPoints: Int = 1000): Int =
15    val os = Vector("penguin", "window", "apple")
16    val correct = if math.random() < 0.5 then os(0) else randomChoice(os)
17    val cheatCode = (os.indexOf(correct) + 1) * math.Pi
18    println(s""|Hello $alien!
19              |You are an alien on Earth.
20              |Your encrypted password is $cheatCode.
21              |You see three strange Earth objects.""|.stripMargin)
22    val choice = readChoice(s"$alien wants? ", os)
23    if choice == correct then maxPoints else 0
24
25  def main(args: Array[String]): Unit =
26    try
27      val name = if args.size > 0 then args(0) else "Captain Zoom"
28      val points = playGame(alien = name)
29      if points > 0 then println(s"Congratulations $name! :)")
30      println(s"You got $points points.")
31    catch case e: Exception =>
32      println(s"Game over. The Earth was hit by an asteroid. :(")
33      if isAnswerYes("Do you want to trace the asteroid?") then
34        e.printStackTrace()

```

- Medan du läser koden, försök lista ut vilket som är bästa strategin för att få så mycket poäng som möjligt. Kompilera och kör spelet i terminalen med ditt favoritnamn som argument. Vilket av de tre objekten på planeten jorden har störst sannolikhet att vara bästa alternativet?

- Para ihop kodsuttarna nedan med bästa beskrivningen.<sup>8</sup>

<sup>7</sup><https://raw.githubusercontent.com/lunduniversity/introprog/master/compendium/examples/AliensOnEarth.scala>

<sup>8</sup>Gör så gott du kan även om allt inte är solklart. Vissa saker kommer vi att gå igenom i detalj först under senare kursmoduler.

<code>options.indices</code>	1	A	gör om en sträng till små bokstäver
<code>"1X2".toLowerCase</code>	2	B	heltalssekvens med alla index i en sekvens
<code>Random.nextInt(n)</code>	3	C	slumptal i intervallet 0 until n
<code>try { } catch { }</code>	4	D	tar bort marginal till och med vertikalstreck
<code>""" ... """</code>	5	E	fångar undantag för att förhindra krasch
<code>s.stripMargin</code>	6	F	sträng som kan sträcka sig över flera kodrader
<code>e.printStackTrace</code>	7	G	skriver ut information om ett undantag

*Tips:* Med hjälp av REPL kan du ta reda på hur olika delar fungerar, t.ex.:

```

1 scala> val os = Vector("p", "w", "a")
2 scala> os.indices
3 scala> os.indices.foreach(i => println(i))
4 scala> os.indexOf("w")
5 scala> os.indexOf("gurka")
6 scala> Vector("hej", "hejsan", "hej").indexOf("hej")
7 scala> try 1 / 0 catch case e: Exception => println(e)

```

Kolla även dokumentationen för `nextInt`, `readLine`, `m.fl` genom att söka här:

<http://www.scala-lang.org/api/current/index.html>

*Tips inför fortsättningen:*

- När jag hittade på `AliensOnEarth` började jag med ett mycket litet program med en enkel `main`-funktion som bara skrev ut något kul. Sedan byggde jag vidare på programmet steg för steg och kompilerade och testade efter varje liten ändring.
- När jag kodar har jag REPL igång i ett eget terminalfönster och api-dokumentationen för Scala i en webbläsare redo för sökningar. Jag återanvänder också användbara snuttar från kod jag gjort tidigare och inspireras ofta av lösningar från <https://stackoverflow.com> (om jag kan begripa dem och de verkar rimliga).
- Detta arbetssätt tar ett tag att komma in i, men är ett bra sätt att uppfinna allt större och bättre program. Ett stort program byggs lättast i små steg och felsökning blir mycket lättare om man bara gör små tillägg åt gången.
- Du får också det mycket lättare att förstå ditt program om du delar upp koden i många korta funktioner med bra namn. Du kan sedan lättare hitta på mer avancerade funktioner genom att återanvända befintliga.
- Under veckans laboration ska du utveckla ditt eget textspel. Då har du nytta av att återanvända funktionerna för indata och slumpdragning från `AliensOnEarth`.

**Uppgift 4. Äkta funktioner.** En äkta funktion<sup>9</sup> (eng. *pure function*) ger alltid samma resultat med samma argument (så som vi är vana vid inom matematiken) och har inga externt observerbara sidoeffekter (till exempel utskrifter).

Vilka funktioner i objektet `inSearchOfPurity` nedan är äkta funktioner?

<sup>9</sup>Äkta funktioner uppfyller per definition *referentiell transparens* (eng. *referential transparency*) som du kan läsa mer om här: [en.wikipedia.org/wiki/Referential\\_transparency](http://en.wikipedia.org/wiki/Referential_transparency)

```
object inSearchOfPurity:
  var x = 0
  val y = x
  def inc(i: Int): Int = i + 1
  def oink(i: Int): String = { x = x + i; "Pig says " + ("oink " * x) }
  def addX(i: Int): Int = x + i
  def addY(i: Int): Int = y + i
  def isPalindrome(s: String): Boolean = s == s.reverse
  def rnd(min: Int, max: Int): Double = math.random() * max + min
```

*Tips:* Klistra in hela singelobjektet i REPL och testa att anropa funktionerna om du är osäker på vad som händer. Om du gör **import** inSearchOfPurity.\* kommer du åt namnen i singelobjektet direkt och kan lätt undersöka variablernas värden.

**Uppgift 5.** *Applicera funktion på varje element i en samling. Funktion som argument.* Deklarera funktionen öka och variabeln xs enligt nedan i REPL:

```
1 scala> def öka(x: Int) = x + 1
2 scala> val xs = Vector(3, 4, 5)
```

Para ihop nedan uttryck till vänster med det uttryck till höger som har samma värde. Om du undrar något, testa uttrycken och olika varianter av dem i REPL.

<b>for</b> i <- 1 to 3 <b>yield</b> öka(i)	1	A	xs
Vector(2, 3, 4).map(i => öka(i))	2	B	Vector(4, 5, 6)
xs.map(öka)	3	C	()
xs.map(öka).map(öka)	4	D	Vector(5, 6, 7)
xs.foreach(öka)	5	E	Vector(2, 3, 4)

**Uppgift 6.** *Funktion som äkta värde.* Funktioner är äkta värden i Scala<sup>10</sup>. Det betyder att variabler kan ha funktioner som värden och funktionsvärden kan vara argument till funktioner som har funktionsparametrar<sup>11</sup>.

En funktion som har en heltalsparameter och ett heltalsresultat är av funktionstypen `Int => Int` (uttalas *int-till-int*) och värdet av funktionen utgör ett objekt som har en metod som heter `apply` med motsvarande funktionstyp.

a) Deklarera nedan funktioner och variabler i REPL. Para sedan ihop nedan uttryck till vänster med det uttryck till höger som skapar samma utskrift. Om du undrar något, testa uttrycken och olika varianter av dem i REPL.

```
1 scala> def hälsa(): Unit = println("Hej!")
2 scala> def fleraAnrop(antal: Int, f: () => Unit): Unit =
3     for _ <- 1 to antal do f()
4 scala> val f1 = () => hälsa()
5 scala> var f2 = (s: String) => println(s)
6 scala> val f3 = () => f2("Thunk")
```

<sup>10</sup>I likhet med t.ex. Javascript, men till skillnad från t.ex. Java.

<sup>11</sup>Funktioner som tar funktioner som argument kallas *högre ordningens funktioner*

<code>fleraAnrop(1, hälsa)</code>	1	A	<code>f2("Hej!\nHej!")</code>
<code>fleraAnrop(3, hälsa)</code>	2	B	<code>fleraAnrop(3, f1)</code>
<code>fleraAnrop(2, f1)</code>	3	C	<code>f3()</code>
<code>fleraAnrop(1, f3)</code>	4	D	<code>f2("Hej!")</code>

- b) Vilka typer har variablerna `f1`, `f2` och `f3`?
- c) Går det bra att skriva `f2 = f1`?
- d) Går det bra att skriva **val** `f4 = fleraAnrop`?
- e) Går det bra att skriva **val** `f4 = hälsa`?
- f) Går det bra att skriva **val** `f4: () => Unit = hälsa`?

**Uppgift 7. Anonyma funktioner.** Vi har flera gånger sett syntaxen `i => i + 1`, till exempel i en loop `(1 to 10).map(i => i + 1)` där funktionen `i => i + 1` appliceras på alla heltal från 1 till och med 10 och resultatet blir en ny sekvenssamling.

Syntaxen `(i: Int) => i + 1` är en litteral för att skapa ett funktionsvärde. Syntaxen liknar den för funktionsdeklarationer, men nyckelordet **def** saknas i funktionshuvudet och i stället för likhetstecken används `=>` för att avskilja parameterlistan från funktionskroppen. Om kompilatorn kan härleda typen ur sammanhanget kan kortformen `i => i + 1` användas.

Det finns ett *ännu* kortare sätt att skriva en anonym funktion *om* typen kan härledas *och* den bara använder sin parameter *en enda gång*; då går funktionslitteraler att skriva med s.k. *platshållarsyntax* som använder understreck, till exempel `_ + 1` och som automatiskt expanderar av kompilatorn till `ngtnamn => ngtnamn + 1` (namnet på parametern spelar ingen roll; kompilatorn väljer något eget, internt namn).

Para ihop uttryck till vänster med uttryck till höger som har samma värde:

<code>(0 to 2).map(i =&gt; i + 1)</code>	1	A	<code>Vector(9.0, 16.0, 25.0)</code>
<code>(1 to 3).map(_ + 1)</code>	2	B	<code>(2 to 4).map(i =&gt; i - 1)</code>
<code>(2 to 4).map(math.pow(2, _))</code>	3	C	<code>Vector(2.0, 2.5, 3.0)</code>
<code>(3 to 5).map(math.pow(_, 2))</code>	4	D	<code>Vector(2, 3, 4)</code>
<code>(4 to 6).map(_.toDouble).map(_ / 2)</code>	5	E	<code>Vector(4.0, 8.0, 16.0)</code>

Funktionslitteraler kallas även *anonyma funktioner*<sup>12</sup>, eftersom de inte har något namn, till skillnad från t.ex. **def** `öka(i: Int): Int = i + 1`, som ju heter `öka`.

**Uppgift 8. Lär dig läsa en stack trace.** Skriv ett program i filen `fel.scala` som orsakar ett *körtidsfel* och kör igång det i terminalen med `scala-cli run fel.scala`. Studera den stack trace som skrivs ut. Vad innehåller en stack trace? Diskutera med handledare hur du kan ha nytta av en stack trace när du felsöker.

### 3.2.2 Extrauppgifter; träna mer

<sup>12</sup>Ett annat vanligt namn är *lambda* efter det datalogiska matematikverktyget *lambdakalkyl*: <https://sv.wikipedia.org/wiki/Lambdakalkyl>

**Uppgift 9.** *Funktion med flera parametrar.* Definiera i REPL två funktioner `sum` och `diff` med två heltalsparametrar som returnerar summan respektive differensen av argumenten:

```
def sum(x: Int, y: Int): Int = x + y
def diff(x: Int, y: Int): Int = x - y
```

Vad har nedan uttryck för värden? Förklara vad som händer.

- a) `diff(0, 100)`
- b) `diff(100, sum(42, 43))`
- c) `sum(sum(42, 43), diff(100, sum(0, 0)))`
- d) `sum(diff(Byte.MaxValue, Byte.MinValue), 1)`

**Uppgift 10.** *Medelvärde.* Skriv och testa en funktion `avg` som räknar ut medelvärdet mellan två heltal och returnerar en `Double`.

**Uppgift 11.** *Funktionsanrop med namngivna argument.*

```
1 scala> def skrivNamn(efternamn: String, förnamn: String) =
2     println(s"Namn: $efternamn, $förnamn")
3 scala> skrivNamn(förnamn = "Stina", efternamn = "Triangelsson")
4 scala> skrivNamn(efternamn = "Oval", "Viktor")
```

- a) Vad skrivs ut efter rad 3 resp. rad 4 ovan?
- b) Nämn tre fördelar med namngivna argument.

**Uppgift 12.** *Bortkastade resultatvärden och returtypen Unit.* Undersök nedan kod i REPL och förklara vad som händer.

a)

```
1 scala> def tom = println("")
2 scala> println(tom)
```

b)

```
1 scala> def bortkastad: Unit = 1 + 1
2 scala> println(bortkastad)
```

c)

```
1 scala> def bortkastad2 = { val x = 1 + 1 }
2 scala> println(bortkastad2)
```

- d) Varför är det bra att explicit ange `Unit` som returtyp för procedurer?

### 3.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 13.** *Föränderlighet av parametrar.* Vad tror du om detta: Är en parameter förändringsbar i funktionskroppen ...

- a) ... i Scala? (Ja/Nej)
- b) ... i Java? (Ja/Nej)
- c) ... i Python? (Ja/Nej)

**Uppgift 14.** *Värdeanrop och namnanrop.* Normalt sker i Scala (och i Java) s.k. *värdeanrop* vid anrop av funktioner, vilket innebär att argumentuttrycket evalueras *före* bindningen till parameternamnet sker.

Man kan också i Scala (men inte i Java) med syntaxen `=>` framför parametertypen deklarerar att *namnanrop* ska ske, vilket innebär att evalueringen av argumentuttrycket *fördröjs* och sker *varje gång* namnet används i metodkroppen.

Deklarera nedan funktioner i REPL.

```
def snark: Int = { print("snark "); Thread.sleep(1000); 42 }
def callByValue(x: Int): Int = x + x
def callByName(x: => Int): Int = x + x
lazy val zzz = snark
```

Förklara vad som händer när nedan uttryck evalueras.

- a) `snark + snark`
- b) `callByValue(snark)`
- c) `callByName(snark)`
- d) `callByName(zzz)`

**Uppgift 15.** *Skapa din egen kontrollstruktur med hjälp av namnanrop.*

- a) Deklarera denna procedur i REPL:

```
def görDettaTvåGånger(b: => Unit): Unit = { b; b }
```

- b) Anropa `görDettaTvåGånger` med ett block som parameter. Blocket ska innehålla en utskriftssats. Förklara vad som händer.
- c) Använd *namnanrop* i kombination med en uppdelad parameterlista och skapa din egen kontrollstruktur enligt nedan.<sup>13</sup>

```
def upprepa(n: Int)(block: => Unit): Unit =
  var i = 0
  while i < n do
    ???
```

- d) Testa din kontrollstruktur i REPL. Låt upprepa 100 gånger att ett slumptal mellan 1 och 6 dras och sedan skrivs ut. Prova även att använda färre klammerparenteser med hjälp av kolon.

<sup>13</sup>Det är så loopens upprepa i Kojo är definierad.



e) Fördelen med upprepa är att den är koncis och lättanvänd. Men den är inte lika lätt att använda om man behöver tillgång till en loopvariabel. Implementera därför nedan kontrollstruktur.

```
def repeat(n: Int)(p: Int => Unit): Unit =
  var i = 0
  while i < n do
    ???
```

f) Använd repeat för att 100 gånger skriva ut loopvariabeln och ett slumpdecimaltal mellan 0 och 1.

**Uppgift 16.** *Uppdelad parameterlista och stegade funktioner.* Man kan dela upp parameterlistorna till en funktion i flera parameterlistor. Funktionen add1 nedan har en parameterlista med två parametrar medan add2 har två parameterlistor med en parameter vardera:

```
def add1(a: Int, b: Int) = a + b
def add2(a: Int)(b: Int) = a + b
```

a) När man anropar funktionen add2 ska argumenten skrivas inom två olika parentespar. Hur kan du använda add2 för att räkna ut 1 + 1?

b) En fördel med uppdelade parameterlistor är att man kan skapa s.k. *stegade funktioner*<sup>14</sup> där argumenten är partiellt applicerade. Prova det stegade funktionsvärdet singLa nedan. Vad skrivs ut på efter raderna 3 och 5?

```
1 scala> def repeat(s: String)(n: Int): String = s * n
2 scala> val song = repeat("doremi ")(3)
3 scala> println(song)
4 scala> val singLa = repeat("la")
5 scala> println(singLa(7))
```

★ **Uppgift 17.** *Rekursion.* En rekursiv funktion anropar sig själv.

a) Förklara vad som händer nedan.

```
1 scala> def countdown(x: Int): Unit =
2   if x > 0 then {println(x); countdown(x - 1)}
3 scala> countdown(10)
4 scala> countdown(-1)
5 scala> def finalCountdown(x: Byte): Unit =
6   {println(x); Thread.sleep(100); finalCountdown((x-1).toByte); 1 / x}
7 scala> finalCountdown(Byte.MaxValue)
```

b) Vad händer om du gör satsen som riskerar division med noll *före* det rekursiva anropet i funktionen finalCountdown ovan?

c) Förklara vad som händer nedan. Varför tar sista raden längre tid än näst sista raden?

```
1 scala> def signum(a: Int): Int = if a >= 0 then 1 else -1
2 scala> def add(x: Int, y: Int): Int =
3   if y == 0 then x else add(x + 1, y - signum(y))
4 scala> add(100, 100)
5 scala> add(Int.MaxValue, 0)
6 scala> add(0, Int.MaxValue)
```

<sup>14</sup>Kallas även Curry-funktioner efter matematikern och logikern Haskell Brooks Curry.

- ★ **Uppgift 18.** *Undersök svansrekursion genom att kasta undantag. Förklara vad som händer. Kan du hitta bevis för att kompilatorn kan optimera rekursionen till en vanlig loop?*

```
1 scala> def explode = throw Exception("BANG!!!")
2 scala> explode
3 scala> def countdown(n: Int): Unit =
4     if n == 0 then explode else countdown(n-1)
5 scala> countdown(10)
6 scala> countdown(10000)
7 scala> def countdown2(n: Int): Unit =
8     if n == 0 then explode else {countdown2(n-1); print("no tailrec")}
9 scala> countdown2(10)
10 scala> countdown2(10000)
```

- ★ **Uppgift 19.** *@tailrec-annotering. Du kan be kompilatorn att ge felmeddelande om den inte kan optimera koden till en motsvarande while-loop. Detta kan användas i de fall man vill vara helt säker på att kompilatorn kan optimera koden och det inte kan finnas risk för en överfull stack (eng. *stack overflow*) på grund av för djup anropsnästling.*

Prova nedan rader i REPL och förklara vad som händer.

```
1 scala> def countNoTailrec(n: Long): Unit =
2     if n <= 0L then println("Klar! " + n) else {countNoTailrec(n-1L); ()}
3 scala> countNoTailrec(1000L)
4 scala> countNoTailrec(100000L)
5 scala> import scala.annotation.tailrec
6 scala> @tailrec def countNoTailrec(n: Long): Unit =
7     if n <= 0L then println("Klar! " + n) else {countNoTailrec(n-1L); ()}
8 scala> @tailrec def countTailrec(n: Long): Unit =
9     if n <= 0L then println("Klar! " + n) else countTailrec(n-1L)
10 scala> countTailrec(1000L)
11 scala> countTailrec(100000L)
12 scala> countTailrec(Int.MaxValue.toLong * 2L)
```

### 3.3 Laboration: irritext

#### Mål

- ☐ Kunna skapa ett större program med din egen kod efter dina egna idéer.
- ☐ Kunna använda en editor och terminalen för att iterativt editera, kompilera, och testa din kod.
- ☐ Kunna använda variabler i kombination med alternativ och repetetition i flera nivåer.
- ☐ Kunna stegvis förbättra din kod för att underlätta förändring och öka läsbarheten.
- ☐ Kunna skapa och använda abstraktioner för att generalisera och möjliggöra återanvändning av kod.

#### Förberedelser

- ☐ Gör övning [functions](#) och repetera övning [programs](#) innan du påbörjar laborationen.
- ☐ Läs appendix [B](#) och [C](#).
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).
- ☐ Utveckla en första, spelbar version av ditt textspel, som du kan jobba vidare på under laborationen.
- ☐ Hitta någon som spelar en tidig version av ditt spel och läser din kod och ger återkoppling på kodens läsbarhet. Skriv ner den återkoppling du får.
- ☐ Spela någon annans textspel och ge återkoppling på kodens läsbarhet.

#### 3.3.1 Krav

- Du ska skapa ett lagom irriterande textspel med hjälp av en editor, till exempel VS code (se appendix [C.1.1](#)). Spelet ska köras i terminalen.
- Under redovisningen av laborationen ska du redogöra för vilka programmeringskoncept du tränat på under utvecklingen av ditt textspel. Du ska också för handledaren beskriva hur du har förbättrat din kod genom den återkoppling du fått från någon som spelat ditt spel och läst koden.
- Ditt textspel ska vara *lagom* irriterande om den som spelar har läst koden, medan spelet gärna får vara orimligt irriterande för den som *inte* läst koden. Det ska gå att klara spelet (du väljer själv vad det innebär) och därmed avsluta programmet inom rimlig tid med kännedom om koden.
- Försök göra din kod *lätt att läsa och förstå*, även om själva spelet stundtals kan vara mer eller mindre obegripligt, knasigt, eller besvärligt, för den spelare som inte har tillgång till koden... Observera att din kod inte behöver vara "perfekt" från början. Börja fritt och förbättra efterhand.
- Allteftersom ditt program blir längre ska du omforma och dela upp din kod i många, korta abstraktioner med väl valda namn för att öka läsbarheten.
- Din kod ska använda de viktiga begrepp som kursen hittills har behandlat, med speciellt fokus på det som just du behöver träna mest på.

#### 3.3.2 Tips för att komma igång

- Skapa en katalog som innehåller en scala-kodfil med valfritt namn.

- Skriv en enkel @main-metod i den nyskapade kodfilen som endast skriver ut strängen "Hello World!".
- Kompilera och kör, rätta eventuella fel tills programmet fungerar korrekt.
- När programmet fungerar, börja utöka @main-metoden i din kodfil och implementera mer funktionalitet, ta en titt under inspiration nedan.
- Börja enkelt och försök formulera vad ditt program ska göra med *psuedokod* som kommentarer innan du skriver koden.
- Kompilera och kör vid varje tillägg och håll varje tillägg så litet som möjligt, så slipper du reda ut en massa svåra följdfel vid kompilering och eventuella körtidsfel blir mer begripliga.
- Fortsätt utöka tills kraven för labben har uppnåtts.

### 3.3.3 Inspiration

Här följer en lista med olika förslag på funktioner som du kan välja bland, kombinera och variera på olika vis. Du kan också låta helt andra funktioner ingå i ditt spel. Det viktigaste är att du kombinerar kodglädje med lärorika utmaningar :)

- Be användaren logga in. Ge knasiga felmeddelande om användaren inte kan lösenordet.
- Låt användaren hamna i en irriterande oändlig loop av meningslösa frågor om den gör "fel".
- Beskriv en läskig fantasiplats där användaren befinner sig, till exempel en grotta | en källare | ett rymdskepp | Kemicentrum.
- Låt användaren välja mellan fåniga vapen, till exempel golvmopp | örontops | foliehatt | förgiftad kexchoklad.
- Låt användaren välja mellan olika vägar | dörrar | tunnlar | sektionsscaféer. Låt valet styra vilka monster som påträffas. Låt användaren bekämpa monstret med olika vapen.
- Inför någon slags poäng som redovisas under spelets gång och i slutet.
- Inför olika sorters poäng för hälsa, stridskraft, uppnådd skicklighetsnivå, etc.
- Fråga användaren om mer eller mindre relevanta detaljer: namn | skonummer | favorithusdjur. Ge knasiga kommentarer där dessa detaljer ingår som delsträngar.
- Spela sten | sax | påse med användaren.
- Spela "gissa talet" och ge ledtrådar om talet är för litet eller för stort.
- Mät hur lång tid det tar för användaren att klara ditt spel och ge poäng därefter.
- Kolla reaktionstiden hos användaren genom att mäta tiden det tar att trycka Enter efter att man fått vänta en slumpmässig tid på att strängen "NU!" skrivs ut. Om man trycker Enter innan startutskriften ges blir den uppmätta tiden 0 och på så sätt kan ditt program detektera att användaren har tryckt för tidigt. Mät reaktionstiden upprepade gånger och ge poäng efter medelvärdet.
- Låt användaren på tid så snabbt som möjligt skriva olika ord baklänges.

- Be användaren skriva en palindrom. Ge poäng efter längd.
- Träna användaren i multiplikationstabellen på tid.
- Låt användaren svara på flervalsfrågor om din favoritfilm.
- Gör det möjligt att ge ett extra argument med en "fuskkod" som ger användaren speciella förmågor eller på annat sätt underlättar för användaren under spelets gång.

## Kapitel 4

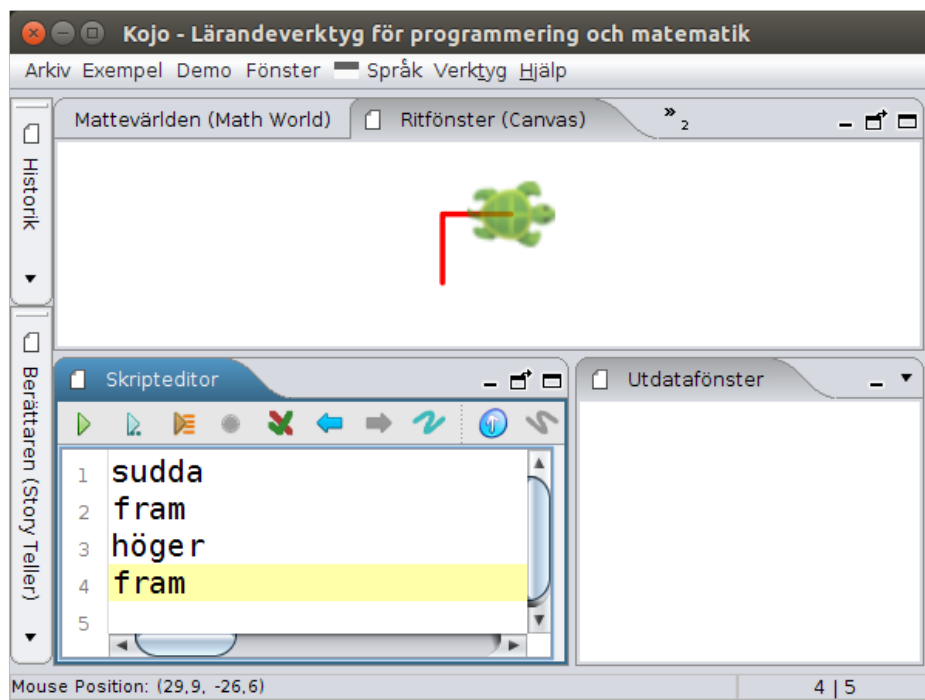
# Objekt och inkapsling

Begrepp som ingår i denna veckas studier:

- ☐ modul
- ☐ singelobjekt
- ☐ punktnotation
- ☐ tillstånd
- ☐ medlem
- ☐ attribut
- ☐ metod
- ☐ paket
- ☐ filstruktur
- ☐ jar
- ☐ dokumentation
- ☐ JDK
- ☐ import
- ☐ selektiv import
- ☐ namnbyte vid import
- ☐ export
- ☐ tupel
- ☐ multipla returvärden
- ☐ block
- ☐ lokal variabel
- ☐ skuggning
- ☐ lokal funktion
- ☐ funktioner är objekt med apply-metod
- ☐ namnrymd
- ☐ synlighet
- ☐ privat medlem
- ☐ inkapsling
- ☐ getter och setter
- ☐ principen om enhetlig access
- ☐ överlagring av metoder
- ☐ introprog.PixelWindow
- ☐ initialisering
- ☐ lazy val
- ☐ typalias

## 4.1 Teori

### 4.1.1 Vad rymmer sköldpaddan i Kojo i sitt tillstånd?



position, riktning, färg, bredd, penna uppe/nere, fyll-färg

### 4.1.2 Vad är ett objekt?

- Ett objekt är en abstraktion som...
  - kan innehålla **data** som objektet "håller reda på" och
  - kan erbjuda **operationer** som gör något eller ger ett *värde*



- Exempel: Sköldpaddan i Kojo
  - Vilken **data** sparas av sköldpaddan?  
position, riktning, pennfärg, ...
  - Vilka **operationer** kan man be sköldpaddan att utföra?  
fram, höger, vänster, ...
- Terminologi:
  - objektets data sparas i variabler som kallas **attribut**
  - alla variabelvärden utgör tillsammans objektets **tillstånd**
  - operationerna är funktioner i objektet och kallas **metoder**
  - attribut, metoder (och annat i objektet) kallas **medlemmar**

### 4.1.3 Deklarera, allokerar, referera

Olika saker man kan göra med objekt:

- **deklarera**: att skriva kod som beskriver objekt; finns flera sätt: singelobjekt, klass, tupel, ...
- **allokera**: att skapa plats i minnet för objektet vid körtid
- **referera**: att använda objektet via ett namn; man kommer åt innehållet i ett objekt med **punktnotation**: `ref.medlem`
- (**avallokera**): att frigöra minne för objekt som inte längre används; detta **sker automatiskt** i Scala, men i många andra språk, t.ex. C++, får man själv hålla reda på avallokering, vilket är knepigt och det blir lätt svåra buggar.

#### 4.1.4 Olika sätt att allokera objekt

1. Använda en **färdig funktion** som skapar ett objekt åt oss, t.ex. `apply`:

```
Vector(1,2,3) // skapa Vector-objekt med apply-metod
Vector.apply(1,2,3) // explicit apply
```

En funktion som skapar objekt kallas **fabriksmetod** (eng. *factory method*).

2. Göra **new** på en klass (mer om klasser senare):

```
new introprog.PixelWindow() // skapa ett fönsterobjekt
```

Med **new** kan man skapa **många upplagor** av samma typ av objekt.

I Scala 3 kan **new** ofta utelämnas: `introprog.PixelWindow()`

3. Deklarera ett **singelobjekt** med nyckelordet **object**
  - Ett singelobjekt finns i exakt **en** upplaga.
  - Allokeras **automatiskt** första gången man refererar objektet; man behöver inte, och kan inte, skriva **new**.
  - Medlemmar i ett Scala-singelobjekt liknar **static**-medlemmar i en Java/C++/C#-klass.
4. Använda en **tupel**, exempel: `val p = (200, 300)`

#### 4.1.5 Vad är ett singelobjekt?

- Ett singelobjekt (eng. *singleton*) deklarerar med nyckelordet **object** och används för att samla **medlemmar** (eng. *members*) som **hör ihop**.
- Ett singelobjekt kallas också **modul** (eng. *module*).
- Medlemmarna kan t.ex. vara **variabler** (**val**, **var**) och **metoder** (**def**).
- En **metod** är en **funktion** som finns i ett objekt. Metoder kallas även **operationer**.
- Exempel: singelobjekt/modul som hanterar highscore:

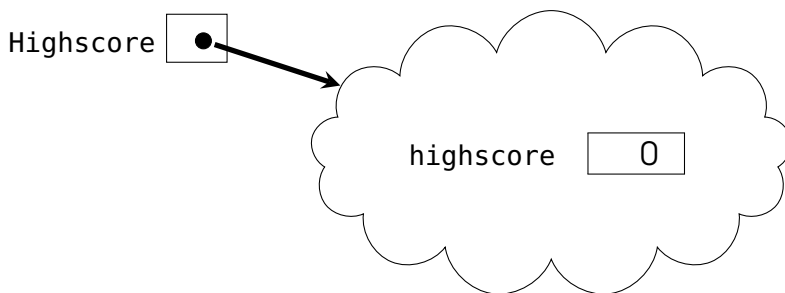
```
object Highscore {
  var highscore = 0
  def isHighscore(points: Int): Boolean = points > highscore
}
```



- Krullparenteser är valfria i Scala 3: du kan använda kolon och indentering i stället.
- Tanken är ofta att abstraktioner ska vara användbar i annan kod, för att underlätta när man bygger applikationer, och kallas då ett **API** (Application Programming Interface). Exempel: ett highscore-API.

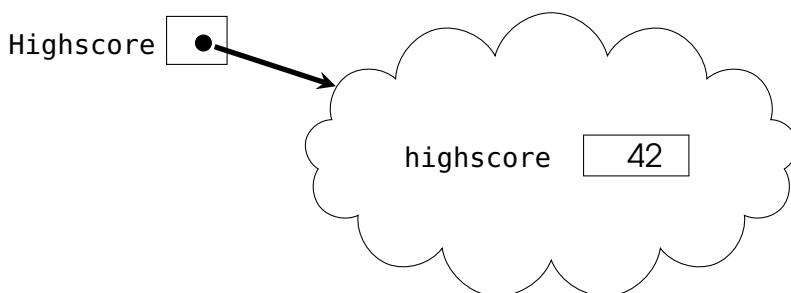
#### 4.1.6 Allokering: minne reserveras med plats för data

```
object Highscore:  
  var highscore = 0  
  def isHighscore(points: Int): Boolean = points > highscore
```



#### 4.1.7 Punktnotation, tillståndsförändring med tilldelning

```
scala> Highscore.isHighscore(5)  
res0: Boolean = true  
  
scala> Highscore.highscore = 42
```



#### 4.1.8 Punktnotation och operatornotation

Punktnotation där metदानropet har **ett** enda argument:

`objekt.metod(argument)`

kan även skrivas med infix **operatornotation**:

`objekt metod argument`

Exempel:

1 + 2

Highscore isHighscore 1000

Operatornotation med metoder vars namn börjar med bokstäver kommer i framtiden kräva deklaration med **infix** före **def**, detta för att uppmuntra konsekvent användning.

#### 4.1.9 Namnrymd och skuggning

- En **namnrymd**<sup>1</sup> (eng. *namespace*) är en omgivning (kontext) i vilken alla namn är unika. Genom att skapa flera olika namnrymder kan man undvika ”**krockar**” mellan lika namn med olika betydelser (homonymer).

Exempel: mejladresser kim@företag1.se ≠ kim@företag2.se

- Medlemmarna i ett singelobjekt finns i en egen namnrymd, där alla namn måste vara unika på samma nivå. De ”krockar” inte med namn ”utanför” objektet. Dock kan det förekomma **skuggning** (eng. *shadowing*):

```
object Game {

  val highscore = 42    // ett annat värde än Game.Highscore.highscore

  object Highscore:
    var highscore = 0    // ett annat värde än Game.highscore
    def isHighscore(points: Int): Boolean = points > highscore
}
```

#### 4.1.10 Inkapsling: att dölja interna delar

Med nyckelordet **private** döljs interna delar för omvärlden. Privata medlemmar kan bara refereras *inifrån* objektet. Denna princip kallas **inkapsling** (eng. *encapsulation*).

```
object Highscore:
  private var myHighscore = 0          // namnet myHighscore syns ej utåt
  def highscore: Int = myHighscore     // en s.k. getter ger ett attributvärde
  def isHighscore(points: Int): Boolean = points > myHighscore
  def update(points: Int): Unit = if isHighscore(points) then myHighscore = points
```

Varför har man nytta av detta?

- Förhindra att man av misstag ändrar objekts tillstånd på fel sätt.
- Förhindra användning av kod som i framtiden kan komma att ändras.
- Erbjuder en enklare ”utsida” genom dölja komplexitet ”på insidan”.
- Inte ”skräpa ner” namnrymden med ”onödiga” namn.

Nackdelar:

- Begränsar användningen, har ej tillgång till alla delar.
- Svårare att experimentera med ett API medan man försöker förstå det.

<sup>1</sup><https://sv.wikipedia.org/wiki/Namnrymd>

#### 4.1.11 Idiom: Privata variabler med understreck vid "krock"

**Idiom:** (d.v.s. ett typiskt, allmänt accepterat sätt att skriva kod)

- Om namnet på en privat variabel krockar med namnet på en getter brukar man börja det privata namnet med ett understreck:

```
object Highscore:
  private var _highscore = 0
  def highscore: Int = _highscore
  def isHighscore(points: Int): Boolean = points > _highscore
  def update(points: Int): Unit = if isHighscore(points) then _highscore = points
```

Namnkrock mellan metoder och variabler uppkommer inte i Java m.fl. språk, där dessa finns i *olika* namnrymder. Men i Scala har man valt att principen om **enhetlig access** ska gälla och alla medlemmar (både metoder och variabler) finns därmed i en gemensam namnrymd.

#### 4.1.12 Principen om enhetlig access

- I Scala så ser access av attribut och anrop av metoder, som är deklarerade utan parameterlista, likadana ut.

```
object A1 { val a = 42 }
object A2 { def a = (41 + math.random()).round.toInt }
```

```
scala> A1.a
scala> A2.a
```

- Många andra språk har olika syntax för access av attribut och anrop av metoder (t.ex. Java m.fl., där alla metodanrop måste ha parenteser).
- Fördel: Det går lätt att ändra i implementationen och växla mellan att använda attribut och använda metoder utan att den kod som använder din implementation behöver ändras.
- Nackdel: Det kan bli namnkrockar mellan metoder och attribut eftersom de finns i samma namnrymd.

#### 4.1.13 Exempel: singelobjektet med förändringsbart tillstånd

```
object mittBankkonto:
  val kontonr: Long      = 1234567L
  var saldo: Int          = 1000
  def ärSkuldsatt: Boolean = saldo < 0
```

```
scala> mittBankkonto.saldo -= 25000

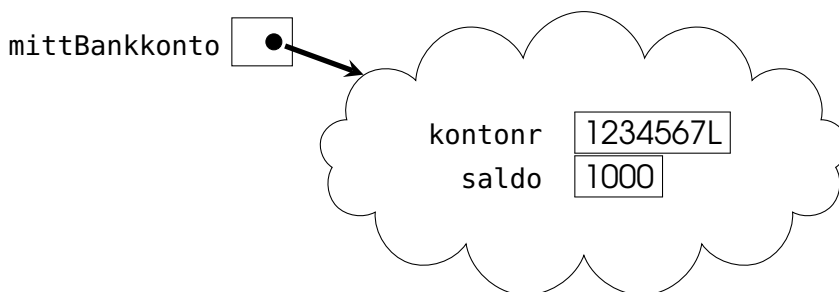
scala> mittBankkonto.ärSkuldsatt
res0: Boolean = true
```

(Vi ska i nästa vecka se hur man med s.k. klasser kan skapa många upplagor av samma typ av objekt, så att vi kan ha flera olika bankkonto.)

#### 4.1.14 Exempel: tillstånd, attribut

Ett objekts **tillstånd** är den samlade uppsättningen av värden av alla de attribut som finns i objektet.

```
object mittBankkonto
  val kontonr: Long      = 1234567L
  var saldo: Int         = 1000
  def ärSkuldsatt: Boolean = saldo < 0
```

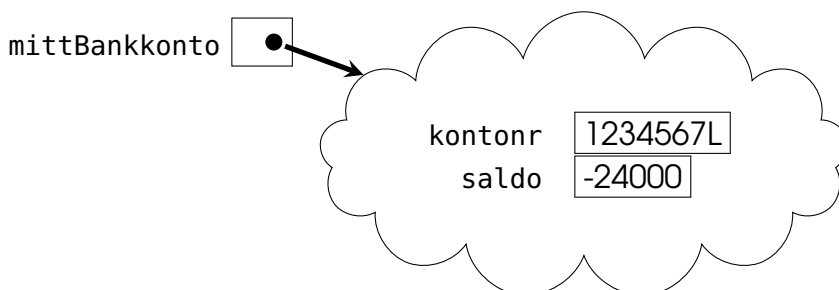


#### 4.1.15 Tillståndsändring

När en variabel tilldelas ett nytt värde sker en **tillståndsändring**. Ett **förändringsbart objekt** (eng. *mutable object*) har ett **förändringsbart tillstånd** (eng. *mutable state*).

```
scala> mittBankkonto.saldo -= 25000

scala> mittBankkonto.saldo
res1: Int = -24000
```



#### 4.1.16 Modul

- En modul samlar kod som utgör en sammanhållen, avgränsad **uppsättning abstraktioner** som kan användas av annan kod för att lösa ett specifikt (del)problem.
- I Scala finns två sätt att skapa moduler:<sup>2</sup>

<sup>2</sup>[en.wikipedia.org/wiki/Modular\\_programming](https://en.wikipedia.org/wiki/Modular_programming)

- **singelobjekt** med nyckelordet **object** och
- **paket** med nyckelordet **package**
- Liknar varandra; t.ex. kan man använda punktnotation och göra **import** på medlemmar i både singelobjekt och paket.
- Skillnader:
  - \* paket medför att **underkataloger** för maskinkoden skapas vid kompilering
  - \* objekt kan ärva medlemmar från klasser och traits (mer om det senare)

#### 4.1.17 Deklarera paket

Med nyckelordet **package** först i en kodfil ges alla deklarationer en gemensam namnrymd. Denna kod ligger i filen `f1.scala`:

```
package mittpaket

object A:
  def hälsa: Unit = println(B.hälsning)
```

Denna kod ligger i filen `f2.scala`:

```
package mittpaket

object B:
  def hälsning: String = "hejsan"
```

Singelobjekten A och B finns båda i namnrymden `mittpaket`.

#### 4.1.18 Kompilera paket

Paketdeklarationer medför att kompilatorn placerar bytekodfiler i en katalog med samma namn som paketet:

```
1 > scalac f1.scala f2.scala // samkompilering av två filer
2 > ls
3 f1.scala f2.scala mittpaket
4 > ls mittpaket
5 A.class 'A$.class' A.tasty
6 B.class 'B$.class' B.tasty
```

Idiom, syntax och semantik:

- Paketnamn brukar bestå av enbart små bokstäver.
- Om paketnamn innehåller punkt(er), skapas nästlade underpaket, exempel: `p1.p2.p3` kompilerar kod till katalogen `p1/p2/p3`
- Du kan ha flera paket och även nästlade paket i **samma** kodfil, genom att använda klammerparentes (eller kolon+indentering):

```
package p1 { object A; package p2 { object B }}
```

### 4.1.19 Paket i REPL

Paket funkar inte i REPL:

```
scala> package mittpaket { def hej = println("Hej") }
-- [E103] Syntax Error: -----
1 |package mittpaket { def hej = println("Hej") }
  |^^^^^^
  |this kind of statement is not allowed here
```

### 4.1.20 Vad är en tupel?

- En  $n$ -tupel är ett objekt som samlar  $n$  st objekt i en enkel datastruktur med koncis syntax; du behöver bara parenteser och kommatecken för att skapa tupel-objekt: `(1, 'a', "hej")`
- Elementen kan alltså vara av **olika** typ.
- `(1, 'a', "hej")` är en **3-tupel** av typen: `(Int, Char, String)`
- Du kan komma åt de enskilda elementen med **`_1`, `_2`, ... `_n`**
- Du kan även använda **`apply(0)`, `apply(1)`, ... `apply(n-1)`**

```
1 scala> val t = ("hej", 42, math.Pi)
2 t: (String, Int, Double) = (hej,42,3.141592653589793)
3
4 scala> t._1    // direkt access
5 res0: String = hej
6
7 scala> t(1)    // notera användningen av apply
8 res1: Int = 42
```

- Tupler är praktiska när man inte vill ta det lite större arbetet att skapa en egen klass. (Men med klasser kan man göra mycket mer än med tupler.)

### 4.1.21 Tupler som parametrar och returvärde.

- Tupler är smidiga som **parametrar** om man vill kombinera värden som hör ihop, till exempel  $x$ - och  $y$ -värdena i en punkt: `(3, 4)`
- Tupler är smidiga när man på ett enkelt och typsäkert sätt vill låta en funktion **returnera mer än ett värde**.

```
scala> def längd(p: (Double, Double)): Double = math.hypot(p._1, p._2)

scala> def vinkel(p: (Double, Double)): Double = math.atan2(p._1, p._2)

scala> def polär(p: (Double, Double)): (Double, Double) = (längd(p), vinkel(p))

scala> polär((3,4))
res2: (Double, Double) = (5.0,0.6435011087932844)
```

- Om typerna passar kan man skippa dubbla parenteser vid **ensamt tupel-argument**:

```
1 scala> polär(3,4)
2 res3: (Double, Double) = (5.0,0.6435011087932844)
```

[https://sv.wikipedia.org/wiki/Polära\\_koordinater](https://sv.wikipedia.org/wiki/Polära_koordinater)

---

#### 4.1.22 Ett smidigt sätt att skapa 2-tupler med metoden ->

Det finns en metod vid namn `->` som kan användas på objekt av **godtycklig** typ för att **skapa par**:

```
1 scala> ("Ålder", 42)
2 res0: (String, Int) = (Ålder,42)
3
4 scala> "Ålder".->(42)
5 res1: (String, Int) = (Ålder,42)
6
7 scala> "Ålder" -> 42
8 res2: (String, Int) = (Ålder,42)
9
10 scala> Vector("Ålder" -> 42, "Längd" -> 178, "Vikt" -> 65)
11 res3: scala.collection.immutable.Vector[(String, Int)] =
12     Vector((Ålder,42), (Längd,178), (Vikt,65))
```

---

#### 4.1.23 Typalias för att abstrahera typnamn

Med hjälp av nyckelordet **type** kan man deklarerera ett **typalias** för att ge ett **alternativt** namn till en viss typ. Exempel:

```
1 scala> type Pt = (Int, Int)           // typalias
2 scala> type Pts = Vector[Pt]         // nästlat typalias
3
4 scala> def distToOrigo(pt: Pt): Double = math.hypot(pt._1, pt._2)
5
6 scala> val xs: Pts = Vector((1,1), (2,2), (3,4))
7 val xs: Pts = Vector((1,1), (2,2), (3,4))
8
9 scala> xs.head
10 val res0: Pt = (1,1)
11
12 scala> xs.map(distToOrigo)
13 val res1: Vector[Double] = Vector(1.4142135623730951, 2.8284271247461903, 5.0)
```

Typalias kan vara bra när:

- man har en lång och krånglig typ och vill använda ett kortare namn,
- man vill kunna lätt byta implementation senare (t.ex. om man vill använda en egen klass i stället för en tupel).

---

#### 4.1.24 Lata variabler och fördröjd initialisering

Med nyckelordet **lazy** före **val** sker **"lat"** evaluering av initialiseringsuttrycket. Motsatsen (det normala i Scala) kallas **strikt** evaluering.

```

1 scala> val strikt = Vector.fill(1000000)(math.random())
2 strikt: scala.collection.immutable.Vector[Double] =
3   Vector(0.7583305221813246, 0.9016192590993339, 0.770022134260162, 0.15667718184929746, ...
4
5 scala> lazy val lat = Vector.fill(1000000)(math.random())
6 lat: scala.collection.immutable.Vector[Double] = <lazy>
7
8 scala> lat
9 res0: scala.collection.immutable.Vector[Double] =
10   Vector(0.5391685014341797, 0.14759775960530275, 0.722606095900537, 0.9025572787055386, ...

```

En **lazy val** initialiseras **inte** vid deklarationen utan när den **refereras första gången**. Uttrycket som anges i deklarationen evalueras med s.k. **fördröjd evaluering** (även "lat" evaluering).

#### 4.1.25 Singelobjekt är lata

- Singelobjekt allokeras **inte** direkt vid deklaration; allokeringen sker först då objektet refereras första gången.
- Exempel:

```

object mittLataObjekt:
  println("jag är lat")
  val storArray = { println("skapar stor Array"); Array.fill(10000)(42) }
  lazy val ännuStörreArray = Array.fill(Int.MaxValue)(42)

```

När sker utskrifterna?

När allokeras variablerna?

#### 4.1.26 Vad är skillnaden mellan val, var, def, lazy val?

```

object exempel:
  println("hej exempel")
  val förAlltidSammaReferens = {println("hej val"); math.random()}
  var kanÄndrasMedTilldelning = {println("hej var"); math.random()}
  def evaluerasVidVarjeAnrop = {println("hej def"); math.random()}
  lazy val fördröjdInit = {println("hej lazy val"); math.random()}

```

I vilken ordning sker utskrifterna?

Lat evaluering är en viktig princip inom funktionsprogrammering som möjliggör effektiva, oföränderliga datastrukturer där element allokeras först när de behövs.

[en.wikipedia.org/wiki/Lazy\\_evaluation](https://en.wikipedia.org/wiki/Lazy_evaluation)

#### 4.1.27 Be kompilatorn att varna vid initialiseringsproblem

Initialisering i fel ordning kan ge oväntade överraskningar:

```

scala> { val b = a; val a = 42 }
val b: Int = 0    // default-värdet för Int är noll och a har ännu inte fått värdet 42
val a: Int = 42

```



```
scala> :settings -Ysafe-init

scala> { val b = a; val a = 42 }
1 warning found
-- Warning: -----
1 | { val b = a; val a = 42 }
  |           ^
  |           |
  |           | Access non-initialized value a.
val b: Int = 0
val a: Int = 42
```

Med kompilator-optionen `-Ysafe-init` får du en välbehövlig varning. Skriv såhär, t.ex. i `project.scala`, om du vill ha denna option påslagen:

```
//> using option -Ysafe-init
```

#### 4.1.28 Be kompilatorn ge fler bra varningar

Slå på mer utförliga meddelanden och varningar:

```
//> using option -unchecked -deprecation -Wunused:all -Wvalue-discard -Ysafe-init
```

<code>-unchecked</code>	Extra varningar vid flera fall av osäker kod.
<code>-deprecation</code>	Förklaring vid användning av utgående funktioner.
<code>-Wunused:all</code>	Varning om deklarationer ej används.
<code>-Wvalue-discard</code>	Varning vid förlorat värde.
<code>-Ysafe-init</code>	Varna vid användning av ännu ej initialiserade variabler.

Om du tycker vissa varningar är irriterande kan du slå av dem med `@annotation.nowarn`

```
scala> { val b = a; @annotation.nowarn val a = 42 }
```

#### 4.1.29 Programmeringsparadigm

[en.wikipedia.org/wiki/Programming\\_paradigm](https://en.wikipedia.org/wiki/Programming_paradigm):

- **Imperativ programmering**: programmet är uppbyggt av sekvenser av olika satser som läser och **ändrar** tillstånd
- **Objektorienterad programmering**: en sorts imperativ programmering där programmet består av objekt som kapslar in tillstånd och erbjuder operationer som läser och **ändrar** tillstånd.
- **Funktionsprogrammering**: programmet är uppbyggt av samverkande (äkta) funktioner som **undviker** föränderlig data och tillståndsändringar. Oföränderliga datastrukturer skapar effektiva program i kombination med lat evaluering och rekursion.

### 4.1.30 Funktioner är äkta objekt i Scala

Scala visar hur man kan **förena** (eng. *unify*) **objektorientering** och **funktionsprogrammering**:

**En funktion är ett objekt som har en apply-metod.**

```
scala> object öka:
  def apply(x: Int) = x + 1

scala> öka.apply(1)
res0: Int = 2

scala> öka(1)    // metoden apply behöver ej skrivas explicit
res1: Int = 2
```

---

### 4.1.31 Fördjupning: Äkta funktionsobjekt är av funktionstyp

Egentligen, mer precist:

**En funktion är ett objekt av funktionstyp som har en apply-metod.**

```
scala> object öka extends (Int => Int):
  def apply(x: Int) = x + 1

scala> öka(1)
res2: Int = 2

scala> Vector(1,2,3).map(öka)
res3: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)

scala> öka.    // tryck TAB
andThen  apply  compose  toString  ...
```

Mer om **extends** senare i kursen...

---

### 4.1.32 Vad är en klass?

Singelobjekt finns bara i exakt EN upplaga:

```
object mittBankkonto:
  val kontonr: Long      = 1234567L
  var saldo: Int         = 1000
  def ärSkuldsatt: Boolean = saldo < 0
```

Om vi vill ha flera bankkonton behöver vi en **klass** (eng. *class*).

---

### 4.1.33 Vad är en klass?

En klass kan användas för att skapa många objekt av samma typ. Varje upplaga har sitt eget tillstånd och kallas en **instans** av klassen (mer om detta nästa vecka).

```
class Bankkonto(val kontonr: Long, var saldo: Int): // klassbeskrivning
  def ärSkuldsatt: Boolean = saldo < 0
```

```
1 scala> val bk1 = new Bankkonto(1234567L, 1000 ) // instansiera en klass
2 bk1: Bankkonto = Bankkonto@5d7399f9
3
4 scala> val bk2 = new Bankkonto(6789012L, -200 )
5 bk2: Bankkonto = Bankkonto@286855ea
6
7 scala> bk1.saldo
8 res0: Int = 1000
9
10 scala> bk2.ärSkuldsatt
11 res1: Boolean = true
```

---

#### 4.1.34 Använda klassen Color

- I JDK (Java Development Kit) finns hundratals paket (moduler) och tusentals färdiga klasser.<sup>3</sup>
- En av dessa klasser heter Color och ligger i paketet `java.awt` och används för att representera RGB-färger med ett tal som beskriver andelen Rött, Grönt och Blått.

```
1 scala> val röd = java.awt.Color(255, 0, 0) // en maximalt röd färg
2
3 scala> import java.awt.Color // namnet Color tillgängligt i aktuell namnrymd
4
5 scala> Color. // tryck TAB och se alla publika medlemmar
```

- Använd klassen `java.awt.Color` på veckans övning.
- Hur ska jag veta hur jag kan använda en färdig klass?
  1. Läs koden, visar "insidan" med all sin komplexitet; kan vara knepigt...
  2. Läs **dokumentationen**, visar "utsidan" som är enklare (?) än "insidan"
  3. **Experimentera** med hjälp av REPL och/eller en IDE

---

#### 4.1.35 Lägg till metoder i efterhand med extension

- Ofta vill man kunna lägga till metoder på godtyckliga typer i efterhand, speciellt när det gäller typer som finns i kod som någon annan skrivit.
- Detta går att göra i Scala med nyckelordet **extension**:  
**extension** (s: String) **def** skrikBaklänges = s.reverse.toUpperCase
- En **extensionsmetod** kan anropas med **punktnotation** som om den vore en medlem av typen.
- Det går också att anropa en extensionsmetod som en fristående funktion utan punktnotation.

```
1 scala> extension (s: String) def skrikBaklänges = s.reverse.toUpperCase
2 def skrikBaklänges(s: String): String
3
```

---

<sup>3</sup><https://stackoverflow.com/questions/3112882/>

```
4 scala> "hejsan".skrikBaklänges
5 val res1: String = NASJEH
6
7 scala> skrikBaklänges("goddag")
8 val res2: String = GADDOG
```

---

#### 4.1.36 Kollektiva extensionsmetoder

- Det går bra att sammanföra flera funktioner under en och samma **extension** så här:

```
extension (s: String)
  def baklänges = s.reverse
  def skrik = s.toUpperCase
```

- Detta kallas **kollektiva extensionsmetoder**.
- Notera att det *inte* ska vara något kolon efter **extension**-deklarationens första rad.

---

#### 4.1.37 Import av alla namn i en viss modul

- Man kan importera **alla** namn i en viss modul (singelobjekt eller paket). Detta kallas på engelska för *wildcard import*.
  - Syntax: **import** p1.p2.\*

- Exempel:

```
1 scala> import java.awt.* // importera ALLA namn i paketet awt
```

- **Fördelar:**
  1. Slipper skriva import på varje enskilt namn.
  2. De abstraktioner som är tänkta att användas tillsammans blir alla synliga i aktuell namnrymd (eng. *in scope*).
- **Nackdelar:**
  1. Kan ge namnkrockar och svåra buggar vid namnskuggning.
  2. Man "skräpar ner" sin namnrymd med namn som kanske inte är tänkta att användas, men som vid misstag, t.ex. felstavning, ändå ger effekt.
  3. Man kan inte genom att studera import-deklarationerna se exakt vilka namn som används, vilket kan göra det svårare att förstå vad koden gör.

---

#### 4.1.38 Namnbyte vid import

- Man kan undvika namnkrockar med **namnbyte vid import**.
- Syntax: **import** p1.p2.befintligtNamn **as** nyttNamn
- Exempel:

```

1 scala> import java.awt.Color as JColor //importera och byt namn
2
3 scala> val grön = JColor(0, 255, 0) //skapa instans med nya namnet
4 grön: java.awt.Color = java.awt.Color[r=0,g=255,b=0]

```

#### 4.1.39 Exkludera (gömma) namn vid import

- Man kan undvika namnkrockar vid import genom att exkludera vissa namn (eng. *import hiding*).
- Syntax: **import** p1.p2.exkluderaMig **as** \_
- Exempel:

```

1 scala> import java.awt.{Event as _, *} // importera allt UTOM Event

```

- Kan kombineras med namnbyte och allimport:

```

1 scala> import java.awt.{Event as _, Color as JColor, *}

```

#### 4.1.40 Lokal import-deklaration

- Man kan begränsa "nedskräpningen" av namnrymden genom att göra import-deklarationer så lokalt som möjligt, till exempel i ett objekt eller i en funktionskropp.
- Exempel:

```

object A:
  def x =
    import java.awt.Color.RED
    /* ... namnet RED syns bara lokalt i denna funktion */

```

#### 4.1.41 Export

- **import** ger direkt synlighet **lokalt** inuti en namnrymd
- Med **export** kan du göra *motsatsen* till import: göra medlemmar direkt synliga **utanför** en namnrymd.

```

object A:
  import java.awt.Color.* // gör färger synliga direkt inuti detta objekt
  def test = RED          // färgen RED synlig direkt i lokala namnrymden

object B:
  export java.awt.Color.* // RED blir medlem som syns utåt via B.RED
  export math.{sin, cos}  // sin och cos blir metoder i B

```

```
scala> A.RED
-- [E008] Not Found Error: -----
1 |A.RED
  |^^^^
  |value RED is not a member of object A

scala> B.RED
val res0: java.awt.Color = java.awt.Color[r=255,g=0,b=0]

scala> (B.cos(0), B.sin(0))
val res1: (Double, Double) = (1.0,0.0)
```

---

#### 4.1.42 Använda dokumentation för färdiga klasser.

- Dokumentation för standardbiblioteket i Scala finns här:  
<https://www.scala-lang.org/api/>
- Övning: Leta upp dokumentationen för metoden `reduceLeft` i klassen `Vector`.
- Dokumentation för standardbiblioteket i Java finns här:  
<https://docs.oracle.com/en/java/javase/11/docs/api/index.html>
- Övning: Leta upp dokumentationen för `java.awt.Color`
- Läs mer i Appendix E om dokumentation.

---

#### 4.1.43 Vad är en jar-fil?

- Jar-filer används för att distribuera färdigkompilerad kod så att andra kan använda den enkelt
- Förkortningen **jar** kommer från "Java Archive"
- En **jar**-fil följer ett standardiserat filformat och används för att **paketera flera filer** i en och samma fil, exempelvis:
  - .class-filer med bytekod
  - resursfiler för en applikation t.ex. bilder .png, .jpg, etc
  - information om vilken klass som innehåller main-funktionen
  - etc.
- En .jar-fil komprimeras på samma sätt som en .zip-fil.
- Fördjupning för den intresserade:  
[https://en.wikipedia.org/wiki/JAR\\_\(file\\_format\)](https://en.wikipedia.org/wiki/JAR_(file_format))

---

#### 4.1.44 Öppen källkod på Maven Central

- På **Maven Central** som hanteras av företaget Sonatype finns tusentals öppet tillgängliga kodbibliotek publicerade som jarfiler.
- Du kan söka bland alla Scala-bibliotek här:  
<https://index.scala-lang.org/>

- Du kan söka bland alla bibliotek här:  
<https://search.maven.org/>

---

#### 4.1.45 Vad är *classpath*?

- Hur hittar kompilatorn färdiga moduler?
- Kompilatorerna `scalac` och `javac` och programmen `scala-cli` och `java` som kör igång JVM använder **en lista med filsökvägar** kallad **`classpath`** när de söker efter kompilerad kod.
- Scalas standardbibliotek läggs automatiskt på `classpath`.
- Med hjälp av optionen `--jar` kan du lägga till en jar-fil till `classpath`.
- Exempel: (punkt används för att ange aktuell katalog)

```
scala-cli run . --jar introprog.jar
```

---

#### 4.1.46 Färdiga grafikmetoder i klassen `PixelWindow`

- På labben ska du använda en `.jar`-fil med kodbiblioteket `introprog`.
- Där finns klassen `PixelWindow` som kan skapa ritfönster.
- Du kan starta REPL så här om du har laddat ner jar-filen manuellt från  
<https://fileadmin.cs.lth.se/introprog.jar>

```
> scala-cli repl --jar introprog.jar
```

- Testa `PixelWindow` i REPL med:

```
scala> val w = introprog.PixelWindow(300, 200, "hejsan")
```

- Studera dokumentationen för `introprog.PixelWindow` här:  
<http://cs.lth.se/pgk/api/>

---

#### 4.1.47 Automatiska beroenden med Scala CLI i REPL:

- Du kan istället låta `scala-cli` **automatiskt** ladda ner ett färdigt kodbibliotek som är publicerat på Maven Central och lägga det på `classpath` med optionen `--dep` som är en förkortning av *dependency*.
- Notera antalet kolon i adressen till kodbiblioteket:

```
> scala-cli repl . --dep se.lth.cs::introprog:1.3.1
Welcome to Scala 3.1.3 (17.0.3, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> introprog.Dialog.show("hello introprog")
```

#### 4.1.48 Köra program + kodbibliotek med Scala CLI

- `scala-cli` kan inkludera kodbibliotek från Maven Central om du skriver en ”magisk” kommentar i början av din `.scala`-filen:

```
//> using scala 3.3
//> using lib se.lth.cs::introprog:1.3.1

@main def run = introprog.Dialog.show("hello introprog")
```

Notera > efter //

- När du kör ditt program såhär så kommer Scala CLI att ladda ner kodbiblioteket om det inte redan är gjort:

```
> scala-cli run .
```

- Läs mer här:  
<https://index.scala-lang.org/lunduniversity/introprog-scalalib> och i Appendix C, stycket om Scala CLI. Mer om `//> using` här:  
<https://scala-cli.virtuslab.org/docs/reference/directives>

---

#### 4.1.49 Kompilera om vid varje ändring

Ange optionen `--watch` så körs kommandot om varje gång du sparar en `scala`-fil med `Ctrl+S`.

```
> scala-cli compile . --watch
```

Kan skrivas kortare:

```
> scala-cli compile . -w
```

Fungerar också för `run`-kommandot, men det är inte lika användbart om appen är interaktiv och väntar på input från användaren innan den avslutas.

```
> scala-cli run . -w
```

**Gör så små ändringar som möjligt** och kompilera och testa vid **varje** ändring! Många ändringar kan ge svårhittade följdfel...

---



## 4.2 Övning objects

### Mål

- ☐ Kunna skapa och använda objekt som moduler.
- ☐ Kunna förklara hur nästlade block påverkar namnsynlighet och namnöverskuggning.
- ☐ Kunna förklara begreppen synlighet, privat medlem, namnrymd och namnskuggning.
- ☐ Kunna skapa och använda tupler.
- ☐ Kunna skapa funktioner som har multipla returvärden.
- ☐ Kunna förklara den semantiska relationen mellan funktioner och objekt i Scala.
- ☐ Kunna förklara kopplingen mellan paketstruktur och kodfilstruktur.
- ☐ Kunna använda en jar-fil och classpath.
- ☐ Kunna använda import av medlemmar i objekt och paket.
- ☐ Kunna byta namn vid import.
- ☐ Kunna förklara skillnaden mellan import och export.
- ☐ Kunna skapa och använda variabler med fördröjd initialisering.

### Förberedelser

- ☐ Studera begreppen i kapitel 4
- ☐ Läs om hur man fixar buggar i appendix D.

### 4.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** *Para ihop begrepp med beskrivning.*

Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

modul	1	A	funktion som är medlem av ett objekt
singelobjekt	2	B	modul som kan ha tillstånd; finns i en enda upplaga
paket	3	C	kodenhet med abstraktioner som kan återanvändas
import	4	D	modul som skapar namnrymd; maskinkod får egen katalog
export	5	E	tillhör ett objekt; nås med punktnotation om synlig
lat initialisering	6	F	gör namn tillgängligt lokalt utan att hela sökvägen behövs
medlem	7	G	allokering sker först när namnet refereras
attribut	8	H	variabel som utgör (del av) ett objekts tillstånd
metod	9	I	omgivning där är alla namn är unika
privat	10	J	metoder med samma namn men olika parametertyper
överlagring	11	K	modifierar synligheten av en objektmedlem
namnskuggning	12	L	lokalt namn döljer samma namn i omgivande block
namnrymd	13	M	ändring mellan def och val påverkar ej användning
enhetlig access	14	N	alternativt namn på typ som ofta ökar läsbarheten
punktnotation	15	O	används för att komma åt icke-privata delar
typalias	16	P	gör namn synligt utåt som medlem i detta objekt

**Uppgift 2.** *Nästlade singelobjekt, import, synlighet och punktnotation.* I den tvådimensionella Underjorden bor Mullvaden och Masken. Masken har gömt sig för Mullvaden och befinner sig på en plats långt bort. Masken har även gjort delar av sin position osynlig för omvärlden:

**object** Underjorden:

```
var x = 0
var y = 1
```

**object** Mullvaden:

```
var x = Underjorden.x + 10
var y = Underjorden.y + 9
```

**object** Masken:

```
private var x = Mullvaden.x
var y = Mullvaden.y + 190
def ärMullvadsmat: Boolean = ???
```

- Skapa ovan kod i filen `Underjorden.scala` med en editor och implementera predikatet `ärMullvadsmat` så att det blir sant om mullvadens koordinater är samma som maskens.
- Testa livet i Underjorden genom att klistra in din modul i REPL. Importera Underjordens medlemmar med asterisk så att du ser Mullvaden och Masken. Flytta med hjälp av tilldelning Maskens y-koordinat så att Masken hamnar på samma plats som Mullvaden. Kontrollera att predikatet `ärMullvadsmat` fungerar som tänkt.
- Importera därefter allt i Mullvaden och sedan allt i Masken och tilldela `x` ett nytt värde enligt raderna 1–3 nedan. Vad ger uttrycken på raderna 4–6 nedan för värde? Förklara vad som händer i termer av namnöverskuggning och synlighet?

```
1 scala> import Mullvaden.*
2 scala> import Masken.*
3 scala> x = -1
4 scala> Mullvaden.x
5 scala> Masken.x
6 scala> Underjorden.x
```

**Uppgift 3.** *Export.*

- Jämför **import** och **export** genom att beskriva en likhet och en skillnad.
- Skapa ett exempel i REPL som demonstrerar nyttan med **export**.

**Uppgift 4.** *Tupler.* Tupler sammanför flera olika värden i ett oföränderligt objekt. Nedan används tupler för att representera en 3D-punkt i underjorden med koordinater (`x`, `y`, `z`) av typen (`Int`, `Int`, `Double`), där `z`-koordinaten anger hur djupt ner i underjorden punkten ligger. På en hemlig plats finns uppgången till överjorden.

**object** Underjorden3D:

```
private val hemlis = ("uppgången till överjorden", (0, 0, 0.0))
```

**object** Mullvaden:

```
var pos = (5, 3, math.random() * 10 + 1)
def djup = ???
```

```
object Masken:
  private var pos = (0, 0, 10.0)
  def ärMullvadsmat: Boolean = ???
  def ärRaktUnderUppgången: Boolean = ???
```

- Funktionen djup ska ge  $z$ -koordinaten för Mullvaden. Vilken typ har djup?
- Vilken typ har hemlis?
- Skriv in koden för Underjorden3D i en editor och implementera de saknade delarna. Predikatet ärMullvadsmat ska vara sant om Masken finns på samma plats som Mullvaden. Predikatet ärRaktUnderUppgången ska vara sant om  $x$ - och  $y$ -koordinaterna sammanfaller med den hemliga uppgången till överjorden. Testa så att dina implementationer fungerar i REPL.
- En tupel med  $n$  värden kallas  $n$ -tupel. Om man betraktar det tomma värdet () som en tupel, vad kan man då kalla detta värde?

**Uppgift 5.** *Lat initialisering.* Med **lazy val** kan man fördröja initialiseringen.

- Vad ger raderna 2 och 3 nedan för resultat?

```
1 scala> lazy val z = { println("nu!"); Array.fill(1e1.toInt)(0) }
2 scala> z
3 scala> z
```

- Prova ovan igen men med så stor array att minnet blir fullt. När sker allokeringen?
- Singelobjekt är lata. Initialiseringsordningen kan bli fel.

```
object test:
  object zzz { val a = { println("nu!"); 42 } }
  object buggig { val a = b ; val b = 42 }
  object funkar { lazy val a = b ; val b = 42 }
```

Klistra in modulen test i REPL. När skrivs "nu!" ut?

- Vad händer i REPL om du refererar de tre olika a-variablerna?
- Vad är det för skillnad på **lazy val** a = uttryck och **def** b = uttryck ?

**Uppgift 6.** *Extensionsmetoder.* Extensionsmetoder möjliggör punktnotation på värden av befintliga typer.

- Skapa extensionsmetod på heltal som möjliggör inkrementering.

```
scala> 42.inc
val res0: Int = 43
```

- Skapa extensionsmetod på heltal som möjliggör dekrementering.

```
scala> 42.dec
val res1: Int = 41
```

- Sammanför extensionsmetoderna så att de blir *kollektiva*, alltså under en och samma **extension**. Använd även math.incrementExact och math.decrementExact efter att du sökt upp dokumentationen för dessa här: <https://docs.oracle.com/en/java/javase/17/docs/api/>

d) Vad är fördelen med `math.incrementExact` och `math.decrementExact`?

**Uppgift 7. Jar-fil. Classpath. Paket.** En jar-fil används för att samla färdigkompilerade program, kod, dokumentation, resursfiler, etc, i en enda fil. En jar-fil är komprimerad på samma sätt som en zip-fil. I kursen använder vi ett paket med namnet `introprog` som ligger i en jarfil som heter något i stil med `introprog_3-1.3.1.jar` (eller senare version) där första numret anger den Scala-version som biblioteket är kompilerat för och andra numret anger bibliotekets version som ändras vid varje ny utgåva.

a) På veckans laboration ska vi använda klassen `PixelWindow` som finns i paketet `introprog`. Vilka parametrar har klassen `PixelWindow` och vilka defaultargument finns? Hur skriver man om man vill skapa en `PixelWindow`-instans?

*Tips:* Läs dokumentationen av `PixelWindow` här: <http://cs.lth.se/pgk/api/> och leta efter beskrivningen av klassens konstruktor.

b) Ladda ner senaste utgåvan av jar-filen med `introprog`-paketet här:

<https://github.com/lunduniversity/introprog-scalalib/releases>

Spara filen som heter `introprog_3-1.3.1.jar` (eller senare version) på lämplig plats.

c) Testa `PixelWindow` i REPL enligt nedan. Använd optionen `-jar` med jar-filens namn som argumentet. Skriv kod som ritar en kvadrat med sidan 100 och som har sitt vänstra, övre hörn i punkten (100,100), genom att fortsätta på nedan påbörjade kod (anpassa namnet på jar-filen efter den version som du laddat ned):

```
1 > scala-cli repl --jar introprog_3-1.3.1.jar
2 scala> val w = introprog.PixelWindow(400,300,"HEJ")
3 scala> w.line(100, 100, 200, 100)
4 scala> w.line(200, 100, 200, 200)
5 scala> // fortsätt så att en hel kvadrat ritas
```

d) Skriv nedan program med en editor i filen `hello-window.scala` och fyll i de saknade delarna så att en röd kvadrat ritas ut, med ledning av dokumentationen:

<http://cs.lth.se/pgk/api/>

```
package hello

object Main:
  val w = new introprog.PixelWindow(400, 300, "HEJ")

  var color = java.awt.Color.red

  /** Kvadrat med övre hörnet i punkten p och storleken side pixlar. */
  def square(p: (Int, Int))(side: Int): Unit =
    if side > 0 then
      // side == 1 ger en kvadrat som är en enda pixel
      val d = side - 1

      w.line(p._1,      p._2,      p._1 + d, p._2,      color)
      w.line(p._1 + d, p._2,      p._1 + d, p._2 + d, color)
      w.line(p._1 + d, p._2 + d, p._1,      p._2 + d, color)
      ???

  def main(args: Array[String]): Unit =
```

```
println("Rita kvadrat:")
square(300,100)(50)
```

Kör programmet med

```
> scala-cli run hello-window.scala --jar introprog_3-1.3.1.jar
Found several main classes. Which would you like to run?
[0] hello.Main
[1] introprog.examples.TestBlockGame
[2] introprog.examples.TestIO
[3] introprog.examples.TestPixelWindow
```

Det finns, förutom ditt eget huvudprogram vid namn `hello.Main`, flera exempel-huvudprogram i paketet `introprog.examples`. När flera huvudprogram detekteras får du frågan vilket du vill köra. Välj ditt eget huvudprogram.

e) Du kan slippa frågan om du explicit pekar ut huvudprogrammet genom att lägga till optionen `--main-class`. Prova det!

f) Du kan slippa själv ladda ner `introprog` med hjälp av optionen `--dep` vid körning i terminalen, vilket beskrivs i bibliotekets `README.md` på github här:

<https://github.com/lunduniversity/introprog-scalalib>

Prova det!

g) Du kan också lägga in beroendet inne i din kodfil med en magisk kommentar, vilket även det beskrivs i ovan nämnda `README.md`. Prova det!

**Uppgift 8. Färg.** Det finns många sätt att beskriva färger. I naturligt språk har vi olika namn på färgerna, till exempel *vitt*, *rosa* och *magenta*. I bildminnen i datorer är det vanligt att beskriva färger som en blandning av *rött*, *grönt* och *blått* i det så kallade RGB-systemet.

På veckans labb ska vi använda `PixelWindow`, som beskriver RGB-färger med klassen `java.awt.Color`. Det finns några fördefinierade färger i `java.awt.Color`, till exempel `java.awt.Color.black` för svart och `java.awt.Color.green` för grönt, se vidare dokumentationen för `java.awt.Color` i JDK<sup>4</sup>. Andra färger kan skapas genom att du själv anger den specifika mängden rött, grönt och blått som behövs för att blanda en viss färg. De tre parametrarna till **new** `java.awt.Color(r, g, b)` anger hur mycket *rött*, *grönt* respektive *blått* som färgen ska innehålla, och mängderna ska vara i intervallet 0–255. Färgen (153,102,51) innebär ganska mycket rött, lite mindre grönt och ännu mindre blått och det upplevs som brunt.

a) På laborationen behöver du dessa tre brunaktiga färger och det är smidigt att samla dem i en egen namnrymd via ett singelobjekt som heter `Color` enligt nedan.

```
object Color:
  val mole    = new java.awt.Color( 51,  51,  0)
  val soil    = new java.awt.Color(153, 102,  51)
  val tunnel  = new java.awt.Color(204, 153, 102)
```

Men vi vill helst göra import på `java.awt.Color` för att kunna använda klassens namn utan att upprepa hela sökvägen, trots att namnet krockar med namnet på vårt singelobjekt. Skriv om koden ovan med hjälp av namnbyte vid import så att färgerna kan skapas med **new** `JColor(...)`. Gör importen lokalt i singelobjektet `Color`.

<sup>4</sup><https://docs.oracle.com/en/java/javase/17/docs/api/>

b) Inspireras av REPL-experimenten nedan och ändra ditt program i `hello-window.scala` så att *tre* överlappande färgfyllda kvadrater ritas enligt den övre bilden till höger. I stället för att rita med den färdiga metoden `fill` som finns i `PixelWindow`, ska du träna på iteration genom att själv implementera ritprocedurerna `rak` och `fyll` enligt nedan. Proceduren `rak` ska rita en horisontell linje med vänstra punkten `p` och med längden `d` pixlar. Proceduren `fyll` ska, med många horisontella linjer, rita en fylld kvadrat med övre vänstra hörnet i punkten `p` och sidan `s` pixlar. Det som ritas ut ska se ut som den övre bilden till höger. Om du t.ex. tar med en pixel för mycket i dina koordinatberäkningar kan det bli som i den felaktiga undre bilden.



```

1 > scala-cli repl --dep se.lth.cs::introprog:1.3.1
2 scala> val w = new introprog.PixelWindow(400,300,"Tre nyanser av brunt")
3 scala> type Pt = (Int, Int)
4 scala> var color = java.awt.Color.red
5 scala> def rak(p: Pt)(d: Int) = w.line(p._1, p._2, ???, ???, color)
6 scala> def fyll(p: Pt)(s: Int) = for i <- ??? do rak((p._1, ???))(s)
7
8 scala> object Color:
9   |   ???
10
11 scala> color = Color.soil
12 scala> fyll(100,100)(75)
13
14 scala> color = Color.tunnel
15 scala> fyll(100,100)(50)
16
17 scala> color = Color.mole
18 scala> fyll(150,150)(25)

```

c) Vid vilka anrop ovan utnyttjas att tupelparenteserna kan skippas?

**Uppgift 9. Händelser.** På veckans laboration ska du implementera ett enkelt spel där användaren kan styra en blockmullvad med tangentbordet. Med `introprog.PixelWindow` kan du hantera de händelser som genereras när användaren trycker ner eller släpper en tangent eller en musknapp.

a) Studera dokumentationen för singelobjektet `introprog.PixelWindow.Event`. Vad heter den oföränderliga heltalsvariabel som representerar att en nedtryckning av en tangentbordsknapp har inträffat? Vad har variabeln för värde?

b) Via dokumentationen för av singelobjektet `introprog.examples.TestPixelWindow` kan du komma åt koden som implementerar objektet genom att klicka på länken `Source` ovanför sökrutan. Vilken rad i huvudprogrammet i `main`-metoden tar hand om fallet att en knappnedtryckningshändelse har inträffat?

c) Kör med `scala-cli run .` (där punkten står för aktuell katalog) huvudprogrammet i `TestPixelWindow` med optionerna

```

--main-class introprog.examples.TestPixelWindow och
--dep se.lth.cs::introprog:1.3.1

```

Ett testfönster öppnas när `main`-metoden körs. Klicka i fönstret på olika ställen och tryck på olika tangenter och observera vad som skrivs ut. Vad skrivs ut när pil-upp-tangenten trycks ned och släpps upp?

d) Med inspiration från implementationen av `TestPixelWindow`, skriv ett program som ritar gröna linjer mellan positionerna för varje musknapp-nedtryck och musknapp-uppsläpp som användaren gör.

*Tips:* När musknappen trycks ned så spara undan positionen i en variabel med namnet `start`. När musknappen släpps upp, rita linjen från den sparade positionen till `w.lastMousePos`.

### 4.2.2 Extrauppgifter; träna mer

**Uppgift 10.** *Funktioner är objekt med en apply-metod.*

Metoden apply är speciell.

```
1 scala> object plus { def apply(x: Int, y: Int) = x + y }
2 scala> plus.apply(42, 43)
```

Går det att utelämna .apply och anropa plus som en funktion?

**Uppgift 11.** *Skapa moduler med hjälp av singelobjekt.*

- Undersök i REPL vad uttrycket "päronisglass".split('i') har för värde.
- Vad skrivs ut om du med Test() anropar apply-metoden nedan?

```
object stringUtils:
  object split:
    def sentences(s: String): Array[String] = s.split('.')
    def words(s: String): Array[String] = s.split(' ').filter(_.nonEmpty)

  object count:
    def letters(s: String): Int = s.count(_.isLetter)
    def words(s: String): Int = split.words(s).size
    def sentences(s: String): Int = split.sentences(s).size

  object statistics:
    var history = ""
    def printFreq(s: String = history): Unit =
      println(s"\n--- FREKVENSPANALYS AV:\n\${s}")
      println(s"# bokstäver: \${count.letters(s)}")
      println(s"# ord      : \${count.words(s)}")
      println(s"# meningar : \${count.sentences(s)}")
      history = (s"\$history \${s}").trim

object Test:
  import stringUtils.*
  def apply(): Unit =
    val s1 = "Fem myror är fler än fyra elefanter. Ät gurka."
    val s2 = "Galaxer i mina braxer. Tomat är gott. Päronsplitt."
    statistics.printFreq(s1)
    statistics.printFreq(s2)
    statistics.printFreq()
```

- Vilket av objekten i modulen stringUtils har tillstånd? Är det förändringsbart?
- Ändra metoderna i singelobjektet count så att de blir extensionsmetoder och kan anropas så här:

```
scala> import stringUtils.count

scala> val s = "Hejsan hoppsan. Gurka är gott."
val s: String = Hejsan hoppsan. Gurka är gott.

scala> (s.nbrOfLetters, s.nbrOfWords, s.nbrOfSentences)
val res0: (Int, Int, Int) = (24,5,2)
```

**Uppgift 12.** *Tupler som parametrar.* Implementera nedan olika varianter av beräkning av avståndet mellan två punkter. *Tips:* Använd math.hypot.

```
def distxy(x1: Int, y1: Int, x2: Int, y2: Int): Double = ???
```



```
def distpt(p1: (Int, Int), p2: (Int, Int)): Double = ???
def distp(p1: (Int, Int))(p2: (Int, Int)): Double = ???
```

**Uppgift 13.** *Tupler som funktionsresultat.* Tupler möjliggör att en funktion kan returnera flera olika värden på samma gång. Implementera funktionen `statistics` nedan. Den ska returnera en 3-tupel som innehåller antalet element i `xs`, medelvärdet av elementen, samt en 2-tupel med variationsvidden (*min*, *max*). Ange returtypen explicit i din implementation. Testa så att den fungerar i REPL. *Tips:* Du har nytta av metoderna `size`, `sum`, `min` och `max` som fungerar på nummersekvenser.

```
/** Returns the size, the mean, and the range of xs */
def statistics(xs: Vector[Double]) = ???
```

**Uppgift 14.** *Skapa moduler med hjälp av paket.*

a) Koden nedan ligger i filen `paket.scala`. Rita en bild av katalogstrukturen som skapas i aktuellt bibliotek i underkatalogen `main` i `.scala-build` när nedan kod kompileras med: `scala-cli compile paket.scala`

```
package gurka.tomat.banan

package p1:
  package p11:
    object hello:
      def hello = println("Hej paket p1.p11!")
  package p12:
    object hello:
      def hello = println("Hej paket p1.p12!")

package p2:
  package p21:
    object hello:
      def hello = println("Hej paket p2.p21!")

object Main:
  def main(args: Array[String]): Unit =
    import p1.*
    p11.hello.hello
    p12.hello.hello
    import p2.{p21 as apelsin}
    apelsin.hello.hello
```

- b) Vad skrivs ut när programmet körs?
- c) Får paket ha tillståndsvariabler utan att de placeras inuti ett singelobjekt eller en klass?

### 4.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 15.** Hur klara sig utan **do while** i Scala 3? I många språk finns en konstruktion med följande syntax: **do** <satser> **while** <villkor> där <satser> görs minst en gång innan sanningsvärdet för <villkor> testas. Denna ”bakvända while” används inte så ofta, men kan vara smidig om man vill köra en repetition minst en gång.

Denna konstruktion finns i Scala 2 men inte i Scala 3 eftersom nyckelordet **do** i Scala 3 används vid valfria klammerparenteser och indenteringssyntax i ”vanliga while”. Ett skäl att det kan anses ok att ta bort **do** <satser> **while** <villkor> är att en ”bakvänd while” ändå i Scala 3 går att skriva om till en ”vanlig while” genom att inkludera satserna som ska göras minst en gång i ett block på villkorets plats och låta satserna i loopen vara tomma värdet, alltså:

```
while
  <satser>
  <villkor>
do ()
```

a) Nedan funkar i Scala 2, men vad händer om du försöker göra detta i Scala 3:

```
> scala-cli repl --scala 2
Welcome to Scala 2.13.8 (OpenJDK 64-Bit Server VM, Java 17.0.3).
Type in expressions for evaluation. Or try :help.

scala> var i = 0
var i: Int = 0

scala> do i += 1 while (i < 10)

scala> i
val res20: Int = 10
```

b) Skriv om ”bakvända” **do while** till en motsvarande ”vanlig” **while do** som fungerar i Scala 3.

**Uppgift 16.** Postfixa operatorer för inkrementering och dekrementering. I många språk, t.ex. Java, C++, C, går det att skriva **i++** och **i--** om man vill räkna upp eller ner heltalsvariabeln **i**. Använd Scalas extensionsmetoder för att göra så att det går att använda operatorerna **++** och **--** på heltal, enligt nedan:

```
scala> 42.++
val res0: Int = 43

scala> 42.--
val res1: Int = 41

scala> import language.postfixOps    // tillåter postfix operatornotation

scala> 43 ++
val res2: Int = 44

scala> 43 --
val res3: Int = 42

scala> val i = 42
val i: Int = 42

scala> i++
val res4: Int = 43
```

```
scala> i--  
val res5: Int = 41
```

**Uppgift 17.** *Använda färdigt paket: Färgväljare.* På laborationen har du nytta av att kunna blanda egna färger så att du kan rita klarblå himmel och frodigt gräs. Du kan skapa en färgväljare med hjälp av introprog-paketet enligt nedan.

```
1 > scala-cli repl --dep se.lth.cs::introprog:1.3.1  
2 scala> introprog.Dialog.selectColor()
```

- a) Vad händer om du trycker **Ok** efter att du valt en grön färg?
- b) Vad händer om du trycker **Cancel** ?
- c) Vad händer om du trycker **Reset** ?
- d) Läs dokumentationen för metoden `selectColor` i singelobjektet `Dialog` i paketet `introprog`. Anropa `selectColor` med default-färgen `java.awt.Color.green`.

**Uppgift 18.** *Använda färdigt paket: användardialoger.*

- a) Läs om dokumentationen för singelobjektet `Dialog` i paketet `introprog`.
- b) Använd proceduren `introprog.Dialog.show` och ge ett meddelande till användaren att det är "Game over!".
- c) Använd funktionen `introprog.Dialog.input` för att visa frågan "Vad heter du?" och ta reda på användarens namn. Vad händer om användaren klickar *Cancel*?
- d) Använd funktionen `introprog.Dialog.select` för att be användaren välja mellan sten, sax och påse. Vad är returtypen?

★ **Uppgift 19.** *Skapa din egen jar-fil.*

- a) Skriv kommandot `jar` i terminalen och undersök med `jar --help` vad det finns för optioner. Vilka optioner ska du använda för skapa (eng. *create*) en jar i en namngiven fil (eng. *file*) med utförlig (eng. *verbose*) utskrift om vad som händer?
- b) Skapa med en editor i filen `hello.scala` ett enkelt program som skriver ut "Hello package!" eller liknande. Koden ska ligga i paketet `hello` och innehålla ett object `Main` med en `main`-metod. Kompilera din fil med optionen `--destination .` så att din kod hamnar i aktuell katalog i stället för i `.scala-build`.
- c) Skriv ett jar-kommando i terminalen som förpackar koden i en jar-fil med namnet `my.jar` och kör igång REPL med jar-filen på `classpath`. Anropa din `main`-funktion i REPL genom att ange sökvägen `paketnamn.objektnamn.metodnamn` med en tom array som argument.
- d) Med vilket kommando kan du köra det kompilerade och jar-förpackade programmet direkt i terminalen (alltså utan att dra igång REPL)?

★ **Uppgift 20.** *Hur stor är JDK8?* Ta med hjälp av <http://stackoverflow.com/> reda på hur många klasser och paket det finns i Java-plattformen JDK8.

## 4.3 Laboration: blockmole

### Mål

- ☐ Kunna förklara hur singelobjekt kan användas som moduler.
- ☐ Kunna förklara hur åtkomst av medlemmar i singelobjekt sker.
- ☐ Kunna skapa kod som reagerar på och förändrar objekts tillstånd.
- ☐ Kunna förklara nyttan med att samla namngivna konstanter i egen modul.
- ☐ Kunna förklara hur import påverkar synlighet av namn.
- ☐ Kunna ge exempel på en situation där man har nytta av namnbyte vid import.
- ☐ Kunna redogöra för skillnaden mellan paket och singelobjekt.
- ☐ Kunna skapa och använda tupler.
- ☐ Kunna skapa och använda uppdelade parameterlistor.

### Förberedelser

- ☐ Gör övning objects och repetera övning functions.
- ☐ Repetera appendix B, C, och D.
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).

### 4.3.1 Bakgrund



**Blockmullvad** (*Talpa laterculus*) är ett fantasidjur i familjen mullvadsdjur. Den är känd för sitt karaktäristiska kvadratiske utseende. Den lever mest ensam i sina underjordiska gångar som, till skillnad från den verkliga mullvadens (*Talpa europaea*) gångar, har helt raka väggar.

### 4.3.2 Obligatoriska uppgifter


**Uppgift 1.** *Skapa katalog och kodfil.* Du ska, steg för steg, skapa ett program som låter användaren interagera med en levande blockmullvad. Använd en editor, t.ex. VS code, kompilera ditt program i terminalen med `scala-cli compile . -watch` och kör i annat terminalfönster med `scala-cli run .`

a) Skapa en ny fil med namnet `blockmole.scala` i en ny katalog i din hemkatalog, till exempel `~/pgk/w04/lab/blockmole.scala`, där `~` är din hemkatalog.

```
> mkdir -p ~/pgk/w04/lab
> code ~/pgk/w04/lab/blockmole.scala
```

b) Navigera till din nya katalog och kontrollera att din nya fil finns där.

```
> cd ~/pgk/w04/lab/
> ls
blockmole.scala
```

- c) Gör en paketdeklaration i början av filen `blockmole.scala` så att koden du ska skriva nedan ingår i paketet `blockmole`.
- d) Deklarera sedan ett singelobjekt med namnet `Main` med en `main`-procedur som skriver ut texten: "Keep on digging!"
- e)  Kompilera ditt program. Kontrollera med `ls`-kommandot att några filer som slutar på `class` har skapats i subkatalogen `blockmole`. Varför hamnade bytekoden i denna katalog?
- f) Kör kommandot `scala blockmole.Main` för att exekvera ditt program och kontrollera utskriften i terminalfönstret.

Nu har du skrivit ett program som uppmanar en blockmullvad att fortsätta gräva. Det programmet är inte så användbart, eftersom mullvadar inte kan läsa. Nästa steg är därför att skriva ett grafiskt program.

**Uppgift 2.** *Skapa en grundstruktur för programmet.* I mindre program fungerar det bra att samla alla funktioner i ett singelobjekt, men i stora program blir det lättare att hitta i koden och förstå vad den gör om man har flera moduler med olika ansvar. Ditt program ska ha följande övergripande struktur:

```
package blockmole

object Color:
  // Skapar olika färger som behövs i övriga moduler

object BlockWindow:
  // Har ett introprog.PixelWindow och ritar blockgrafik

object Mole: // Representerar en blockmullvad som kan gräva
  def dig(): Unit = println("Här ska det grävas!")

object Main:
  def drawWorld(): Unit = println("Ska rita ut underjorden!")

  @main def run =
    drawWorld()
    Mole.dig()
```

Skapa programskelettet ovan i filen `blockmole.scala` och se till att koden kompilerar utan fel och går att köra med utskrifter som förväntat.

Vi lägger i denna laboration alla moduler i samma fil, men i andra situationer när modulerna blir stora och/eller ska återanvändas av flera olika program är det bra att ha dem i olika filer så att de kan kompileras och testas separat.

**Uppgift 3.** *Lägg till färger i färgmodulen.* I singelobjektet `Color` ska vi skapa färger med hjälp av Java-klassen `java.awt.Color`. Eftersom vårt singelobjektnamn "krockar" med namnet på Java-färgklassen så byter vi namn på Java-klassen till `JColor` i importdeklarationen.

- a) Lägg in en importdeklaration med namnbytet direkt efter paketdeklarationen. Vi lägger importen så att den syns i hela paketet eftersom flera objekt behöver tillgång till `JColor`. Säkerställ att koden fortfarande kompilerar utan fel.

b) Skapa sedan nedan färger i objektet Color:

**object** Color:

```
val black = new JColor( 0, 0, 0)
val mole = new JColor( 51, 51, 0)
val soil = new JColor(153, 102, 51)
val tunnel = new JColor(204, 153, 102)
val grass = new JColor( 25, 130, 35)
```

**Uppgift 4.** Skapa ett ritfönster i modulen för blockgrafik. Lägg till nedan tre variabler i singelobjektet BlockWindow:

```
val windowSize = (30, 50) // (width, height) in number of blocks
val blockSize = 10 // number of pixels per block

val window = new PixelWindow(???, ???, ???)
```

- Importera `introprog.PixelWindow` lokalt i `BlockWindow`. (En lokal import-deklaration är bra här eftersom det bara är detta objekt som behöver tillgång till `PixelWindow`.)
- Gör så att storleken på `window` motsvarar blockstorleken gånger bredd resp. höjd i `windowSize`.
- Ge fönstret en lämplig titel, t.ex. "Digging Blockmole".
- När du kompilerar behöver du se till att `introprog` finns tillgänglig på classpath (se övning objects).
- Om du glömt ordningen på parametrarna till klassen `PixelWindow` så kolla i dokumentationen för `PixelWindow` <sup>5</sup>. Använd namngivna argument vid skapandet av fönstret. Tycker du att koden blir mer läsbar med namngivna argument? <sup>6</sup>

För att testa fönstret, lägg till en enkel testritning genom att i proceduren `drawWorld` använda `BlockWindow.window`, till exempel:

```
def drawWorld(): Unit =
  BlockWindow.window.line(100, 10, 200, 20)
```

Kompilera och kör och säkerställ att allt fungerar som förväntat.

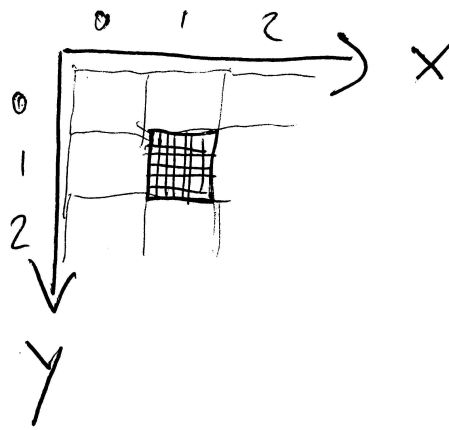
**Uppgift 5.** Skapa procedur för blockgrafik. Nu har du gjort ett grafiskt program, men ännu syns ingen mullvad. Det är dags att skapa koordinatsystemet i blockmullvadens blockvärld.



a) Säkerställ att du kan förklara hur koordinaterna i ett `PixelWindow` tolkas, genom att med papper och penna rita en enkel skiss av ungefär var positionerna (0,0), (300,0), (0,300) och (300,300) ligger i ett fönster som är 300 bildpunkter brett och 300 bildpunkter högt. Använd figur 4.1 för att förklara relationen mellan underliggande fönsterkoordinater och blockkoordinater. Notera att y-axeln pekar nedåt.

<sup>5</sup><http://cs.lth.se/pgk/api/>

<sup>6</sup>Det går tyvärr inte att använda namngivna argument när man instansierar Java-klasser i Scala, men `PixelWindow` är implementerad i Scala så här fungerar det fint.



**Figur 4.1:** Varje block består av många pixlar. Det markerade blocket har koordinat (1,1) i blockkoordinater medan blockets översta vänstra pixel har koordinat (7,7) i PixelWindow-koordinater, om det t.ex. går sju-gång-sju pixlar per block. Vad är block-koordinaten för blocket till höger om det markerade blocket i bilden? Vad är dess PixelWindow-koordinater för översta vänstra och nedersta högra pixlarna?

b) Koordinatsystem i BlockWindow ska ha kvadratiska, *stora* bildpunkter som består av många fönsterpixlar. Vi kallar dessa stora bildpunkter för *block* för att lättare skilja dem från de enpixelstora bildpunkterna i PixelWindow.

I block-koordinatsystemet för BlockWindow gäller följande:

*Blockstorleken* anger sidan i kvadraten för ett block räknat i antalet pixlar. Om blockstorleken är  $b$ , så ligger koordinaten  $(x,y)$  i BlockWindow på koordinaten  $(bx,by)$  i PixelWindow.

Implementera funktionen `block` i modulen `BlockWindow` enligt nedan, så att en kvadrat ritas ut när proceduren anropas. Parametern `pos` anger block-koordinaten och parametern `color` anger färgen. Typ-alias-deklarationen av `Pos` ger ett beskrivande typnamn för en 2-tupel av heltal, som vi kan använda i parameterlistor för att betecknande positioner i ett `BlockWindow`. Se dokumentationen av `fill`-metoden i `PixelWindow`. Observera att du behöver räkna om block-koordinaterna i `pos` till fönsterkoordinater i `windows.fill`. Fyll i det som saknas nedan.

```
type Pos = (Int, Int)
```

```
def block(pos: Pos)(color: JColor = JColor.gray): Unit =
    val x = ??? //räkna ut blockets x-koordinat i pixelfönstret
    val y = ??? //räkna ut blockets y-koordinat i pixelfönstret
    window.fill(???)
```

Säkerställ att koden kompilerar utan fel.

c) För att testa din procedur, anropa funktionen `BlockWindow.block` några gånger i `Main.drawWorld`, dels med utelämnat defaultargument, dels med olika färger ur färgmodulen. Kompilera och kör ditt program och kontrollera att allt fungerar som det ska.

**Uppgift 6.** Skapa *rektangelprocedur* och *underjorden*. Du ska nu skriva en procedur med namnet `rectangle` som ritas en rektangel med hjälp av proceduren `block`. Sen ska du



använda `rectangle` i `Main.drawWorld` för att rita upp mullvadens underjordiska värld.



a) Lägg till proceduren `rectangle` i grafikmodulen. Procedurhuvudet ska ha följande parametrar uppdelade i tre olika paramterlistor, samt returtyp `Unit`:

```
(leftTop: Pos)(size: (Int, Int))(color: JColor = JColor.gray)
```

Parametern `leftTop` anger blockkoordinaten för rektangelns övre vänstra hörn, och `size` anger (bredd, höjd) uttryckt i antal block.

Använd denna nästlade repetition för att rita ut rektangeln:

```
for y <- ??? do
  for x <- ??? do
    block(x, y)(color)
```

-  b) I vilken ordning ritas blocken i rektangeln ut (lodrätt eller vågrätt)? Om du är osäker kan du lägga in en utskrift av `(x, y)` i den innersta loopen för att se ordningen.
-  c) En annan lösning är att i stället anropa `fill`-metoden i `PixelWindow` direkt för att rita en motsvarande stor rektangel *utan* nästlad loop. Vilka argument ska `fill`-anropet då ha?
- d) Lägg följande kod i `Main.drawWorld` så att programmet ritat ut underjorden (det vill säga en massa jord där blockmullvaden kan gräva sina tunnlar) och även lite gräs.

```
def drawWorld(): Unit =
  BlockWindow.rectangle(0, 0)(size = (30, 4))(Color.grass)
  BlockWindow.rectangle(0, 4)(size = (30, 46))(Color.soil)
```

e) Anropa `Main.drawWorld` i `Main.main` och testa att det fungerar. Om någon del av fönstret förblir svart istället för att få gräsfärg eller jordfärg, kontrollera att `block` och `rectangle` är korrekt implementerade.

**Uppgift 7.** I `PixelWindow` finns funktioner för att känna av tangenttryckningar och musklick. Du ska använda de funktionerna för att styra en blockmullvad. Studera dokumentationen för `awaitEvent` och `Event` i `PixelWindow`, samt koden i exempelprogrammet `TestPixelWindow` i paketet `introprog.examples`.

a) Lägg till denna funktion i `BlockWindow`:

```
val maxWaitMillis = 10

def waitForKey(): String =
  window.awaitEvent(maxWaitMillis)
  while window.lastEventType != PixelWindow.Event.KeyPressed do
    window.awaitEvent(maxWaitMillis) // skip other events
  println(s"KeyPressed: ${window.lastKey}")
  window.lastKey
```

Det finns olika sorters händelser som ett `PixelWindow` kan reagera på, till exempel tangenttryckningar och musklick. Funktionen som du precis lagt in väntar på en händelse i ditt `PixelWindow` med hjälp av `(window.awaitEvent)` ända tills det kommer en tangenttryckning (`KEY_EVENT`). När det kommit en tangenttryckning anropas `window.lastKey` för att ta reda på vilken bokstav eller vilket tecken det blev, och det resultatet blir också resultatet av `waitForKey`, eftersom det ligger sist i blocket.



b) Utöka proceduren `Mole.dig` enligt nedan:

```
def dig(): Unit =
  var x = BlockWindow.windowSize._1 / 2
  var y = BlockWindow.windowSize._2 / 2
  var quit = false
  while !quit do
    BlockWindow.block(x, y)(Color.mole)
    val key = BlockWindow.waitForKey()
    if key == "w" then ???
    else if key == "a" then ???
    else if key == "s" then ???
    else if key == "d" then ???
    else if key == "q" then quit = true
  end while
```

c) Fyll i alla ??? så att 'w' styr mullvaden ett steg uppåt, 'a' ett steg åt vänster, 's' ett steg nedåt och 'd' ett steg åt höger.

d) Kontrollera så att `main` bara innehåller två anrop: ett till `drawWorld` och ett till `dig`. Kompilera och kör ditt program för att se om programmet reagerar på tangenterna w, a, s och d.

e) Om programmet fungerar kommer det bli många mullvadar som tillsammans bildar en lång mask, och det är ju lite underligt. Lägg till ett anrop i `Mole.dig` som ritar ut en bit tunnel på position (x,y) efter anropet till `BlockWindow.waitForKey` men innan `if`-satserna. Kompilera och kör ditt program för att gräva tunnlar med din blockmullvad.

### 4.3.3 Kontrollfrågor

✓ 👁 Repetera teorin för denna vecka och var beredd på att kunna svara på dessa frågor när det blir din tur att redovisa vad du gjort under laborationen:

1. Till vad används *classpath*?
2. Vad är en jar-fil?
3. Vad innebär punktnotation?
4. Ge exempel på användning av `import` och förklara vad som händer.
5. Vad är fördelen med skuggning och lokala namn?
6. Vi använde flera singelobjekt som olika s.k. moduler i denna laboration. Vad är fördelen med att dela upp koden i moduler?
7. Gå igenom målen med laborationen och kontrollera så du har uppfyllt dem.

### 4.3.4 Frivilliga extrauppgifter

**Uppgift 8.** Mullvaden kan för tillfället gräva sig utanför fönstret. Lägg till några `if`-satser i början av `while`-satsen som upptäcker om x eller y ligger utanför fönstrets kant och flyttar i så fall tillbaka mullvaden precis innanför kanten.

**Uppgift 9.** Mullvadar är inte så intresserade av livet ovanför jord, men det kan vara trevligt att se hur långt ner mullvaden grävt sig. Lägg till en himmelsfärg i objektet `Color` och rita ut himmel ovanför gräset i `Mole.drawWorld`. Justera också det du gjorde i föregående uppgift,

så att mullvaden håller sig på marken. *Tips:* Du har nytta av en interaktiv färgväljare som du kan få genom att anropa `introprog.Dialog.selectColor()` i Scala REPL.

**Uppgift 10.** Ändra så att mullvaden inte lämnar någon tunnel efter sig när den springer på gräset.

**Uppgift 11.** Låt mullvaden fortsätta gräva även om man inte trycker ned någon tangent. Tangenttryckning ska ändra riktningen.

a) Skapa en ny metod `BlockWindow.waitForKeyNonBlocking` som möjliggör tangentbords-avläsning som ej blockerar exekveringen enligt nedan:

```
def waitForKeyNonBlocking(): String =
  import PixelWindow.Event.{KeyPressed, Undefined}

  window.awaitEvent(maxWaitMillis)
  while
    window.lastEventType != KeyPressed &&
    window.lastEventType != Undefined)
  do window.awaitEvent(maxWaitMillis)
  if window.lastEventType == KeyPressed then window.lastKey else ""
```

b) Lägg till en ny metod `BlockWindow.delay` som ska göra det möjligt att hindra blockmullvaden från att springa alltför fort:

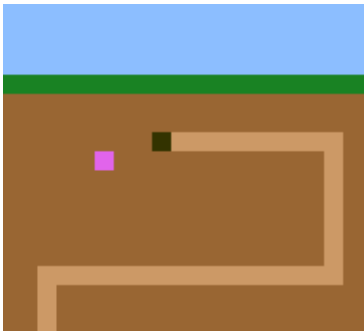
```
def delay(millis: Int): Unit = Thread.sleep(millis)
```

c) Skapa en ny metod `Mole.keepOnDigging` som från början är en kopia av metoden `dig`. Gör följande tillägg/ändringar:

1. Lägg till två variabler **var** `dx` och **var** `dy` i början, som ska hålla reda på riktningen som blockmullvaden gräver. Initialisera dem till 0 respektive 1.
2. Lägg in en fördröjning på 200 millisekunder i den oändliga loopen. Deklarera en konstant `delayMillis` på lämpligt ställe i `Mole` och använd denna konstant som argument till `delay`.
3. Anropa `waitForKeyNonBlocking` i stället för `waitForKey` och kolla efter knapptryckning enligt nedan kodskelett. Fyll i de saknade delarna så att blockmullvaden rör sig ett steg i rätt riktning i varje looprunda.

```
if key == "w" then { dy = -1; dx = 0 }
else if key == "a" then { ??? }
else if key == "s" then { ??? }
else if key == "d" then { ??? }
else if key == "q" then { quit = true }
y += ???
x += ???
```

**Uppgift 12.** Fånga blockmasken.



**Blockmask** (*Lumbricus quadratus*) är ett fantasidjur i familjen daggmaskar. Den är känd för att kunna teleportera sig från en plats till en annan på ett ögonblick och är därför svårfångad. Den har i likhet med den verkliga daggmasken (*Lumbricus terrestris*) RGB-färgen (225,100,235), men är kvadratisk och exakt ett block stor. Blockmasken är ett eftertraktat villebråd bland blockmullvadar.

- a) Lägg till modulen Worm nedan i din kod och använd procedurerna i keepOnDigging så att blockmullvaden får en blockmask att jaga.

```
object Worm:
  import BlockWindow.Pos

  def nextRandomPos(): Pos =
    import scala.util.Random.nextInt
    val x = nextInt(BlockWindow.windowSize._1)
    val y = nextInt(BlockWindow.windowSize._2 - 7) + 7
    (x, y)

  var pos = nextRandomPos()

  def isHere(p: Pos): Boolean = pos == p

  def draw(): Unit = BlockWindow.block(pos)(Color.worm)

  def erase(): Unit = BlockWindow.block(pos)(Color.soil)

  val teleportProbability = 0.02

  def randomTeleport(notHere: Pos): Unit =
    if math.random() < Worm.teleportProbability then
      erase()
      while
        pos = nextRandomPos()
        pos == notHere
      do ()
      draw()

end Worm
```

- b) Koden i Worm förutsätter att himmel finns i fönstrets översta 7 block. Hur många block som är himmel kan egentligen med fördel vara en konstant med ett bra namn på en bra plats. Denna konstant bör användas även i drawWorld. Fixa det!
- c) Gör så att texten "WORM CAUGHT!" skrivs ut i terminalen om blockmullvaden är på samma plats som blockmasken.

- d) Använd parametern `notHere` till att förhindra att blockmasken teleporterar sig till samma plats som blockmullvaden.
- e) Gör så att blockmullvaden får 1000 poäng varje gång den fångar blockmasken.
- f) Gör så att spelet varar en bestämd, lagom lång tid, innan `Game Over`. Använd `System.currentTime` som ger aktuella antalet millisekunder sedan den förste januari 1970. När spelet är slut ska den totala poängen som blockmullvaden samlat skrivas ut i terminalen.
- g) Gör så att spelets hastighet ökar (d.v.s. att fördröjningen i spel-loopen minskar) efter en viss tid. I samband med det ska sannolikheten för att blockmasken teleporterar sig öka.

## Kapitel 5

# Klasser och datamodellering

Begrepp som ingår i denna veckas studier:

- ☐ applikationsdomän
- ☐ datamodell
- ☐ objektorientering
- ☐ klass
- ☐ instans
- ☐ Any
- ☐ instanceof
- ☐ toString
- ☐ new
- ☐ null
- ☐ this
- ☐ accessregler
- ☐ private
- ☐ private[this]
- ☐ klassparameter
- ☐ primär konstruktor
- ☐ fabriksmetod
- ☐ alternativ konstruktor
- ☐ förändringsbar
- ☐ oföränderlig
- ☐ case-klass
- ☐ kompanjonsobjekt
- ☐ referenslikhet
- ☐ innehållslikhet
- ☐ eq
- ☐ ==

## 5.1 Teori

Begreppet **klass** är en viktig abstraktionsmekanism inom **objekt-orienterad programmering** (OOP) för att modellera data i en applikationsdomän, t.ex. data om *användare* och deras *favoritmusik* i applikationsdomänen *musikspelare*. Klasser används för att samla funktioner och data. En klass har ett namn och kan ha parametrar. En klass deklarerar med nyckelordet **class** och är en beskrivning hur en viss typ av objekt ska utformas när de så småningom skapas. Det går att skapa **många** objekt ur en och samma klass.

### 5.1.1 En metafor för klass: Stämpel

En klass liknar en **stämpel**.



- En stämpel kan **tillverkas** – motsvarar **deklaration** av klassen.
- Det händer inget förrän man **stämp-lar** – motsvarar **instansiering**.
- Då skapas **avbildningar** av stämpeln – motsvarar **allokering av ett objekt** som är en **instans** av klassen.
- Allokering kallas också **konstruk-tion** och funktionen/koden som gör själva allokeringen kallas **konstruk-tor**.

---

### 5.1.2 Vad är en klass?

- En klass är en mall (eng. *template*) för att skapa objekt.
- Objekt kan skapas med **new** Klassnamn(parametrar), vilket kallas **instansiering**.
- I Scala 3 är **new** valfritt, det räcker med Klassnamn(parametrar).
- Ett objekt som skapats med klassen Klassnamn som mall kallas för en **instans** av klassen Klassnamn.
- En klass innehåller **medlemmar** (eng. *members*), som bl.a. kan vara:
  - **attribut**, kallas även fält (eng. *field*): **val**, **lazy val**, **var**
  - **metoder**, kallas även operationer: **def**
- Varje instans har sin **egen** uppsättning värden på attributen, som tillsammans utgör instansens **tillstånd**.

---

### 5.1.3 Datamodellering

Varför behövs klasser?

- I en viss **applikationsdomän** (eng. *application domain*), tex. skatteverkets deklara-tionssystem, behövs en **modell av domänspecifik data**, t.ex. personer, personnummer, adresser, inkomster, avdrag, fastigheter, etc.
- Med klasser kan du skapa **nya** typer (utöver `Int`, `String` ...) som bättre representerar domänens data.
- Med klasser implementerar du modeller som representerar väsentliga **attribut** ur applikationsdomänen.
- Med **metoder** (funktioner i klasser) kan du skapa och behandla domänens data.
- Datamodellering i Scala görs ofta med **case**-klasser och **oföränderliga** instanser.

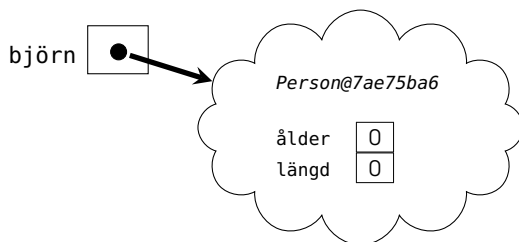
#### 5.1.4 Singelobjekt jämfört med klass

Vi har tidigare deklarerat **singelobjekt** som bara finns i **en** enda upplaga:

```
scala> object Björn { var ålder = 54; val längd = 178 }
```

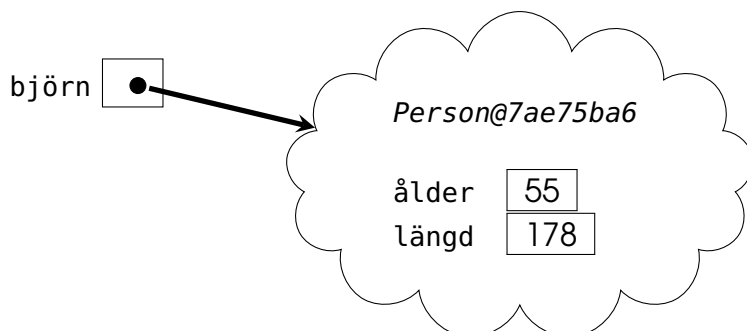
Med en **klass** kan man skapa **godtyckligt många instanser av klassen** med hjälp av nyckelordet **new** följt av klassens namn:

```
scala> class Person { var ålder = 0; var längd = 0 }  
  
scala> val björn = new Person // allokera plats i minnet  
björn: Person = Person@7ae75ba6 // unikt id för instansen
```



#### 5.1.5 Förändring av objektets tillstånd

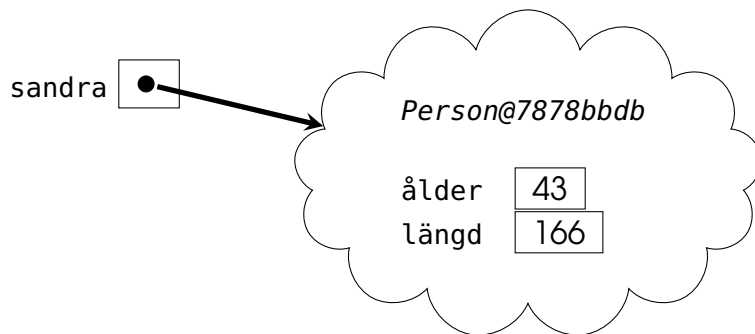
```
scala> björn.ålder = 55  
  
scala> björn.längd = 178
```



#### 5.1.6 Bättre att initialisera med hjälp av klassparametrar

```
scala> class Person(var ålder: Int, var längd: Int)

scala> val sandra = new Person(43, 166)
sandra: Person = Person@7878bbdb
```



### 5.1.7 KlassdeklARATIONER och instansIERING

- Syntax för deklaration av klass:  
**class** Klassnamn(parametrar){ medlemmar }
- Exempel: **deklaration**

```
class Klassnamn(val attribut1: Int, attribut2: String): //klassparametrar
  val attribut3: Double = 42.0 //publikt oföränderligt attribut
  private var attribut4: Boolean = false //privat medlem syns inte utåt
  def metod(parameter: Int) = attribut1 + 1 //funktion i objekt kallas metod
  lazy val attr5 = Vector.fill(100000)(42.0) //fördröjd initialisering
```

- Parametrar initialiseras med de argument som ges vid **new**.
- Exempel: **instansiering** med argument för initialisering av klassparametrar

```
val instansReferens = new Klassnamn(42, "hej") // new är valfritt i Scala 3
```

- Parametrar som inte föregås av modifierare (t.ex. **private val**, **val**, **var**) blir **attribut** som bara är synliga i **denna** instans.
- Attribut i klasskroppen är **publika** (alltså synliga utåt) om de inte är **private** (eller **protected** som begränsar synlighet till subtyper som vi ska se senare).

### 5.1.8 ÖVNING: en klass som representerar en person

1. Deklarera en klass Person med dessa publika attribut:
  - oföränderligt förnamn
  - oföränderligt efternamn
  - förändringsbar ålder med defaultargument 0
2. lägg till en metod i klasskroppen med explicit returtyp som ger en 2-tupel med förnamn och efternamn
3. skriv en deklaration som deklarerar en variabel p som initialiseras med värdet av ett uttryck som instansierar klassen Person med ditt namn och din ålder som nyfödd.



4. skriv en sats som skriver ut ditt förnamn genom att referera attribut med punktnotation
5. skriv en tilldelningssats som ändrar tillståndet för den instans som referensen p refererar till så att åldersattributets värde blir din nuvarande ålder

---

### 5.1.9 Lösning: klassen Person

```
class Person(  
  val givenName: String,  
  val familyName: String,  
  var age: Int = 0  
):  
  def name: (String, String) = (givenName, familyName)
```

```
scala> val p = Person("Björn", "Regnell")  
val p: Person = Person@783dc0e7  
  
scala> println(p.name._1)  
Björn  
  
scala> p.age = 50
```

Kan vi få se något som är finare än Person@783dc0e7 ?

---

### 5.1.10 Skapa egen najs toString

```
class Person(  
  val givenName: String,  
  val familyName: String,  
  var age: Int = 0  
):  
  def name: (String, String) = (givenName, familyName)  
  override def toString = "najs toString"
```

```
scala> val p = Person("Björn", "Regnell")  
val p: Person = najs toString  
  
scala> println(p.name._1)  
Björn  
  
scala> p.age = 55
```

Vad vill du se i stället för "najs toString"?

Övning: Visa instansens tillstånd med stränginterpolatorn s"?"

---

### 5.1.11 Instansprivata klassparametrar

- Parametrar som **inte** föregås av någon modifierare alls (t.ex. **val**, **var** etc.) blir medlemmar som är bara är synliga i **denna** instans.
- Exempel på konsekvensen av **instansprivata** parametrar:

```

1 scala> class C(a: Int){ def add(other: C): Int = a + other.a }
2 -- Error:
3 1 |class C(a: Int){ def add(other: C): Int = a + other.a }
4   |                                     ^^^^^^^
5   | value a cannot be accessed as a member of (other : C) from class C.
```

- Men detta fungerar fint:

```

1 scala> class D(private val a: Int){ def add(other: D): Int = a + other.a }
2
3 scala> D(42).add(D(43))
4 res0: Int = 85
```

...eftersom modifieraren **private val** ger en medlem som "bara" är **klassprivat** och ger därmed synlighet i **alla** D-instanser (men bara där; medlemmen är inte ens synlig i subtyper till D).

### 5.1.12 Case-klasser är som vanliga klasser med extra godis

Med **case** framför **class** får du en massa **godis** på köpet, bland annat detta:

- En najs `toString`-metod med klassens namn och dess attributvärden.

```

scala> case class Person(name: String, age: Int)

scala> val p = Person("Björn", 55)

scala> p.toString
val res0: String = Person(Björn,55)
```

- Parameter till case-klass blir automatiskt ett **publikt oföränderligt attribut**, alltså en **val**-medlem utan att du behöver skriva något.

```

scala> p.age
val res1: Int = 55
```

- En `copy`-metod med alla attribut som parametrar och instansens attributvärden som default-argument. Den gör det smidigt att skapa förändrade kopior där några attribut ändrats och andra förblir som innan.

```

scala> p.copy(age = p.age + 1)
val res2: Person = Person(Björn,56)
```

### 5.1.13 Fördjupning: Styra synlighet med `private[X]`

Med hjälp av `private[X]` kan du begränsa synlighet till `X`, där `X` kan vara ett singelobjekt, en typ eller ett paket:

```
scala> object X:
  |   object Y { private[X] var y = 42 }
  |   def visaHemlis = Y.y // y syns i X
  |
// defined object X

scala> X.Y.y
-- Error:
1 |X.Y.y
  |^^^^
  |variable y cannot be accessed as a member of X.Y.type from module class rs$line$26.

scala> X.visaHemlis
val res0: Int = 42
```

### 5.1.14 Styra användningen av infix alfanumeriska operatorer

Någon gång i **framtiden** kommer följande gälla:

- Metoder som har **alfanumeriska namn**, alltså namn med bokstäver och ev. siffror ger en **varning** vid operatornotation om de **inte** är deklarerade med nyckelordet **infix**.

```
case class Box(x: Int): // kompilera med optionen "-source:future"
  def +(other: Box): Box = Box(x + other.x) // utan varning
  def plus(other: Box) = Box(x + other.x) // ger varning (i framtiden)
  infix def add(other: Box) = Box(x + other.x) // utan varning
```

-source:future aktiverar regler för framtida Scala-versioner

-deprecation varnar för sådant som kommer att så småningom tas bort.

```
> scala-cli repl --scala-opt -source:future -deprecation
scala> Box(41) plus Box(1)
-- Warning:
1 |Box(41) plus Box(1)
  |      ^^^^^
  |Alphanumeric method plus is not declared `infix`; it should not be used as infix operator.
  |The operation can be rewritten automatically to `plus` under -deprecation -rewrite.
  |Or rewrite to method syntax .plus(...) manually.
val res0: Box = Box(42)
```

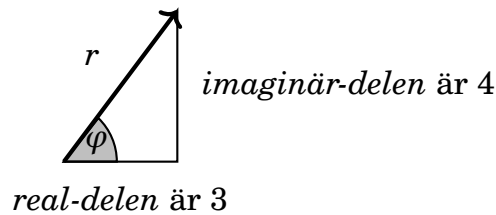
### 5.1.15 Övning: Klassen Complex

Implementera klassen Complex nedan som representerar komplexa tal:

```
class Complex(val re: Double, val im: Double):
  def r = ??? // absolutbeloppet
  def fi = ??? // vinkeln i radianer
  def +(other: Complex): Complex = ??? // resultatet av addition
  var imSymbol = 'i' // symbol för imaginärdel, används i toString
```

```
override def toString = ??? // en strängrepresentation av talet
```

Exempel:  
 $z = 3 + 4i$



### 5.1.16 Exempel: Klassen Complex

```
class Complex(val re: Double, val im: Double):
  def r = math.hypot(re, im)
  def fi = math.atan2(im, re) // motstående sida först
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

```
1 scala> val z1 = new Complex(3, 4) // konstruktion av instans av Complex
2 z: Complex = 3.0 + 4.0i
3
4 scala> val polärForm = (z1.r, z1.fi)
5 polärForm: (Double, Double) = (5.0,0.6435011087932844)
6
7 scala> val z2 = Complex(1, 2) // new behövs inte i Scala 3
8 z2: Complex = 1.0 + 2.0i
9
10 scala> z1 + z2
11 res0: Complex = 4.0 + 6.0i
```

<https://scala-lang.org/api/3.x/scala/math.html#atan2-44b>

### 5.1.17 Exempel: Principen om enhetlig access

```
class Complex(val re: Double, val im: Double):
  val r = math.hypot(re, im)
  val fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

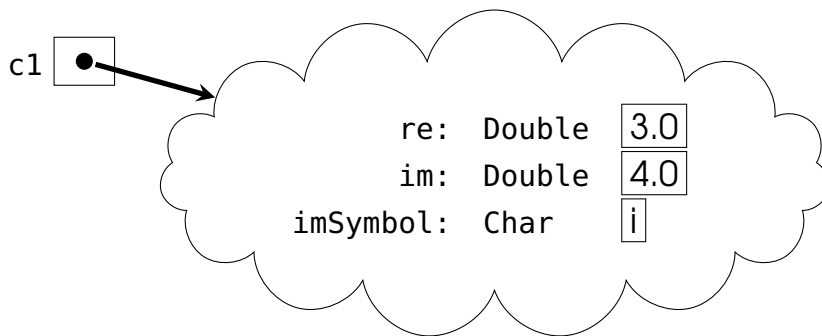
- Efter som attributen `re` och `im` är oföränderliga, kan vi lika gärna ändra i klass-implementationen och göra om metoderna `r` och `fi` till **val**-variabler utan att klientkoden påverkas.
- Då anropas `math.hypot` och `math.atan2` bara en gång vid initialisering (och inte varje gång som med **def**).
- Vi skulle även kunna använda **lazy val** och då bara räkna ut `r` och `fi` om och när de verkligen refereras av klientkoden, annars inte.

- Eftersom klientkoden inte ser skillnad på metoder och variabler, kallas detta **principen om enhetlig access**. (Många andra språk har **inte** denna möjlighet, tex Java där metoder *måste* ha parenteser.)

### 5.1.18 Instansiering med direkt användning av new

Instansiering genom **direkt användning** av **new**  
(här första varianten av Complex med r och fi som metoder)

```
scala> val c1 = new Complex(3, 4)
```



Ofta vill man göra **indirekt** instansiering så att vi senare har friheten att ändra hur instansiering sker.

### 5.1.19 Indirekt instansiering med fabriksmetoder

En **fabriksmetod** är en metod som används för att instansiera objekt.

```
object MyFactory {
  def createComplex(re: Double, im: Double) = new Complex(re, im)
  def createReal(re: Double)                = new Complex(re, 0)
  def createImaginary(im: Double)           = new Complex(0, im)
}
```

Instansiera **inte direkt**, utan **indirekt** genom användning av **fabriksmetoder**:

```
1 scala> import MyFactory.*
2
3 scala> createComplex(3, 4)
4 res0: Complex = 3.0 + 4.0i
5
6 scala> createReal(42)
7 res1: Complex = 42.0 + 0.0i
8
9 scala> createImaginary(-1)
10 res2: Complex = 0.0 + -1.0i
```

### 5.1.20 Hur förhindra direkt instansiering?

Om vi vill **förhindra direkt instansiering** kan vi göra primärkonstruktorn **privat**:

```
class Complex private (val re: Double, val im: Double):
  def r = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

MEN... då går det ju **inte** längre att instansiera något alls! : (

```
scala> new Complex(3,4)
error:
  constructor Complex in class Complex cannot be accessed
```

### 5.1.21 Kompanjonsobjekt med indirekt instansiering

- Ett **kompanjonsobjekt** (eng. *companion object*) är ett singelobjekt som ligger i **samma kodfil** som en klass, och som har **samma namn** som klassen.
- Medlemmar i ett kompanjonsobjekt **får accessa privata** medlemmar i kompanjonsklassen (och vice versa) och kompanjonsobjektet får därför accessa privat konstruktor och kan göra **new**.
- Fabriksmetod + privat konstruktor: tillåt **enbart indirekt instansiering**.

```
class Complex private (val re: Double, val im: Double):
  def r = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"

object Complex:
  def apply(re: Double, im: Double) = new Complex(re, im) // new behövs här
  def real(re: Double) = new Complex(re, 0)
  def imag(im: Double) = new Complex(0, im)
```

- **new** behövs för att förhindra rekursivt anrop av apply och stack overflow

### 5.1.22 Användning av kompanjonsobjekt med fabriksmetoder

Nu kan vi **bara** instansiera **indirekt**! : )

```
scala> Complex.real(42.0)
res0: Complex = 42.0 + 0.0i

scala> Complex.imag(-1)
res1: Complex = 0.0 + -1.0i

scala> Complex.apply(3,4)
res2: Complex = 3.0 + 4.0i
```

```
scala> Complex(3,4)
res3: Complex = 3.0 + 4.0i

scala> new Complex(3, 4)
error:
  constructor Complex in class Complex cannot be accessed
```

### 5.1.23 Alternativa direktinstansieringar med default-argument

Med **default-argument** kan vi erbjuda **alternativa** sätt att direktinstansiera.

```
class Complex(val re: Double = 0, val im: Double = 0):
  def r = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

```
1 scala> new Complex()
2 res0: Complex = 0.0 + 0.0i
3
4 scala> new Complex(re = 42) //anrop med namngivet argument
5 res1: Complex = 42.0 + 0.0i
6
7 scala> new Complex(im = -1)
8 res2: Complex = 0.0 + -1.0i
9
10 scala> new Complex(1)
11 res3: Complex = 1.0 + 0.0i
```

### 5.1.24 Alternativa sätt att instansiera med fabriksmetod

Vi kan också erbjuda **alternativa** sätt att instansiera **indirekt** med fabriksmetoden `apply` i ett kompanjonsobjekt genom default-argument:

```
class Complex private (val re: Double, val im: Double):
  def r = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"

object Complex:
  def apply(re: Double = 0, im: Double = 0) = new Complex(re, im)
  def real(r: Double) = apply(re = r)
  def imag(i: Double) = apply(im = i)
  val zero = apply()
```

### 5.1.25 Medlemmar som bara behövs i en enda upplaga

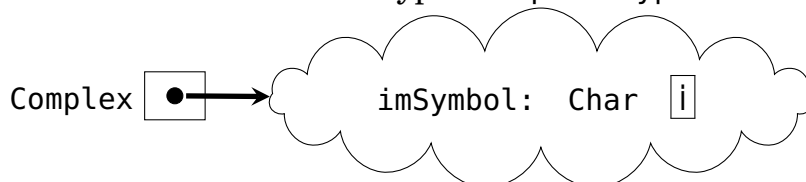
Attributet `imSymbol` passar bättre att ha i **kompanjonsobjektet**, eftersom det räcker att ha **en enda upplaga**, som kan vara gemensam för alla objekt:

```
class Complex private (val re: Double, val im: Double):
  def r = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  override def toString = s"$re + $im${Complex.imSymbol}"

object Complex:
  var imSymbol = 'i'
  def apply(re: Double = 0, im: Double = 0) = new Complex(re, im)
  def real(r: Double) = apply(re = r)
  def imag(i: Double) = apply(im = i)
  val zero = apply()
```

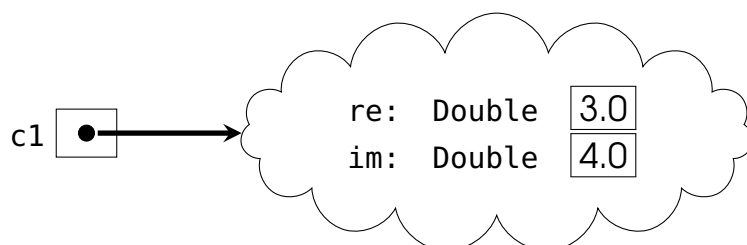
### 5.1.26 Medlemmar i singelobjekt är statiskt allokerade

Minnesplatsen för **attribut i singelobjekt** allokeras automatiskt en gång för alla, och kallas därför **statiskt** allokerad. Singelobjektets namn `Complex` utgör en statisk referens till den enda instansen och är av typen `Complex.type`.



Nu bereder vi inte plats för `imSymbol` i varenda **dynamiskt** allokerade instans:

```
scala> val c1 = Complex(3, 4)
```



### 5.1.27 Attribut i kompanjonsobjekt användas för sådant som är gemensamt för alla instanser

Om vi ändrar på statiska `imSymbol` så ändras `toString` för **alla** dynamiskt allokerade instanser.



```
scala> val c1 = Complex(3, 4)
c1: Complex = 3.0 + 4.0i

scala> Complex.imSymbol = 'j'
Complex.imSymbol: Char = j

scala> val c2 = Complex(5, 6)
c2: Complex = 5.0 + 6.0j

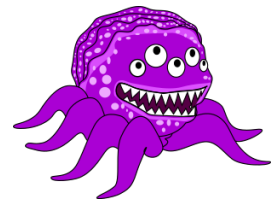
scala> c1
res0: Complex = 3.0 + 4.0j
```

### 5.1.28 Övning: en läskig mutant

1. Skapa en klass med namnet `Mutant` som har ett förändringsbart attribut som klassparameter med namnet `i` av typen `Int` med default-argumentet 5.

Deklarera två **val**-variabler som kallas `fem1`

2. och `fem2` och som båda refererar till **samma** `Mutant`-instans.



En `Mutant`-instans där `i` kanske är fem.

3. Skriv kod som ändrar tillstånd via den ena mutantreferensen.
4. Syns ändringen via den andra mutantreferensen?

### 5.1.29 Case-klasser

Case-klasser är ett smidigt sätt att skapa **oföränderliga** datastrukturer. Med nyckelordet **case** framför **class** får du mycket ”godis på köpet”:

- Klassparametrar blir automatiskt publika<sup>1</sup> oföränderliga attribut och du slipper alltså skriva **val**.
- Du får en automatisk **toString** med klassens namn och värdet av alla **val**-attribut som ges av klassparametrarna
- och en **copy**-metod för att skapa nya, delvis förändrade instanser, med attributvärdena som defaultargument.
- Du får ett automatiskt kompanjonsobjekt med en fabriksmetod `apply` för indirekt instansiering där alla klassparametrarnas **val**-attribut initialiseras.
- ... och mer därtill men mer om det senare...

---

<sup>1</sup>alltså **inte** instansprivata som i vanliga klasser.

### 5.1.30 Exempel: oföränderliga case-klassen Point

```
case class Point(x: Double, y: Double)
```

```
scala> val p1 = Point(3, 4)
p1: Point = Point(3.0,4.0)

scala> val p2 = p1
p2: Point = Point(3.0,4.0)

scala> p1.x = 42
error: reassignment to val
```

Vi kan utan risk dela med oss av en referens till en oföränderlig klass – ingen kan ändra dess innehåll. (Jämför läskiga mutanten i tidigare exempel.)

### 5.1.31 Vad är en konstruktor?

- En **konstruktor** är den maskinkod som exekveras när klasser instansieras med **new**.
- Konstruktorn skapar ett nytt objekt i minnet vid varje anrop.
- I Scala **genererar kompilatorn** en **primärkonstruktor** åt dig med maskinkod som initialiserar alla attribut baserat på klassparametrarna som du deklarerat.
- I Scala **kan** man också skriva egna alternativa s.k. **hjälpkonstruktorer**, men det är **ovanligt**, eftersom man har möjligheten med fabriksmetoder i kompanjonsobjekt och default-argument.

### 5.1.32 Hjälpkonstruktorer i Scala (ovanliga)

Fördjupning för kännedom:

- I Scala kan man skapa ett alternativ till primärkonstruktorn, en så kallad **hjälpkonstruktor** (eng. *auxilliary constructor*) genom att deklarera en metod med det speciella namnet **this**.
- Hjälpkonstruktorer **måste** börja med att anropa en **annan** konstruktor som står **före** i koden, till exempel primärkonstruktorn.

```
class Point(val x: Int, val y: Int, val z: Int): // primärkonstruktor
  def this(x: Int, y: Int) = this(x, y, 0) // anropa primärkonstruktorn
  def this(x: Int) = this(x, 0) // anropa hjälpkonstruktor
```

- Varför? Enklare att använda från **Java**-kod jämfört med apply i kompanjonsobjekt. (Men om din Scala-kod inte ska användas från Java så är detta onödigt.)

### 5.1.33 Användning av hjälpkonstruktor

```
1 scala> val p1 = Point(1)
2 p1: Point = Point@21312342
3
4 scala> val p2 = Point(1, 2)
5 p2: Point = Point@43254325
6
7 scala> val p3 = Point(1, 2, 3)
8 p3: Point = Point@346654
```

Men man gör **mycket oftare** så här i Scala:

```
case class Point(x: Int, y: Int = 0, z: Int = 0)
```

Använd alltså defaultargument hellre än hjälpkonstruktor.  
(Eller överlagrad fabriksmetod i kompanjonsobjekt.)

---

### 5.1.34 Referens saknas: null

- I Java och många andra språk använder man ofta nyckelordet **null** för att representera att ett **värde saknas**.
  - En referens som är **null** refererar inte till någon instans.
  - Om du försöker referera till instansmedlemmar med punktnotation genom en referens som är **null** kastas ett **undantag** `NullPointerException`.
  - Oförsiktig användning av **null** är en vanlig källa till **buggar**, som kan vara svåra att hitta och fixa.
- 

### 5.1.35 Exempel: null

```
1 scala> class Gurka(val vikt: Int)
2
3 scala> var g: Gurka = null           // ingen instans allokerad än
4 var g: Gurka = null
5
6 scala> g.vikt
7 java.lang.NullPointerException
8
9 scala> g = Gurka(42)                 // instansen allokeras
10 g: Gurka = Gurka@1ec7d8b3
11
12 scala> g.vikt
13 val res0: Int = 42
14
15 scala> g = null                     // instansen kommer att destrueras av skräpsamlaren
```

- Scala har **null** av kompatibilitetsskäl, men det är brukligt att **endast** använda **null** om man anropar Java-kod.
  - Scala erbjuder smidiga `Option`, `Some` och `None` för säker hantering av saknade värden; mer om detta kommande vecka.
-

### 5.1.36 Defaultvärden under pågående konstruktion

Int	0
Double	0.0
Float	0.0F
Long	0L
Short	0.toShort
Byte	0.toByte
Char	0.toChar
Boolean	false
Alla referenstyper, tex. String	<b>null</b>

### 5.1.37 Problem med initialisering av attribut vid konstruktion

```

class InitBug1:
  val HEJ = hej.toUpperCase
  val hej = "hej"

class InitBug2:
  val b = a
  val a = 10

class InitBug3:
  val hej2 = hej1
  val hej1 = "hej"

```

```

1 scala> val ib1 = new InitBug1
2 scala> val ib2 = new InitBug2
3 scala> ib2.b
4 scala> val ib3 = new InitBug3
5 scala> ib3.hej2

```

Vad händer?

### 5.1.38 Vilka värden har attribut medan konstruktion pågår?

```

class InitBug1:
  val HEJ = hej.toUpperCase
  val hej = "hej"

class InitBug2:
  var b = a
  var a = 10

```

```
class InitBug3:
  val hej2 = hej1
  val hej1 = "hej"
```

```
1 scala> val ib1 = new InitBug1 // java.lang.NullPointerException
2 scala> val ib2 = new InitBug2
3 scala> ib2.b // val res0: Int = 0 // WHAT???
4 scala> val ib3 = new InitBug3
5 scala> ib3.hej2 // val res1: String = null //WHAT???
```

Varför? Vad finns det för lösningar?

### 5.1.39 Hur undvika initialiseringsproblem vid konstruktion?

Några tips för att undvika initialiseringsproblem av attribut:

- Ändra om möjligt ordningen på attribut-deklarationer
- Använd om möjligt i stället **lazy val** (init sker senare)
- Använd om möjligt i stället **def** (evaluering vid varje anrop)
- Använd denna kompilatoroption för att få hjälp med varningar vid risk för initialiseringsproblem: `-Ysafe-init`

Om du **verkligen** behöver ha ett oinitialiserat värde:

```
class Box:
  private var x: String = scala.compiletime.uninitialized // tydliggör null-risk
  def get: String = if x != null then x else "" // glöm ej kolla null
  def getOrElse(alt: String): String = if x != null then x else alt
  def set(value: String): Unit = x = value
```

Försök att **undvika null** om det går eftersom det ger stor risk för buggar!

(I ovan fiktiva exempel hade vi kunnat undvika detta enkelt genom att ge x startvärdet "" i stället för null. En sådan lösning förutsätter att det finns en rimlig representation av ett saknat värde. Mer om hantering av saknade värden senare...)

### 5.1.40 Referensen this

- Nyckelordet `this` ger en referens till den aktuella instansen.

```
scala> class Gurka(var vikt: Int){def jagSjälvt = this}

scala> val g = Gurka(42)
val g: Gurka = Gurka@5ae9a829

scala> g.jagSjälvt
val res0: Gurka = Gurka@5ae9a829

scala> g.jagSjälvt.vikt
val res1: Int = 42

scala> g.jagSjälvt.jagSjälvt.vikt
val res2: Int = 42
```

- Referensen `this` används ofta för att komma runt ”namnkrockar” där variabler med samma namn gör så att den ena variabeln inte syns.

#### 5.1.41 Getters och setters

- I många språk (t.ex. Java, Python) finns inget motsvarande nyckelord **val** som garanterar oföränderliga attributreferenser.<sup>2</sup>
- Därför gör man i dessa språk nästan alltid alla attribut **privata** för att förhindra att de ändras på ett okontrollerat sätt.
- Därför är det normalt att införa metoder som kallas **getters** och **setters**, som används för att **indirekt** läsa och uppdatera **attribut**.
- Dessa metoder känns i många språk igen genom konventionen att de heter något som börjar med **get** respektive **set**. (Men **ej** vanligt i Scala.)
- Med **indirekt access** av attribut kan man åstadkomma **flexibilitet**, så att implementationen kan ändras utan att ändra i klientkoden:
  - man kan t.ex. i efterhand ändra representation av de privata attributen eftersom all access sker genom getters och setters.
- Man kan åstadkomma **oföränderliga** datastrukturer där attributreferenserna inte förändras efter allokering om klassen **inte** erbjuder en **setter** för privata attribut.

#### 5.1.42 Java-exempel: Klassen JPerson

**Indirekt** access av **privata** attribut:

```
public class JPerson {
    private String name;
    private int age = 0;

    public JPerson(String name){
        //namnkrock fixas med this
        this.name = name;
    }

    public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }

    public void setAge(int age){
```

```
        this.age = age;
    }
}
```

<sup>2</sup>Java har visserligen **final** men det är annorlunda som vi ska se senare.

```
> scala-cli repl .
scala> val p = JPerson("Björn")
val p: JPerson = JPerson@7e774085

scala> p.getAge
val res0: Int = 0

scala> p.setAge(42)
```

```
scala> p.getAge
val res1: Int = 42

scala> p.age
-- Error:
p.age
^^^^
value age is not a member of JPerson
```

### 5.1.43 Motsvarande JPerson i Scala

Så här brukar man åstadkomma ungefär motsvarande i Scala:

```
class Person(val name: String):
  var age = 0
```

Notera att alla attribut här är **publika**.

### 5.1.44 Förhindra felaktiga attributvärden med setters

Med hjälp av **setters** kan vi förhindra **felaktig** uppdatering av attributvärden, till exempel **negativ ålder** i klassen JPerson i Java:

```
public void setAge(int age){
    if (age >= 0) {
        this.age = age;
    } else {
        this.age = 0;
    }
}
```

Hur kan vi åstadkomma **motsvarande i Scala**?

Antag att vi började med nedan variant, men **ångrar** oss och sedan vill införa funktionalitet som förhindrat negativ ålder **utan att ändra i klientkod**:

```
class Person(val name: String):
  var age = 0
```

Om vi inför en ny metod setAge och gör attributet age privat så funkar det **inte** längre att skriva `p.age = 42` och vi ”kvaddar” klientkoden! :(

### 5.1.45 Getters och setters i Scala

- Principen om **enhetlig access** tillsammans med **specialsyntax** för **setters** kommer till vår räddning!
- En **setter** kan i Scala skapas med **procedur vars namn slutar med \_=**
- I Scala kan man utan att kvadda klientkod införa getter+setter så här:

```
class Person(val name: String): // ändrad implementation men samma access
  private var myPrivateAge = 0
  def age = myPrivateAge        // getter
  def age_=(a: Int): Unit =     // setter
    if a >= 0 then myPrivateAge = a else myPrivateAge = 0
```

```
1 scala> val p = Person("Björn")
2 val p: Person = Person@28ac3dc3
3
4 scala> p.age = 42          // najs syntax om getter parad med setter enl ovan
5 val p.age: Int = 42
6
7 scala> p.age = -1         // nu förhindras negativ ålder
8 val p.age: Int = 0
```

### 5.1.46 Referenslikhet eller innehållslikhet?

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturelikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.
- I Scala finns flera metoder som testar likhet:
  - metoden `eq` testar **referenslikhet** och `r1.eq(r2)` ger **true** om `r1` och `r2` refererar till **samma** instans.
  - metoden `ne` testar referensolikhet och `r1.ne(r2)` ger **true** om `r1` och `r2` refererar till **olika** instanser.
  - metoden `==` som anropar metoden `equals` som default testar referenslikhet med `eq` men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturelikhet.
- Scalas **standardbibliotek** och **grundtyperna** `Int`, `String` etc. testar **innehållslikhet** genom metoden `==`

### 5.1.47 Exempel: referenslikhet och innehållslikhet

I Scalas standardbibliotek har man överskuggat `equals` så att metoden `==` ger test av **innehållslikhet** mellan instanser:



```
1 scala> val v1 = Vector(1,2,3)
2 v1: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
3
4 scala> val v2 = Vector(1,2,3)
5 v2: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
6
7 scala> v1 eq v2 //referenslikhetstest: olika instanser
8 res0: Boolean = false
9
10 scala> v1 ne v2
11 res1: Boolean = true
12
13 scala> v1 == v2 //innehållslikhetstest: samma innehåll
14 res2: Boolean = true
15
16 scala> v1 != v2
17 res3: Boolean = false
```

---

### 5.1.48 Referenslikhet och egna klasser

Om du inte gör något speciellt med dina egna klasser så ger metoden `==` test av **referenslikhet** mellan instanser:

```
scala> class Gurka(val vikt: Int)

scala> val g1 = new Gurka(42)
g1: Gurka = Gurka@2cc61b3b

scala> val g2 = new Gurka(42)
g2: Gurka = Gurka@163df259

scala> g1 == g2 // samma innehåll men olika instanser
res0: Boolean = false

scala> g1.vikt == g2.vikt
res1: Boolean = true
```

---

### 5.1.49 Case-klasser ger innehållslikhet

Förutom annat ”godis på köpet” får du med **case class** även detta:

- Metoden `==` ger **innehållslikhet** (och inte referenslikhet).

---

### 5.1.50 Likhet och case-klasser

Metoden `equals` är i case-klasser automatiskt överskuggad så att metoden `==` ger test av strukturell likhet.

```
1 scala> case class Gurka(vikt: Int)
2
3 scala> val g1 = Gurka(42)
```

```
4 g1: Gurka = Gurka(42)
5
6 scala> val g2 = Gurka(42)
7 g2: Gurka = Gurka(42)
8
9 scala> g1 eq g2          // olika instanser
10 res0: Boolean = false
11
12 scala> g1 == g2          // samma innehåll!
13 res1: Boolean = true
```

---

### 5.1.51 Sammanfattning case-klass-godis

Kom-ihåg-lista med "godis" i **case**-klasser så här långt:

1. klassparametrar blir **val**-attribut
2. najs toString
3. automatisk fabriksmetod apply i kompanjonsobjekt
4. == ger innehållslighet (eng. *structural equality*)
- ...

Men vi har inte sett allt godis än:  
Mönstermatchning (mer om det senare).

---

### 5.1.52 Implementation saknas: ???

- Ofta vill man bygga kod iterativt och steg för steg lägga till olika funktionalitet.
- Standardfunktionen ??? ger vid anrop undantaget **NotImplementedError** och kan användas på platser i koden där man ännu inte är färdig.
- ??? tillåter **kompilering av ofärdig kod**.
- Undantag har bottenotypen Nothing som är subtyp till *alla* typer och kan därmed tilldelas referenser av godtycklig typ.

```
scala> lazy val sprängsSnart: Int = ???

scala> sprängsSnart + 42
scala.NotImplementedError: an implementation is missing
```

---

### 5.1.53 Exempel: ofärdig kod

```
case class Person(name: String, age: Int):
  def ärTonåring = age >= 13 && age <= 19
  def ärUng = !ärGammal
  def ärGammal: Boolean = ???    //implementation ännu ej klar
```

```
scala> Person("Björn", 51).ärTonåring
res23: Boolean = false

scala> Person("Sandra", 39).ärUng
scala.NotImplementedError: an implementation is missing
```

---

## 5.2 Övning classes

### Mål

- ☐ Kunna deklarerera klasser med klassparametrar.
- ☐ Kunna skapa instanser med och utan **new**.
- ☐ Kunna ge argument vid instansiering.
- ☐ Förstå innebörden av referensvariabler och värdet **null**.
- ☐ Kunna använda nyckelordet **private** för att styra synlighet av attribut och konstruktorparametrar.
- ☐ Förstå syftet med getters och setters.
- ☐ Kunna förklara accessregler för kompanjonsobjekt.
- ☐ Kunna skapa fabriksmetod i kompanjonsobjekt.
- ☐ Känna till nyttan med en privat konstruktor.
- ☐ Förstå skillnaden mellan referenslikhet och strukturlikhet.
- ☐ Känna till skillnaden mellan `==` och `eq`, samt `!=` och `ne`.
- ☐ Kunna förklara hur case-klasser hanterar instansiering.
- ☐ Känna till hur case-klasser hanterar likhet.

### Förberedelser

- ☐ Studera begreppen i kapitel 5

#### 5.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** *Para ihop begrepp med beskrivning.*

Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

klass	1	A	indirekt tilldelning av attributvärde
instans	2	B	instanser anses olika även om tillstånden är lika
konstruktor	3	C	nyckelord vid direkt instansiering av klass
klassparameter	4	D	ser privata medlemmar i klass med samma namn
referenslikhet	5	E	hjälpfunktion för indirekt konstruktion
innehållslikhet	6	F	slipper skriva new; automatisk innehållslikhet
case-klass	7	G	ett värde som ej refererar till någon instans
getter	8	H	en mall för att skapa flera instanser av samma typ
setter	9	I	upplaga av ett objekt med eget tillståndsminne
kompanjonsobjekt	10	J	instanser anses lika om de har samma tillstånd
fabriksmetod	11	K	binds till argument som ges vid konstruktion
<b>null</b>	12	L	indirekt åtkomst av attributvärde
<b>new</b>	13	M	skapar instans, allokerar plats för tillståndsminne

**Uppgift 2.** *Klass och instans.* Du har i övning objects sett hur singelobjekt i en egen namnrymd kan samla funktioner (metoder) och ha tillstånd (attribut). Men singelobjekt finns bara i en upplaga. Vill du kunna skapa många objekt av samma typ behöver du en

*klass*. En objektupplaga som skapats ur en klass kallas en *instans* av klassen. Varje instans har sitt eget tillstånd. Deklarera singelobjektet och klassen nedan och klistra in i REPL.

```
object Singelpunkt { var x = 1; var y = 2 }
class Punkt       { var x = 3; var y = 2 }
```

a) Antag att uttrycken till vänster evalueras uppifrån och ned. Vilket resultat till höger hör ihop med respektive uttryck? Prova i REPL om du är osäker.<sup>3</sup>

Singelpunkt.x	1	A	java.lang.NullPointerException
Punkt.x	2	B	1
val p = new Singelpunkt	3	C	Not found: type
val p1 = new Punkt	4	D	p1: Punkt = Punkt@27a1a53c
val p2 = Punkt()	5	E	3
{ p1.x = 1; p2.x }	6	F	p2: Punkt = Punkt@51ab04bd
(new Punkt).y	7	G	value is not a member of <b>object</b>
{ val p: Punkt = null; p.x }	8	H	2

b) Vid tre tillfällen blir det fel. Varför? Är det kompileringsfel eller exekveringsfel?

**Uppgift 3. Klassparametrar.** Klassen Punkt i föregående uppgift är inte så smidig att använda eftersom man först *efter* instansiering kan ge attributen x och y de koordinatvärden man önskar och detta måste ske med explicita tilldelningssatser.

Detta problem kan du lösa med *klassparametrar* som låter dig initialisera attributen med konstruktionsargument och på så sätt ange ett initialtillstånd direkt i samband med instansiering.

Deklarera klassen nedan i REPL.

```
class Point(var x: Int, var y: Int)
```

a) Antag att uttrycken till vänster evalueras uppifrån och ned. Vilket resultat till höger hör ihop med respektive uttryck? Prova i REPL om du är osäker.

val p1 = Point(1, 2)	1	A	missing argument for parameter
val p2 = Point()	2	B	2
val p2 = Point(3, 4)	3	C	p1: Point = Point@30ef773e
p2.x - p1.x	4	D	too many arguments for constructor
Point(0, 1).y	5	E	p2: Point = Point@218cf600
Point(0, 1, 2)	6	F	1

b) Vid två tillfällen blir det fel. Varför? Är det kompileringsfel eller exekveringsfel?

<sup>3</sup>Strängen efter @-tecknet är en hexadecimal representation av det heltal som tillordnas varje objekt för att systemet ska kunna särskilja olika instanser. <https://stackoverflow.com/questions/4712139>

**Uppgift 4.** *Oföränderlig klass med defaultargument.* Det som gäller för parametrar och argument till funktioner är även tillämpligt på klassparametrar, t.ex. defaultargument och namngivna argument. Man kan *dessutom* framför klassparametrar använda nyckelorden **var** och **val** och då blir parametern ett synligt attribut. Vill man ha privata attribut kan man ange t.ex. **private val** framför klassparameternamnet. Om inget anges framför en klassparameter är det den allra mest restriktiva synligheten **private[this] val** som gäller, vilket innebär att namnet bara syns i den aktuella instansen<sup>4</sup>.

Deklarera nedan klass i REPL.

```
class Point3D(val x: Int = 0, val y: Int = 0, z: Int = 0)
```

a) Antag att uttrycken till vänster evalueras uppifrån och ned. Vilket resultat till höger hör ihop med respektive uttryck? Prova i REPL om du är osäker.

<b>val</b> p1 = Point3D()	1	A	<b>false</b>
<b>val</b> p2 = Point3D(y = 1)	2	B	Reassignment to <b>val</b>
Point3D(z = 2).z	3	C	p1: Point3D = Point3D@2eb37eee
p2.y = 0	4	D	<b>true</b>
p2.y == 0	5	E	value cannot be accessed
p1.x == Point3D().x	6	F	p2: Point3D = Point3D@65a9e8d7

b) Vad är problemet med ovan klass om man vill använda den för att representera punkter i 3 dimensioner?

**Uppgift 5.** *Case-klass, this, likhet, toString och kompanjonsobjekt.*  
Klistra in nedan klasser i REPL.

```
case class Pt(x: Int = 0, y: Int = 0):
  def moved(dx: Int = 0, dy: Int = 0): Pt = Pt(x + dx, y + dy)

class MutablePt(private var p: (Int, Int) = (0, 0)):
  def x: Int = p._1
  def y: Int = p._2
  def move(dx: Int = 0, dy: Int = 0) = { p = (x + dx, y + dy); this }
  override def toString = s"MPt($x,$y)"
```

a) Antag att uttrycken till vänster evalueras uppifrån och ned. Vilket REPL-svar till höger hör ihop med respektive uttryck? Prova i REPL om du är osäker.

<sup>4</sup>För case-klasser, som vi ska se snart, är det i stället **val** medförande synlighet och oföränderlighet som gäller (alltså inte **private[this] val**).

<b>val</b> p1 = Pt(1, 2)	1	A	MPt(5,6)
<b>val</b> p2 = Pt(y = 3)	2	B	<b>false</b>
<b>val</b> p3 = MutablePt(5, 6)	3	C	Pt(0,3)
<b>val</b> p4 = Mutable()	4	D	Not found
p2.moved(dx = 1) == Pt(1, 3)	5	E	Pt(1,2)
p3.move(dy = 1) == MutablePt(5, 7)	6	F	<b>true</b>

- b) Vilken returtyp kommer kompilatorn härleda för funktionen `MutablePt.move`?
- c) Vad är skillnaden mellan instansiering med universella apply-metoder och instansiering med **new**? Finns det något fall där **new** måste användas?
- d) Vad kallas sådana metoder som **def** x och **def** y ovan?

**Uppgift 6.** Implementera delar av klasserna *Pos*, *KeyControl*, *Mole* och *BlockWindow* som behövs under laborationen *blockbattle1*. I nästa laboration ska du bygga vidare på blockmolelabben och göra ett spel för två spelare där varje spelare styr sin *egen* instans av en blockmole. Vi måste då göra om *Mole* så att den blir en klass i stället för ett singelobjekt. Gör färdigt klasserna nedan och testa noggrant så att de fungerar.

Alla klasser ska tillhöra **package** *blockbattle* och ligga i varsin egen fil med samma namn som klassen, t.ex. *Pos.scala*.

Tips: Ha ett separat terminalfönster igång och kör Scala CLI med ändringsbevakning enligt nedan kommando. Då kompileras din ändrade kod om automatiskt varje gång du sparar en scala-fil i aktuell katalog.

```
scala-cli compile . --watch
```

Optionen `--watch` kan skrivas kortare med `-w` i stället.

- a) Under laborationen är det smidigt att kunna representera flyttbara positioner i ett pixelfönster. Implementera case-klassen *Pos* i ett nytt terminalfönster enligt nedan så att den fungerar enligt efterföljande REPL-tester.

```
package blockbattle

case class Pos(x: Int, y: Int):
  def moved(delta: (Int, Int)): Pos = ???
```

Testa så att *Pos* fungerar med hjälp av REPL enligt nedan:

```
1 > scala-cli repl .
2 Welcome to Scala 3.3.0 (17.0.6, Java OpenJDK 64-Bit Server VM).
3 Type in expressions for evaluation. Or try :help.
4
5 scala> blockbattle.Pos(1,2)
6 val res0: blockbattle.Pos = Pos(1,2)
7
8 scala> import blockbattle.*
9
10 scala> val p = Pos(1,2)
11 val p: blockbattle.Pos = Pos(1,2)
12
13 scala> p.moved(0,1)
14 val res1: blockbattle.Pos = Pos(1,3)
```

Testa även att anropa `moved` på klassnamnet, t.ex. `Pos.moved(0,1)`. Fungerar detta? Varför/varför inte? Hur skiljer sig anrop till metoder i singelobjekt respektive klassinstanser?

b) Under laborationen är det smidigt att kunna representera vilka tangenter som motsvarar de olika riktningar som en användare kan styra sin mullvad i. Gör klart case-klassen `KeyControl` enligt nedan så att den fungerar enligt efterföljande REPL-tester. Metoden `direction` ska ge ett delta-steg i rätt (x, y)-riktning för ett givet tangentnamn. Metoden ska ge **true** om tangentnamnet finns i någon av de fyra riktningstangenterna i denna `KeyControl`-instans, annars **false**.

```
package blockbattle

case class KeyControl(left: String, right: String, up: String, down: String):
  def direction(key: String): (Int, Int) = ???

  def has(key: String): Boolean = ???
```

```
1 scala> import blockbattle.*
2
3 scala> val kc1 = KeyControl(right="d",left="a",up="w",down="s")
4 val kc1: blockbattle.KeyControl = KeyControl(a,d,w,s)
5
6 scala> val kc2 = KeyControl("Left","Right","Up","Down")
7 val kc2: blockbattle.KeyControl = KeyControl(Left,Right,Up,Down)
8
9 scala> kc2.left
10 val res0: String = Left
11
12 scala> kc2.has("a")
13 val res1: Boolean = false
14
15 scala> kc2.has("Up")
16 val res2: Boolean = true
17
18 scala> kc1.direction("a")
19 val res3: (Int, Int) = (-1,0)
20
21 scala> kc1.direction("s")
22 val res4: (Int, Int) = (0,1)
23
24 scala> kc1.direction("d")
25 val res5: (Int, Int) = (1,0)
26
27 scala> kc1.direction("w")
28 val res6: (Int, Int) = (0,-1)
29
30 scala> Pos(1,2).moved(kc1.direction("a"))
31 val res7: blockbattle.Pos = Pos(0,2)
```

c) Gör klart klassen `Mole` enligt nedan. `Mole` är en klass som representerar en blockmullvad med föränderliga attribut för position, riktning och poäng. Varje instans har även oföränderliga attribut som håller reda på dess namn, dess färg och vilka tangenter som kan användas för att styra mullvaden. Implementera klassens medlemmar en i taget och testa noga med lämpliga testfall efter varje tillägg/buggfix. Skapa ett huvudprogram t.ex. i filen `Main.scala` med dina tester som skapar instanser och skriver ut attribut etc.

```
package blockbattle

class Mole(
  val name: String,
```



```

var pos: Pos,
var dir: (Int, Int),
val color: java.awt.Color,
val keyControl: KeyControl
):
  var points = 0

  override def toString =
    s"Mole[name=$name, pos=$pos, dir=$dir, points=$points]"

  /** Om keyControl.has(key) så uppdateras riktningen dir enligt keyControl */
  def setDir(key: String): Unit = ???

  /** Uppdaterar dir till motsatta riktningen. */
  def reverseDir(): Unit = ???

  /** Uppdaterar pos så att den blir nextPos */
  def move(): Unit = ???

  /** Ger nästa position enligt riktningen dir utan att uppdatera pos */
  def nextPos: Pos = ???

```

d) Under laborationen behöver du en klass `blockbattle.BlockWindow` som med hjälp av `introprog.PixelWindow` erbjuder blockgrafik. Varje instans av `BlockWindow` ska ha ett attribut som refererar till en `PixelWindow`-instans. Detta kallas **aggregering** (eng. *aggregation*).<sup>5</sup>

För att det ska gå att kompilera och testa din `BlockWindow`-klass behöver du ha `introprog`-paketet på classpath. Ladda ner filen <https://fileadmin.cs.lth.se/introprog.jar> via din webbläsare eller med kommandot `curl` nedan (notera att det är stora bokstaven `O` och inte en nolla i optionen `-sLO`):

```

curl -o introprog.jar -sLO https://fileadmin.cs.lth.se/introprog.jar
scala-cli run . --jar introprog.jar

```

Då hamnar `introprog.jar` automatiskt på classpath.

Gör klart klassen `BlockWindow` enligt nedan. Metoden `setBlock` ska med hjälp av metoden `pixelWindow.fill` fylla ett kvadratisk område med sidan `blockSize` pixlar på en viss position `pos` i block-koordinater och med en viss färg `color`. Metoden `getBlock` ska med hjälp av metoden `pixelWindow.getPixel` ge färgen för övre vänstra hörnet i blocket på position `pos` i block-koordinater.

```

package blockbattle

class BlockWindow(
  val nbrOfBlocks: (Int, Int),
  val title: String = "BLOCK WINDOW",
  val blockSize: Int = 14
):
  import introprog.PixelWindow

  val pixelWindow = new PixelWindow(
    nbrOfBlocks._1 * blockSize, nbrOfBlocks._2 * blockSize, title)

  def setBlock(pos: Pos, color: java.awt.Color): Unit = ???

  def getBlock(pos: Pos): java.awt.Color = ???

  def write(

```

<sup>5</sup>[https://en.wikipedia.org/wiki/Object\\_composition#Aggregation](https://en.wikipedia.org/wiki/Object_composition#Aggregation)

```

text: String,
pos: Pos,
color: java.awt.Color,
textSize: Int = blockSize
): Unit = pixelWindow.drawText(
    text, pos.x * blockSize, pos.y * blockSize, color, textSize)

def nextEvent(maxWaitMillis: Int = 10): BlockWindow.Event.EventType =
    import BlockWindow.Event._
    pixelWindow.awaitEvent(maxWaitMillis)
    pixelWindow.lastEventType match
        case PixelWindow.Event.KeyPressed => KeyPressed(pixelWindow.lastKey)
        case PixelWindow.Event.WindowClosed => WindowClosed
        case _ => Undefined

object BlockWindow:
    def delay(millis: Int): Unit = Thread.sleep(millis)

object Event:
    trait EventType
    case class KeyPressed(key: String) extends EventType
    case object WindowClosed extends EventType
    case object Undefined extends EventType

```

I instruktionerna till laborationen `blockbattle1` finns tips om hur du kan hantera händelser i ett `BlockWindow` med hjälp av metoden `nextEvent` ovan.

e) Gör så att huvudprogrammet i `Main.scala` ritar några valfria block i en instans av klassen `BlockWindow`. Skapa även en **while** (!quit)-loop som med hjälp av `nextEvent()` skriver ut händelser i terminalen som inte är av typen `Undefined`.

Metoden `nextEvent()` ligger i klassen `BlockWindow`. Varje looprunda ska även innehålla en 200 millisekunders fördröjning genom anrop av `delay`-metoden som definierats i kompanjonsobjektet `BlockWindow` ovan. Om händelsen `WindowClosed` inträffar ska loopen avslutas. Kör huvudprogrammet och kontrollera så att resultatet blir som förväntat.

## 5.2.2 Extrauppgifter; träna mer

**Uppgift 7.** *Instansiering med **new** och värdet **null**.* Man skapar instanser av klasser med **new**. Då anropas konstruktorn och plats reserveras i datorns minne för objektet. Variabler av referenstyp som inte refererar till något objekt har värdet **null**.

a) Vad händer nedan? Vilka rader ger felmeddelande och i så fall hur lyder felmeddelandet?

```

1 scala> class Gurka(val vikt: Int)
2 scala> var g: Gurka = null
3 scala> g.vikt
4 scala> g = new Gurka(42)
5 scala> g.vikt
6 scala> g = null
7 scala> g.vikt

```

b) Rita minnessituationen efter raderna 2, 4, 6.

**Uppgift 8.** *Skapa en punktklass som kan hantera polära koordinater.* Du ska skapa en oföränderlig case-klass `Point` som kan representera en koordinat både med "vanliga" kartesiska

koordinater<sup>6</sup> och med polära koordinater<sup>7</sup>.

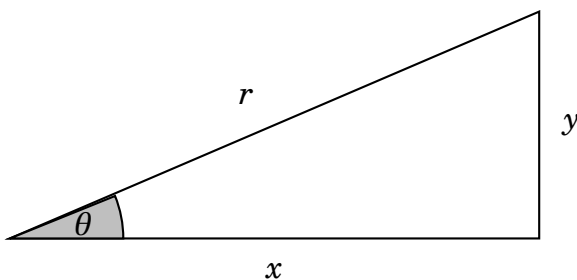
a) Skapa kod med hjälp av en editor, t.ex. VS code, i filen `Point.scala` enligt följande riktlinjer:

1. `Point` ska ligga i paketet `graphics`.
2. `Point` ska ha följande två publika, oföränderliga klassparametrar:
  - `x`: `Double` för x-koordinaten.
  - `y`: `Double` för y-koordinaten.
3. `Point` ska ha följande publika medlemmar (två oföränderliga attribut och två metoder):
  - `val r`: `Double` ska ge motsvarande polära koordinatens avstånd till origo.
  - `val theta`: `Double` ska ge polära koordinatens vinkel i radianer.
  - `def negY`: `Point` ska ge en ny punkt med y-koordinaten negerad.
  - `def +(p: Point)`: `Point` ska ge en ny punkt vars koordinat är summan av x- respektive y-koordinaterna för denna instans och punkten `p`.
4. `Point` ska ha ett kompanjonsobjekt med en metod som konstruerar en punkt från polära koordinater. Metoden ska ha detta huvud:
 

```
def polar(r: Double, theta: Double): Point
```

Tips:

- Du har nytta av metoderna `r = math.hypot(x, y)` och `θ = math.atan2(y, x)` vid omvandling till polära koordinater:



- Du har nytta av metoderna `math.cos(theta)` och `math.sin(theta)` vid omvandling från polära koordinater.
- Attributet `negY` är användbar vid omvandling till fönsterkoordinater där y-axeln är omvänd jämfört med kartesiska koordinater.
- Notera att klassens attribut är av typen `Double` och inte `Int`, trots att vi senare ska använda punkten för att beskriva en diskret pixelposition i ett `PixelWindow`. Anledningen till detta är att det kan uppstå avrundningsfel vid numeriska beräkningar. Detta blir särskilt märkbart vid upprepade räkning med små värden, t.ex. när man ritar en approximerad cirkel med många linjesegment.

b) Använd din punktklass då du deklarerar en funktion `cyclic(n, r, p)` som ger en sekvens av punkter som beskriver en liksidig månghörning med `n` hörn där hörnen är placerade på en cirkel med radien `r` och mittpunkten `p`.

c) Skapa en funktion `drawPolygon(pts)` som ritar månghörningar enligt en punktsekvens `pts` i ett `PixelWindow`.

d) Hur många hörn behövs det för att en liksidig månghörning ska se ut som en cirkel?

<sup>6</sup>[https://sv.wikipedia.org/wiki/Kartesiskt\\_koordinatsystem](https://sv.wikipedia.org/wiki/Kartesiskt_koordinatsystem)

<sup>7</sup>[https://sv.wikipedia.org/wiki/Pol%C3%A4ra\\_koordinater](https://sv.wikipedia.org/wiki/Pol%C3%A4ra_koordinater)

**Uppgift 9.** *Klasser, instanser och skräp.* För länge sedan i en galax långt långt borta...

```
case class Arm(ärTillVänster: Boolean)

case class Ben(ärTillVänster: Boolean)

case class Huvud(harHår: Boolean = true)

case class Rymdvarelse(
  arm1: Arm = Arm(true),
  arm2: Arm = Arm(false),
  ben1: Ben = Ben(true),
  ben2: Ben = Ben(false),
  huvud1: Huvud = Huvud(harHår = false),
  var huvud2: Huvud = Huvud()
):
  def ärSkallig = !huvud1.harHår && !huvud2.harHår
```

a) Klistra in ovan rymdkod i REPL och evaluera nedan rader. Rita minnessituationen efter rad 5 och beskriv vad som händer.

```
1 scala> val alien = Rymdvarelse()
2 scala> alien.ärSkallig
3 scala> val predator = Rymdvarelse()
4 scala> predator.ärSkallig
5 scala> predator.huvud2 = alien.huvud1
6 scala> predator.huvud2 eq alien.huvud1 // test av referenslikhet
7 scala> println(predator)
8 scala> predator.ärSkallig
```

b) Vad händer så småningom med det ursprungliga huvud2-objektet i predator efter tilldelningen på rad 5? Går det att referera till detta objekt på något sätt?

**Uppgift 10.** *Case-klass. Oföränderlig kvadrat.*

a) Implementera nedan kvadrat med en editor och klistra in den i REPL.

```
case class Square(val x: Int = 0, val y: Int = 0, val side: Int = 1):
  val area: Int = ???

  /** Creates a new Square moved to position (x + dx, y + dy) */
  def moved(dx: Int, dy: Int): Square = ???

  def isEqualSizeAs(that: Square): Boolean = ???

  /** Multiplies the side with factor and rounded to nearest integer */
  def scale(factor: Double): Square = ???

object Square:
  /** A Square at (0, 0) with side 1 */
  val unit: Square = ???
```

b) Testa din kvadrat enligt nedan. Förklara vad som händer.

```
1 scala> val (s1, s2) = (Square(), Square(1, 10, 1))
2 scala> val s3 = s1 moved (1,-5)
3 scala> s1 isEqualSizeAs s3
4 scala> s2 isEqualSizeAs s1
5 scala> s1 isEqualSizeAs Square.unit
6 scala> s2.scale(math.Pi) isEqualSizeAs s2
7 scala> s2.scale(math.Pi) isEqualSizeAs s2.scale(math.Pi)
8 scala> s2.scale(math.Pi) eq s2.scale(math.Pi)
9 scala> Square.unit eq Square.unit
```

### 5.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 11.** *Innehållslighet mellan olika typer.* Klistra in nedan klasser i REPL och undersök vad som händer.

```
class Gurka(val vikt: Int)
```

```
class Bil(val typ: String)
```

```
1 scala> class Gurka(val vikt: Int)
2   |
3   | class Bil(val typ: String)
4 // defined class Gurka
5 // defined class Bil
6
7 scala> 42 == "Fyrtiotvå"
8
9 scala> Gurka(50) == Bil("Sedan")
```

Finns det något resultat som är problematiskt, och i så fall, varför?

**Uppgift 12.** *Attributrepresentation. Privat konstruktör. Fabriksmetod.* Kim Kodkunnig skapade för länge sedan denna klass som används på många ställen i befintlig kod:

```
class Point private (val x: Int, val y: Int)
object Point:
  def apply(x: Int = 0, y: Int = 0): Point = new Point(x, y)
  val origo = apply()
```

- Vad händer om du försöker instansiera Kim Kodkunnigs klass direkt med nyckelordet **new**?
- Varför använder Kim Kodkunnig ett kompanjonsobjekt med en fabriksmetod? Vilka accessregler gäller mellan ett kompanjonsobjekt och klassen med samma namn?
- Hjälp Kim Kodkunnig att ändra attributrepresentationen så att det oföränderliga tillståndet utgörs av en 2-tupel **val** p: (Int, Int) i stället. Befintlig kod ska inte behöva ändras och klassen Point ska bete sig från "utsidan" precis som innan.

**Uppgift 13.** *Synlighet av klassparametrar och konstruktör, **private**[this].*

- En av gurk-klasserna nedan är trasig. Varför och vad blir det för fel?

```
class Gurka1(vikt: Int)
```

```
class Gurka2(val vikt: Int)
```

```
class Gurka3(private val vikt: Int)
```

```
class Gurka4(private val vikt: Int, kompis: Gurka4):
  def kompisVikt = kompis.vikt
```

```
class Gurka5(private[this] val vikt: Int, kompis: Gurka5):
  def kompisVikt = kompis.vikt
```

```

class Gurka6 private (vikt: Int)

class Gurka7 private (var vikt: Int)
object Gurka7:
  def apply(vikt: Int) =
    require(vikt >= 0, "negativ vikt: " + vikt)
    new Gurka7(vikt)

```

b) Undersök nedan vad nyckelorden **val** och **private** får för konsekvenser. Förklara vad som händer. Vilka rader ger vilka felmeddelanden?

```

1 scala> new Gurka1(42).vikt
2 scala> new Gurka2(42).vikt
3 scala> new Gurka3(42).vikt
4 scala> val ingenGurka: Gurka4 = null
5 scala> new Gurka4(42, ingenGurka).kompisVikt
6 scala> new Gurka4(42, new Gurka4(84, null)).kompisVikt
7 scala> new Gurka6(42)
8 scala> new Gurka7(-42)
9 scala> Gurka7(-42)
10 scala> val g = Gurka7(42)
11 scala> g.vikt
12 scala> g.vikt = -1
13 scala> g.vikt

```

**Uppgift 14.** *Egendefinierad setter kombinerat med privat konstruktor.* Klistra in denna kod i REPL:

```

class Gurka8 private (private var _vikt: Int):
  def vikt = _vikt
  def vikt_=(v: Int): Unit =
    require(v >= 0, "negativ vikt: " + v)
    _vikt = v

object Gurka8:
  def apply(vikt: Int) =
    require(vikt >= 0, "negativ vikt: " + vikt)
    new Gurka8(vikt)

```

a) Förklara vad som händer nedan. Vilka rader ger vilka felmeddelanden?

```

1 scala> val g = Gurka8(-42)
2 scala> val g = Gurka8(42)
3 scala> g.vikt
4 scala> g.vikt = 0
5 scala> g.vikt = -1
6 scala> g.vikt += 42
7 scala> g.vikt -= 1000

```

b) Vad är fördelen med möjligheten att skapa egendefinierade setters?

**Uppgift 15.** *Objekt med föränderligt tillstånd (eng. mutable state).* Du ska implementera en modell av en hoppande groda som uppfyller följande krav:

1. Varje grodobjekt ska hålla reda på var den är.
2. Varje grodobjekt ska hålla reda på hur långt grodan hoppat totalt.
3. Varje grodobjekt ska kunna beräkna hur långt det är mellan grodans nuvarande position och utgångsläget.
4. Alla grodor börjar sitt hoppande i origo.
5. En groda kan hoppa enligt två metoder:
  - relativ förflyttning enligt parametrarna  $dx$  och  $dy$ ,
  - slumpmässig relativ förflyttning  $[1, 10]$  i x-ledsförändring och  $[1, 10]$  i y-ledsförändring.

a) Implementera klassen Frog enligt nedan kodskelett och ovan krav.

```
class Frog private (initX: Int = 0, initY: Int = 0):
  def x: Int = ???
  def y: Int = ???

  def jump(dx: Int, dy: Int): Unit = ???
  def randomJump: Unit = ???

  def distanceToStart: Double = ???
  def distanceJumped: Double = ???
  def distanceTo(that: Frog): Double = ???

object Frog:
  def spawn(): Frog = ???
```

*Tips:*

- Om namnet man vill ge ett privat föränderligt attribut ”krockar” med ett metodnamn, är det vanligt att man börjar attributets namn med understreck, t.ex. **private var** `_x` för att på så sätt undkomma namnkonflikten.
- Inför en metod i taget och klistra in den nya grodan i REPL efter varje utvidgning och testa.

b) Skapa en metod **def** `test(): Unit` i ett singelobjekt `FrogTest` som innehåller kod som gör minst en kontroll av varje krav. Om ingen kontroll går fel ska "Test Ok!" skrivas ut, annars ska exekveringen avbrytas. *Tips:* Använd `assert(b, msg)` som avbryter exekveringen och skriver ut `msg` om `b` är falsk.

c) Vad kallas en metod som enbart returnerar värdet av ett privat attribut?

d) Inför setters för attributen som håller reda på x- och y-positionen. Förändringar av positionen i x- eller y-led ska räknas som ett hopp och alltså registreras i det attribut som håller reda på det ackumulerade hoppavståndet.

e) Simulera ett massivt grodhoppande med krockdetektering genom att skapa 100 grodor som till att börja med är placerade på x-axeln med avståndet 8 längdenheter mellan sig. För varje runda i en **while**-sats, låt en slumpmässigt vald groda göra ett `randomJump` tills någon groda befinner sig närmare än 0.5 längdenheter, vilket är definitionen på att de har krockat. Räkna hur många looprundor som behövs innan något grodpar krockar och skriv ut antalet. Skriv även ut det totala antalet

*Tips:* Börja med pseudokod på papper. Använd en grodvektor.



**Uppgift 16.** Objekt med föränderligt tillstånd (eng. *mutable state*). Webbshoppen **UberSquare** säljer flyttbara kvadrater. I affärsmodellen ingår att ta betalt per förflyttning. Du ska hjälpa UberSquare att utveckla en enkel prototyp för att imponera på riskkapitalister. (En variant av denna uppgift ingick i tentamen 2017-08-23.)

a) Implementera Square enligt dokumentationskommentarerna i efterföljande kodskiss och enligt dessa krav:

1. Varje instans av Square ska räkna antalet förflyttningar som gjorts sedan instansen konstruerats.
2. För att kunna övervaka sina kunder vill UberSquare även räkna det totala antalet förflyttningar som gjorts av alla kvadrater som någonsin skapats (s.k. *big data*).
3. Varje gång förflyttning sker ska ett visst belopp adderas till den ackumulerade kostnaden för respektive kvadrat, enligt kostnadsberäkningen i krav 4.
4. UberSquare är oroliga för att kvadraterna flyttas för långt bort och bestämmer därför att för varje förflyttning ska den ackumulerade kvadratkostnaden ökas med den nya positionens avstånd till ursprungsläget vid kvadratens konstruktion multiplicerat med aktuell storlek på kvadraten.
5. För att framstå som goda berättar UberSquare i sin marknadsföring att det är gratis att skala kvadrater.<sup>8</sup>

```
/** A mutable and expensive Square. */
class Square private (val initX: Int, val initY: Int, val initSide: Int):
  private var nMoves = 0
  private var sumCost = 0.0

  private var _x = initX
  private var _y = initY

  private var _side = initSide

  private def addCost(): Unit =
    sumCost += ???

  def x: Int = ???
  def y: Int = ???

  def side = ???

  /** Scales the side of this square and rounds it to nearest integer */
  def scale(factor: Double): Unit = ???

  /** Moves this square to position (x + xd, y + dy) */
  def move(dx: Int, dy: Int): Unit = ???

  /** Moves this square to position (x, y) */
  def moveTo(x: Int, y: Int): Unit = ???

  /** The accumulated cost of this Square */
  def cost: Double = ???

  /** Returns the accumulated cost. Sets the accumulated cost to zero. */
  def pay: Double = ???
```

<sup>8</sup>D.v.s. ett anrop av metoden scale orsakar ingen omedelbar kostnad.

```

override def toString: String =
  s"Square[($x, $y), side: $side, #moves: $nMoves times, cost: $sumCost]"

object Square:
  private var created = Vector[Square]()

  /** Constructs a new Square object at (x, y) with size side */
  def apply(x: Int, y: Int, side: Int): Square =
    require(side >= 0, s"side must be positive: $side")
    ???

  /** Constructs a new Square object at (0, 0) with side 1 */
  def apply(): Square = ???

  /** The total number of moves that have been made for all squares. */
  def totalNumberOfMoves: Int = ???

  /** The total cost of all squares. */
  def totalCost: Double = ???

```

b) Testa din kvadratprototyp i REPL. Använd t.ex. koden nedan:

```

1 scala> val xs = Vector.fill(10)(Square())
2 scala> xs.foreach(_.move(2, 3))
3 scala> xs.foreach(_.scale(2.9))
4 scala> val (m, c) = (Square.totalNumberOfMoves, Square.totalCost)
5 val m: Int = 10
6 val c: Double = 36.055512754639885

```

★ **Uppgift 17. Hjälpkonstruktor.** I tidigare uppgifter har vi möjliggjort alternativa sätt att skapa instanser genom default-argument och fabriksmetoder i kompanjonsobjekt.

Ett annat sätt att göras detta på, som i Scala är ovanligt<sup>9</sup>, är att definiera flera konstruktörer inne i klasskroppen. I Scala kallas en sådan extra konstruktor för **hjälpkonstruktor** (eng. *auxiliary constructor*).

En hjälpkonstruktor skapar man i Scala genom att definiera en metod som har det speciella namnet `this`, alltså en deklaration `def this(...) = ...`. Hjälpkonstruktörer måste börja med att anropa en annan konstruktor, antingen den primära konstruktorn (d.v.s. den som klasshuvudet definierar) eller en tidigare definierad hjälpkonstruktor.

a) Läs mer om hjälpkonstruktörer här:

[www.artima.com/pins1ed/functional-objects.html#6.7](http://www.artima.com/pins1ed/functional-objects.html#6.7)

b) Hitta på en egen uppgift med hjälpkonstruktörer, baserat på någon av klasserna i tidigare övningar.

<sup>9</sup>Men i Java är detta mycket vanligt då defaultargument m.m. inte ingår i språket.

## 5.3 Laboration: blockbattle0

### Förberedelser

- ☐ Gör övning classes i avsnitt [5.2](#).
- ☐ Du har två veckor på dig att göra blockbattle. Läs redan nu igenom alla uppgifter i avsnitt [6.3](#), men gör först grundövningarna innan du påbörjar labben, speciellt uppgift [6](#) i denna veckas övningar.

## Kapitel 6

# Mönster och felhantering

Begrepp som ingår i denna veckas studier:

- ☐ mönstermatchning
- ☐ match
- ☐ Option
- ☐ throw
- ☐ try
- ☐ catch
- ☐ Try
- ☐ unapply
- ☐ sealed
- ☐ flatten
- ☐ flatMap
- ☐ partiella funktioner
- ☐ collect
- ☐ wildcard-mönster
- ☐ variabelbindning i mönster
- ☐ sekvens-wildcard
- ☐ bokstavliga mönster
- ☐ implementera equals
- ☐ hashCode

## 6.1 Teori

### 6.1.1 Bastypen för alla typer: Any

Scalas typsystem är **fullständigt**:

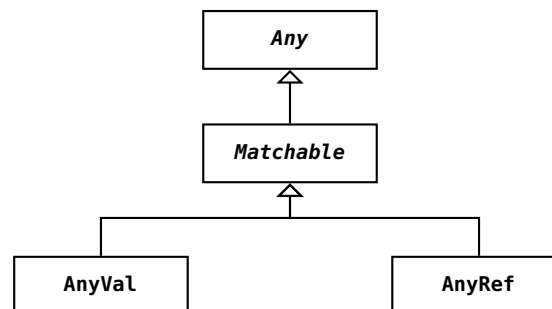
- Alla värden är objekt som har en typ.
- Alla typer är subtyper till bastypen Any.
- Typen Any kallas därför **topptyp**.
- Alla objekt har vissa grundläggande metoder, så som toString och ==

```
trait Any:           // en förenklad beskrivning av Any
  def toString
  def ==
  def !=
  def equals
  def isInstanceOf
  def asInstanceOf
  def ##
  def hashCode
  def getClass

trait Matchable extends Any
```

Det kommer mer om abstrakta klasser och traits i veckan om arv.

### 6.1.2 Alla typer är subtyper till Any



- Alla **värdetyper**, t.ex. Int, Double, Boolean, är subtyper till AnyVal
- Alla **referenstyper** t.ex. String är subtyper till AnyRef
- Värden av typen Matchable kan användas vid s.k mönstermatchning.
- (Det finns även s.k. opaka typer som inte kan mönstermatchas.)

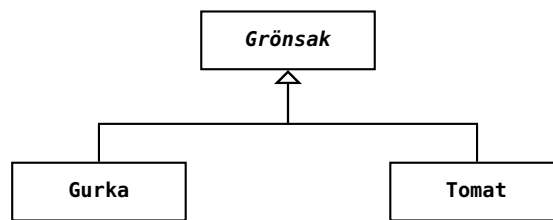
### 6.1.3 Dina egna referenstyper är subtyper till AnyRef

Alla typer du skapar är subtyper till AnyRef utan att du behöver skriva det.

```
trait Grönsak:           // din egen bastyp
  def vikt: Int

case class Gurka(vikt: Int) extends Grönsak // din egen subtyp
```

```
case class Tomat(vikt: Int) extends Grönsak // en annan subtyp
```



Det kommer mer om typhierarkier och **extends** i veckan om arv.

I ett match-uttryck kan man matcha på ett visst värde eller på en viss typ och match-uttryck används gärna istället för nästlade if-uttryck, då de ofta är lättare att läsa och begripa. Med match-uttryck kan man också göra **mönstermatchning** mot case-klass-instanser, t.ex. för att på ett smidigt sätt undersöka om attribut har speciella värden. Match-uttryck i Scala är en mer kraftfull variant av switch-satser som finns i många andra språk.

#### 6.1.4 Vad är matchning?

- Matchning gör man då man vill jämföra ett värde mot andra värden och hitta överensstämmelse (eng. *match*) enligt olika **mönster**.
- Med mönster kan man även **plocka isär** objekt i sina beståndsdelar.

#### 6.1.5 Plocka isär ett objekt i sina beståndsdelar med mönster

```
scala> case class Point(x: Int, y: Int)

scala> val p = Point(1, 2) // konstruera en punkt
val p: Point = Point(1,2)
```

```
scala> val Point(x, y) = p // plocka isär en punkt
val x: Int = 1
val y: Int = 2
```

`Point(x, y)` kallas ett **konstruktormönster**.

Namnen `x` och `y` blir nya variabler.

Det finns många olika sorters mönster.

Vanligaste användningen av mönster är i **match**-uttryck.

#### 6.1.6 Kolla om det passar med nästlade if-uttryck

Ett vanligt problem:

att kolla vilket bland många värden som passar

Kan göras med nästlade **if-then-else**-uttryck:

```

val g = scala.io.StdIn.readLine("Ange en grönsak:")
val smak =
  if      g == "gurka"      then "gott!"
  else if g == "tomat"     then "jättegott!"
  else if g == "broccoli"  then "ganska gott..."
  else "inte gott :("

println(g + " är " + smak)

```

### 6.1.7 Kolla om det passar med match-uttryck

I stället för nästlade **if** kan du använda Scalas kraftfulla **match-uttryck**:

```

val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak =
  g match
    case "gurka"      => "gott!"
    case "tomat"      => "jättegott!"
    case "broccoli"   => "ganska gott..."
    case _            => "mindre gott..."

```

- Varje **case**-gren testas var för sig i tur och ordning **uppifrån och ned**.
- Det som står mellan **case** och **=>** kallas ett **mönster** (eng. *pattern*)
- Om ett mönster matchar så görs det som står efter **=>**
- **Inga efterföljande case-grenar testas efter lyckad match.**
- Ovan är exempel på matchning mot **konstant-mönster**, i detta fallet tre stycken strängkonstantmönster.
- Sista default-grenen ovan kallas **wildcard-mönster**: **case \_ =>**
- Det finns många andra sätt att skriva mönster.

### 6.1.8 Syntax för match-uttryck

Ett **match**-uttryck består av godtyckligt många **case ... => ...**

```

värdeAttUndersöka match {
  case mönster1 => resultat1
  case mönster2 => resultat2
  case mönster3 => resultat3
  case mönsterN => resultatN
}

```

- Klammerparenteser efter **match** valfria om **case** på ny rad.
- Varje resultat-uttryck kan bestå av många rader.

- Klammerparenteser behövs ej efter **=>** vid många rader.

Om många rader efter **case** så blir sista uttrycket resultatet.  
Vi ska nu se exempel på många olika mönster

---

### 6.1.9 Matchning med gard

Man kan stoppa in en s.k **gard** (eng. *guard*) innan pilen **=>** för att villkora matchningen: (notera **if** utan **then**)

```
val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak = g match
  case "gurka" if math.random() > 0.5 => "gott ibland!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
```

**case**-grenen med gard ger bara en lyckad matchning om uttrycket efter **if** är sant; annars provas nästa gren, etc.

---

### 6.1.10 Matchning med variabelmönster

Om det finns ett namn efter **case** som börjar med liten begynnelsebokstav, blir detta namn en variabel som automatiskt binds till uttrycket före **match**:

```
val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak = g match
  case "gurka" if math.random() > 0.5 => "gott ibland!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case other => "smakar bakvänt: " + other.reverse
```

Ett **enkelt variabelmönster**, så som  
**case** other => ...  
i exemplet ovan, matchar **allt**!  
other får alltså värdet av g om g **inte** är "gurka", "tomat", "broccoli".

---

### 6.1.11 Matchning med eller-mönster

Om man har samma utfall för olika grenar kan dessa slås ihop och mönstret separeras med vertikalstreck: |

```
val g = scala.io.StdIn.readLine("Ange en grönsak: ")
val smak = g match
  case "gurka" => "gott"
  case "tomat" => "gott"
  case "lök"   => "gott"
```



```
case _ => "inte gott"
```

Mer koncist med eller-mönster:

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak = g match
  case "gurka" | "tomat" | "lök" => "gott"
  case _ => "inte gott"
```

### 6.1.12 Matchning med typade mönster

Med en typannotering efter en variabel får man ett **typat mönster** (eng. *typed pattern*). Om matchningen lyckas blir värdet **omvandlat** till den specifika typen och binds till variabeln.

```
def f() =
  if math.random() < 0.5 then 42 + math.random()
  else s"gurka ${math.random()}"

val i = f() match
  case x: Double => x.round.toInt
  case s: String => s.length
```

`f()` får typen `Matchable` som är subtyp till `Any`. Vilken typ får `i`? `Int`

Matchning mot specifika typer enl. ovan används i idiomatisk Scala hellre än `isInstanceOf` och `asInstanceOf` men man kan göra motsvarande ovan så här:

```
val i2: Int =
  val x = f()
  if x.isInstanceOf[Double] then x.asInstanceOf[Double].round.toInt
  else if x.isInstanceOf[String] then x.asInstanceOf[String].length
  else throw scala.MatchError(x)
```

### 6.1.13 Fördjupning: Unionstyper och typen Matchable

- Exempel: För de orelaterade typerna `String` och `Int` är den mest specifika typen som kan härledas `Int | String`, läses "Int eller String" och kallas en **unionstyp** (eng. *union type*).

```
scala> def f() = math.random() match
         case a if a > 0.5 => 42
         case a if a < 0.2 => "hej"
         ;
def f(): Int | String
```

- Alla värden som kan undersökas med `match` har typen `Matchable`.
- Typen `Matchable` är nästan lika generell som toptypen `Any`.

```
scala> (f().asInstanceOf[Matchable], f().asInstanceOf[Any])
val res0: (Boolean, Boolean) = (true,true)
```

- Matchable infördes i Scala 3 med **opaka typalias** som garanterat aldrig boxas men inte kan mönstermatchas. (Ingår ej i denna kurs.)
- Fördjupning om Matchable och **opaque type** i Scala 3 finns här: <https://dotty.epfl.ch/docs/reference/other-new-features>

### 6.1.14 Konstruktormönster med case-klasser

En basclass med gemensamma delar och två subtyper:

```
trait Grönsak:
  def vikt: Int
  def ärRutten: Boolean

case class Gurka(vikt: Int, ärRutten: Boolean) extends Grönsak
case class Tomat(vikt: Int, ärRutten: Boolean) extends Grönsak
```

Tack vare case-klasserna kan man använda **konstruktormönster** (eng. *constructor pattern*) för att se vad som finns **inuti** en instans:

```
def testa(g: Grönsak): String = g match
  case Gurka(v, false) => "gott, väger " + v
  case Gurka(_, true)  => "inte gott"
  case Tomat(v, r)     => (if r then "inte " else "") + s"gott, väger $v"
  case _               => s"okänd grönsak: $g"
```

Konstruktormönster **”plockar isär”** det som matchas och binder variabler till de attribut som finns i case-klassens konstruktor.

### 6.1.15 Plocka isär samlingar med djupa mönster

- Man kan plocka isär innehållet i en samling så här:

```
def visa(xs: Vector[Grönsak]): String = xs match
  case Vector()           => "tom grönsaksvektor"
  case Vector(Gurka(v, true)) => s"en rutten gurka som väger $v"
  case Vector(g)          => s"exakt en grönsak: $g"
  case Vector(g1, g2)     => s"exakt två grönsaker: $g1, $g2"
  case Vector(g, gs*)     => s"först en $g och sedan svansen: $gs"
```

- Vad händer om du byter ordning på 2:a och 3:e mönstret?
- Vector(g, gs\*) kan också skrivas som g +: gs

### 6.1.16 Matchning på tupler

Det går fint att plocka isär tupler med mönstermatchning:<sup>1</sup>

```
var pair = ("hej", 42)

pair match
  case (a, b) if b == 42 => s"livets mening är funnen: $a"
  case (_, b)           => s"fattas mening: $b"
```

### 6.1.17 Mönstermatchning och uppräknings med case-objekt

En bastyp och specifika singelobjekt av gemensam typ:

```
trait Färg
case object Spader extends Färg // funkar utan case men vi vill ha najs toString
case object Hjärter extends Färg
case object Ruter extends Färg
case object Klöver extends Färg

def parallellFärg(f: Färg): Färg = f match
  case Spader => Klöver
  case Klöver => Spader
  case Hjärter => Ruter
```

Vilken case-gren har vi glömt? Kan kompilatorn hjälpa oss?

```
1 scala> parallellFärg(Ruter)
2 scala.MatchError: Ruter
```

**Undantag vid körtid :** (

### 6.1.18 Mönstermatchning och förseglade typer

Med nyckelordet **sealed** får vi en kompilersvarning.

```
sealed trait Färg //tryck Alt+Enter i REPL för tolkning av flera rader ett svep
case object Spader extends Färg
case object Hjärter extends Färg
case object Ruter extends Färg
case object Klöver extends Färg

def parallellFärg(f: Färg): Färg = f match
  case Spader => Klöver
  case Klöver => Spader
  case Hjärter => Ruter
```

<sup>1</sup><https://youtu.be/aboZctrHfK8>

```

1 |def parallellFärg(f: Färg): Färg = f match
2 |   ^
3 |   match may not be exhaustive.
4 |
5 |   It would fail on pattern case: Ruter
6 |def parallellFärg(f: Färg): Färg

```

**Varning vid kompilering :**) Sista raden visar att det bara är en varning!

---

### 6.1.19 Mönstermatcha enumeration

I stället för **sealed trait ... case object ...** kan du använda en **enumeration** (ä.k. uppräknig, uppräknad datatyp, (eng. *enumeration*)).

```

enum Färg:
  case Spader, Hjärter, Ruter, Klöver

def parallellFärg(f: Färg): Färg =
  import Färg.*
  f match
    case Spader => Klöver
    case Klöver => Spader
    case Hjärter => Ruter

```

```

1 |def parallellFärg(f: Färg): Färg
2 | 3 | f match
3 |   ^
4 |   match may not be exhaustive.
5 |
6 |   It would fail on pattern case: Ruter

```

**Även här får vi hjälpsam varning vid kompilering :**)

---

### 6.1.20 Stora/små begynnelsebokstäver vid matchning

**Fallgrop:** matcha **värde** som börjar med **liten** bokstav.

```

1 |scala> val livetsMening = 42
2 |
3 |scala> def ärLivetsMeningBuggig(svar: Int) = svar match
4 |   case livetsMening => true    // lokalt namn som matchar allt!
5 |   case _ => false
6 |
7 |scala> ärLivetsMeningBuggig(43)
8 |val res0: Boolean = true
9 |
10|scala> val LivetsMening = 42    // stor begynnelsebokstav
11|
12|scala> def ärLivetsMening(svar: Int) = svar match
13|   case LivetsMening => true    // funkar fint!
14|   case _ => false
15|
16|scala> ärLivetsMening(43)
17|val res1: Boolean = false

```

### 6.1.21 Stora/små begynnelsebokstäver vid matchning

Ett sätt att komma runt problemet med liten begynnelsebokstav:

**backticks** to the rescue!

```
1 scala> val livetsMening = 42
2
3 scala> def ärLivetsMeningBackTicks(svar: Int) = svar match
4     case `livetsMening` => true    // nu funkar det!
5     case _ => false
6
7 scala> ärLivetsMeningBackTicks(43)
8 val res2: Boolean = false
```

### 6.1.22 Mönster på andra ställen än i match

Mönster i **deklarationer**:

```
1 scala> case class Point(x: Int, y: Int)
2
3 scala> val p = Point(0, 1)
4
5 scala> val Point(x, y) = p           // konstruktormönster med case-klass
6 val x: Int = 0
7 val y: Int = 1
8
9 scala> val (x, y, z) = (0, 1, 2)     // konstruktormönster med tupel
10 val x: Int = 0
11 val y: Int = 1
12 val z: Int = 2
```

Mönster i **for-uttryck**:

```
1 scala> val xs = for (x, y) <- Vector((1,2), (3,4)) yield x
2 val xs: Vector[Int] = Vector(1, 3)
```

### 6.1.23 Mönsterdelar och variabelt antal argument

Met två olika specialtecken går det att

- binda variabler till **mönsterdelar** med **@**  
**case** Vector(xs@Vector(a), Vector(42)) => ...
- matcha **variabelt antal argument** med **\***  
**case** Vector(a, \_, c) => ... matchar om 3 element, \_ kvittar  
**case** Vector(a, svans\*) => ... matchar om minst ett element  
**case** Vector(a, \_\*) => ... intresserad av första, svans kvittar

### 6.1.24 Partiella funktioner och metoden collect

- En **partiell funktion** är, till skillnad från en **total funktion**, inte definierad för alla parametervärden.
- Partiella funktioner kan skapas med **case** utan **match**:

```
val pf: PartialFunction[Int, Double] = { case z if z != 0 => 1.0 / z }
```

- Funktionen är inte definierad för argumentet 0:

```
scala> pf(0)
scala.MatchError: 0
```

- Detta är användbart tillsammans med samlingsmetoden collect som applicerar en partiell funktion endast på definierade värden:

```
scala> Vector(1, 2, 0, 4).collect(pf)
val res0: Vector[Double] = Vector(1.0, 0.5, 0.25)

scala> Vector(1 -> 2, 0 -> 3, 42 -> 0).collect{ case (a,b) if a > 0 => a }
val res1: Vector[Int] = Vector(1, 42)
```

- Notera att **krullparentes behövs** vid ensamt **case**.

### 6.1.25 Fördjupning: metoden unapply

När du deklarerar en case-klass kommer kompilatorn att **automatiskt generera en metod** med namnet **unapply**.

```
1 scala> case class Gurka(vikt: Int, ärRutten: Boolean)
2
3 scala> Gurka.unapply // tryck ENTER för att se typen
4 val res0: Gurka => Gurka = Lambda$1914/0x000000008408cf840@b0e7bde
5
6 scala> val g = Gurka(100, false)
7
8 scala> Gurka.unapply(g)
9 val res1: Gurka = Gurka(100,false)
```

**Vad ska detta vara bra för?** Metoden unapply genereras av kompilatorn och används internt vid matchning och det är den metoden som gör att case-klasser kan användas i konstruktormönster. Principen är generell: Man kan skapa **egna** s.k. **extraktorer** (eng. *extractors*) som kan plocka isär ett värde med mönstermatchning, även utan case-klass. För den nyfikne: <https://docs.scala-lang.org/scala3/reference/changed-features/pattern-matching.html>

### 6.1.26 Hur hantera saknade värden?

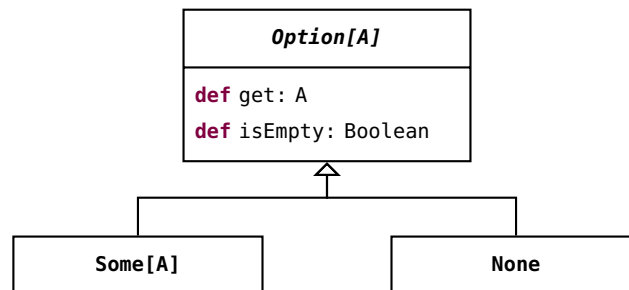
Olika sätt att hantera saknade värden:

- Hitta på ett specialvärde: exempel -1 för saknat värde
- **null** om värde saknas (vanligt i Java m.fl. språk, mkt ovanligt i Scala)
- Använd en samling och låt tom samling representera saknat värde:

```
val sums = Vector(Vector(42), Vector(32), Vector(), Vector(21))
```

- `Option[A]` gemensam bastyp för:  
`None` som representerar **saknat värde**, och  
`Some[A]` som representerar att **värde finns**

### 6.1.27 En gemensam bastyp för ett värde som kanske saknas



```

1 scala> var x: Option[Int] = Some(42)
2
3 scala> x.isEmpty
4 val res0: Boolean = false
5
6 scala> x = None
7
8 scala> x.isEmpty
9 val res1: Boolean = true

```

### 6.1.28 Option för hantering av ev. saknade värden

Alla vill inte berätta för Facebook vad de har för kön.  
 Förbättra Facebooks kod med ett litet Scala-program:

```

enum Gender:
  case Male, Female

case class Person(name: String, gender: Option[Gender])

```

```

1 scala> val p1 = Person("Björn", Some(Gender.Male))
2 scala> val p2 = Person("Sandra", Some(Gender.Female))
3 scala> val p3 = Person("Kim", None)
4 scala> val g2 = p2.gender
5 scala> def show(g: Option[Gender]): String = g match {
6     case Some(x) => x.toString
7     case None    => "unknown"
8   }
9 scala> show(g2)
10 scala> show(p3.gender)
11 scala> val ps = Vector(p1,p2,p3)
12 scala> ps.map(_.gender).map(show) // None ignoreras av map

```

### 6.1.29 Några smidiga metoder på Option

Metoden `getOrElse` gör att man ofta kan undvika matchning.

```
var opt: Option[Int] = None

val x = opt.getOrElse(42) // get the value, give default if missing
```

Flera av de vanliga samlingsmetoderna funkar, t.ex. `foreach` och `map`.

```
opt.foreach(x => println(x)) // only done if value exists

opt.map(x => x + 1)           // only done if value exists

opt = Some(42)              // change opt to now have some value

opt.foreach(x => println(x)) // done as value now exists

opt.map(x => x + 1)          // done as value now exists
```

### 6.1.30 Några samlingsmetoder som ger en Option, övning

```
1 scala> val (xs, ys) = (Vector(1,2,3), Vector())
2
3 scala> xs.headOption
4 res0: ???
5
6 scala> ys.headOption
7 res1: ???
8
9 scala> xs.find(_ > 1)
10 res2: ???
11
12 scala> xs.find(_ > 5)
13 res3: ???
14
15 scala> val huvudstad = Map("Sverige" -> "Sthlm", "Skåne" -> "Malmö")
16
17 scala> huvudstad.get("Skåne")
18 res4: ???
19
20 scala> huvudstad.get("Danmark")
21 res5: ???
```

### 6.1.31 Några samlingsmetoder som ger en Option, svar

```
1 scala> val (xs, ys) = (Vector(1,2,3), Vector())
2
3 scala> xs.headOption
4 res0: Option[Int] = Some(1)
5
```



```
6 scala> ys.headOption
7 res1: Option[Nothing] = None
8
9 scala> xs.find(_ > 1)
10 res2: Option[Int] = Some(2)
11
12 scala> xs.find(_ > 5)
13 res3: Option[Int] = None
14
15 scala> val huvudstad = Map("Sverige" -> "Sthlm", "Skåne" -> "Malmö")
16
17 scala> huvudstad.get("Skåne")
18 res4: Option[String] = Some(Malmö)
19
20 scala> huvudstad.get("Danmark")
21 res5: Option[String] = None
```

---

### 6.1.32 Vad är ett undantag (eng. *exception*)?

Undantag representerar ett fel eller ett onormalt tillstånd som upptäcks under exekvering och som behöver hanteras på särskilt sätt vid sidan av det normala exekveringsflödet.

[sv.wikipedia.org/wiki/Undantagshantering](https://sv.wikipedia.org/wiki/Undantagshantering)

Exempel på undantag:

- Indexering utanför vektorns indexgränser.
- Läsning bortom filens slut.
- Försök att öppna en fil som inte finns.
- Minnet är slut.
- Heltalsdivision med noll ger `java.lang.ArithmeticException`.
- `"hej".toInt` ger `java.lang.NumberFormatException`

---

### 6.1.33 Orsaka undantag indirekt med `require` och `assert`

- Med funktionen `require(b)` skapas ett `IllegalArgumentException("requirement failed")` om `b` är **false**
- `require` används om man vill begränsa vilka argument som är giltiga
- Med funktionen `assert(b)` skapas ett `AssertionError("assertion failed")` om `b` är **false**
- `assert` används om man vill förhindra ogiltiga tillstånd

Se implementationen av `require` här:

<https://github.com/scala/scala/blob/v2.13.6/src/library/scala/Predef.scala#L315>

---

### 6.1.34 Kasta dina egna undantag med throw

Man kan själv generera ett undantag med **throw**, vilket kallas att **kasta** ett undantag som (om det inte **fångas**), gör att exekveringen **avbryts**.

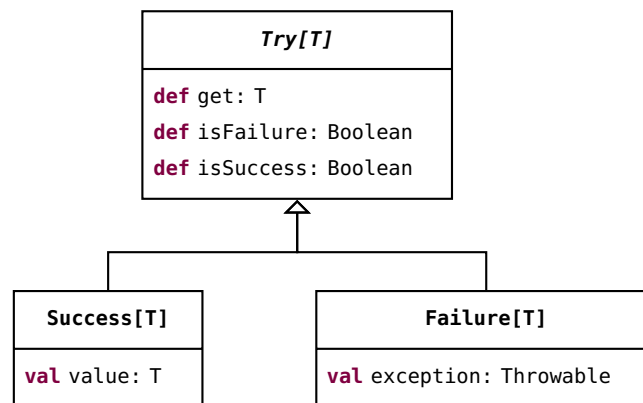
```
1 scala> def pang = throw Exception("PANG!")
2 pang: Nothing
3
4 scala> pang
5 java.lang.Exception: PANG!
```

Olika sätt att hantera undantag och förhindra att exekveringen avbryts:

- **try catch**-uttryck omvandlar undantag till ngt lämpligt värde.
- `scala.util.Try`  **kapslar in**  kod som kan ge undantag.

### 6.1.35 En gemensam bas typ för något som kan misslyckas

```
import scala.util.{Try, Success, Failure}
```



### 6.1.36 Hantera undantag med Try

```
scala> def pang = throw new Exception("PANG!")

scala> def kanskePang = if math.random() < 0.5 then 42 else pang

scala> import scala.util.{Try, Success, Failure}

scala> def försök = Try { kanskePang }

scala> val xs = Vector.fill(15){försök}

scala> val trettonde = xs(12) match
  case Success(value) => value
  case Failure(e) => println(e); -1

scala> (xs(12).isSuccess, xs(12).isFailure)

scala> xs(12).getOrElse(0)

scala> xs(12).toOption
```

```
scala> försök.foreach(println)

scala> försök.map(_ + 1)

scala> for Success(x) <- xs yield x
```

### 6.1.37 try-catch-uttryck

Man kan fånga undantag med ett **try** ... **catch**-uttryck:

```
def carola =
  try
    if math.random() > 0.5 then throw Exception("stormvind")
    42
  catch
    case e: Exception =>
      println("Fångad av en " + e.getMessage)
      -1
```

```
1 scala> Vector.fill(5)(carola)
2 Fångad av en stormvind
3 Fångad av en stormvind
4 Fångad av en stormvind
5 val res0: Vector[Int] = Vector(-1, 42, 42, -1, -1)
```

### 6.1.38 Unvik undantag om det går

**Fördelar** med undantag:

- Vid allvarliga fel då det inte är mycket att göra än att starta om, t.ex. `OutOfMemoryException`, är det bra att få veta vad som är fel.
- Onormala fall som uppkommer sällan kan hanteras separat (t.ex. i huvudprogrammet) utan att koden för normalfallet blir tillkrånglad.

**Nackdelar** med undantag:

- Ett slags "goto" som gör exekveringsflödet svårt att följa.
- Skapa stack-trace tar tid; undantag som sker ofta påverkar prestanda.

Exempel: undantagslösa `toIntOption` är både säker och snabb!

```
scala> def time(op: => Unit): Long = {val t0 = System.nanoTime; op; System.nanoTime - t0}

scala> def min(op: => Unit, n: Int = 1000): Long = Seq.fill(n)(time(op)).drop(n / 20).min

scala> min(util.Try("hello".toInt))
val res0: Long = 3549

scala> min(try "hello".toInt catch (_: Throwable) => ())
val res1: Long = 3046

scala> min("hello".toIntOption)
val res2: Long = 157
```

### 6.1.39 Fördjupning: Kontrollerade undantag

- Det finns möjligheter i Scala att låta kompilatorn kontrollera om undantag hanteras.
- Läs mer här:  
<https://docs.scala-lang.org/scala3/reference/experimental/canthrow.html>

När du jämför värden med `==` anropas metoden `equals` som finns för alla typer. Du kan i dina egna klasser överskugga `equals` med en din egna definition av vad likhet ska innebära. Då är det lämpligt att använda matchning. Det är dock ett ganska omfattande arbete att implementera en korrekt likhetsjämförelse som fungerar under alla omständigheter. Ett recept för en fullständig implementation av `equals` ges i fördjupningen nedan.

### 6.1.40 Fördjupning: Implementera `equals` med `match`

Det visar sig att **innehållslikhet** är **förvånansvärt komplicerat** att implementera, speciellt i samband med arv.

- Det enklare fallet: Gör fördjupningsuppgift ”Metoden `equals`” och implementera `equals` för innehållslikhet utan arv.  
 En bra träning på att använda **match**!
- Svårare: Gör fördjupningsuppgifterna ”Överskugga `equals`” och ”Överskugga `equals` vid arv” om du vill se hur en **komplett** `equals` ska se ut som fungerar **i alla lägen**.

Det krävs i denna kurs inte att du själv ska kunna implementera en generellt fungerande `equals`. Men du ska förstå skillnaden mellan referenslikhet och innehållslikhet. Mer om `equals` i fortsättningkursen, men en liten inblick i problemet nu...

Om en klass markeras **final** kan den ej ha några subclasser. Kompilatorn kontrollerar att detta gäller alla finala klasser och ger kompileringsfel om du försöker göra **extends** på en final klass. Om en klass garanterat inte har några subclasser kan implementationen av `equals` göra enklare.

### 6.1.41 Fördjupning: `equals` som fungerar för finala klasser

Recept för implementation av `equals` som fungerar för typer som **inte** har några subtyper:

```
final class Gurka(val vikt: Int, val ärÄtbar: Boolean):
  override def equals(other: Any): Boolean = other match
    case that: Gurka => vikt == that.vikt && ärÄtbar == that.ärÄtbar
    case _ => false

  override def hashCode: Int = (vikt, ärÄtbar).## // ger bra hashcode
```

- Du **måste** alltid överskugga `hashCode` också om du överskuggar `equals` annars funkar inte gurksamlingar (lång story ...)

- Notera typen Any – detta följer hur man valde att göra i Java (tyvärr?).
- Ett **typsäkrare** innehållslikhetstest som **garanterat** bara jämför en gurka med en gurka och inget annat:

```
def ==(other: Gurka): Boolean =
  vikt == other.vikt && ärÄtbar == other.ärÄtbar
```

#### 6.1.42 Fördjupning: Recept i 8 steg för arvssäker equals

1. Inför denna metod: **def** canEqual(other: Any): Boolean  
Observera att typen på parametern ska vara Any. Om subclass behövs **override**.
2. Metoden canEqual ska ge **true** om other är av samma typ som this, t.ex.:  
**override def** canEqual(other: Any): Boolean = other.isInstanceOf[Gurka]
3. Inför metoden equals och var noga med att parametern har typen Any:  
**override def** equals(other: Any): Boolean
4. Implementera metoden equals med ett match-uttryck som börjar så här:  
other **match** { ... }
5. Match-uttrycket ska ha två grenar. Den första grenen ska ha ett typat mönster för den klass som ska jämföras, t.ex.:  
**case** that: Gurka =>
6. Om du implementerar equals i den klass som inför canEqual, börja med:  
(that canEqual this) &&  
och skapa därefter en fortsättning som baseras på innehållet i klassen, t.ex.:  
this.vikt == that.vikt && this.längd == that.längd  
Om du överskuggar equals vill du nog börja med **super.equals(that)** &&
7. Den andra grenen i matchningen ska vara: **case \_ => false**
8. Överskugga hashCode, t.ex. med tupel av attributvärden och metoden ##:  
**override def** hashCode: Int = (vikt, längd).##

<http://www.artima.com/pinsled/object-equality.html>

#### 6.1.43 Fördjupning: Säkrare likhetstest i Scala 3

- **Problem:** equals tar värden av vilken typ som helst.
- Detta kallas **universell likhet**.

```
scala> case class Hund(namn: String)
scala> case class Katt(namn: String)
scala> Hund("bob") == Katt("bob") // knasig jämförelse; kan aldrig bli sant
val res0: Boolean = false        // men kompilatorn låter dig göra likhetstestet
```

- I Scala 3 kan du få typsäker likhetstest med **derives CanEqual**
- Detta kalla **multiversell likhet**.

```
scala> case class Hund(namn: String) derives CanEqual
scala> Hund("bob") == Katt("bob") // tack kompilatorn för fel:
-- Error:
1 |Hund("bob") == Katt("bob")
  |^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |Values of types Hund and Katt cannot be compared with == or !=
```

- Du **slipper** skriva **derives** CanEqual om du gör:  
**import** scala.language.strictEquality
  - Läs mer här: <https://docs.scala-lang.org/scala3/reference/contextual/multiversal.html>
-

## 6.2 Övning patterns

### Mål

- ☐ Kunna skapa och använda **match**-uttryck med konstanta värden, gardar och mönstermatchning med case-klasser.
- ☐ Kunna skapa och använda case-objekt för matchningar på uppräknade värden.
- ☐ Kunna hantera saknade värden med hjälp av typen `Option` och mönstermatchning på `Some` och `None`.
- ☐ Kunna fånga undantag med `scala.util.Try`.
- ☐ Känna till **try**, **catch** och **throw**.
- ☐ Känna till nyckelordet **sealed** och förstå nyttan med förseglade typer.

### Förberedelser

- ☐ Studera begreppen i kapitel 6

### 6.2.1 Grunduppgifter; förberedelse inför laboration

#### Uppgift 1. Matcha på konstanta värden.

a) Skriv nedan program med en kodeditor och spara i filen `Match.scala`. Kompilera och kör och ge som argument din favoritgrönsak. Vad händer? Förklara hur ett **match**-uttryck fungerar.

```

1 object Match:
2   def main(args: Array[String]): Unit =
3     val favorite = if args.length > 0 then args(0) else "selleri"
4     println("Din favoritgrönsak: " + favorite)
5     val firstChar = favorite.toLowerCase.charAt(0)
6     val meThink = firstChar match
7       case 'g' => "gurka är gott!"
8       case 't' => "tomat är gott!"
9       case 'b' => "broccoli är gott!"
10      case _   => s"$favorite är mindre gott..."
11    println(s"Jag tycker att $meThink")

```

b) Vad blir det för felmeddelande om du tar bort case-grenen för defaultvärden och indata väljs så att inga case-grenar matchar? Är det ett exekveringsfel eller ett kompileringsfel?

**Uppgift 2. Gard i case-grenar.** Med hjälp en gard (eng. *guard*) i en case-gren kan man begränsa med ett villkor om grenen ska väljas.

Utgå från koden i uppgift 1a och byt ut case-grenen för 'g'-matchning till nedan variant med en gard med nyckelordet **if** (notera att det inte behövs parenteser runt villkoret):

```
case 'g' if math.random() > 0.5 => "gurka är gott ibland..."
```

Kompilera om och kör programmet upprepade gånger med olika indata tills alla grenar i **match**-uttrycket har exekverats. Förklara vad som händer.

**Uppgift 3. Mönstermatcha på attributen i case-klasser.** Scalas **match**-uttryck är extra kraftfulla om de används tillsammans med **case**-klasser: då kan attribut extraheras automatiskt och bindas till lokala variabler direkt i case-grenen som nedan exempel visar (notera att `v` och `rutten` inte behöver deklarerats explicit). Detta kallas för **mönstermatchning**. Vad skrivs ut nedan? Varför? Prova att byta namn på `v` och `rutten`.

```

1 scala> case class Gurka(vikt: Int, ärRutten: Boolean)
2 scala> val g = Gurka(100, true)
3 scala> g match { case Gurka(v,rutten) => println("G" + v + rutten) }

```

**Uppgift 4.** Matcha på case-objekt och nyttan med **sealed**. Skriv nedan kodrader i en REPL en för en. Notera nyckelordet **sealed** som används för att försegla en typ. En **förseglad typ** måste ha alla sina subtyper i en och samma kodfil.

```

1 scala> sealed trait Färg
2 scala> case object Spader extends Färg

```

a) Hur lyder felmeddelandet och varför sker det? Är det ett kompileringsfel eller ett körtidsfel?

b) Skapa nu nedan kod i en editor och klistra in i REPL.

```

object Kortlek:
  sealed trait Färg
  object Färg:
    val values = Vector(Spader, Hjärter, Ruter, Klöver)
    case object Spader extends Färg
    case object Hjärter extends Färg
    case object Ruter extends Färg
    case object Klöver extends Färg

```

c) Skapa en funktion **def** parafärg(f: Färg): Färg i en editor, som med hjälp av ett match-uttryck returnerar parallellfärgen till en färg. Parallellfärgen till Hjärter är Ruter och vice versa, medan parallellfärgen till Klöver är Spader och vice versa. Klistra in funktionen i REPL. Passa även på att skriva en **import**-sats för det yttre objektet **Kortlek**, så medlemmarna av objektet kan nå enkelt.

```

1 scala> parafärg(Spader)
2 scala> val xs = Vector.fill(5)(Färg.values((math.random() * 4).toInt))
3 scala> xs.map(parafärg)

```

d) Vi ska nu undersöka vad som händer om man glömmer en av case-grenarna i matchningen i parafärg. "Glöm" alltså avsiktligt en av case-grenarna och klistra in den nya parafärg med den ofullständiga matchningen. Hur lyder varningen? Kommer varningen vid körtid eller vid kompilering?

e) Anropa parafärg med den "glömda" färgen. Hur lyder felmeddelandet? Är det ett kompileringsfel eller ett körtidsfel?

f) Förklara vad nyckelordet **sealed** innebär och vilken nytta man kan ha av att **försegla** en supertyp.

**Uppgift 5.** Mönstermatcha enumeration. Vi ska nu undersöka och jämföra skillnad mellan nyckelorden **enum** och **sealed trait**. Skriv nedan kod i en REPL.

```

enum Färg:
  case Spader, Hjärter, Ruter, Klöver

```



a) Skapa med hjälp av en editor igen en funktion **def** `parafärg(f: Färg): Färg`, nästintill likadan som den som vi skapade i deluppgift 4c. Funktionen ska återigen utnyttja `match`-uttryck för att returnera parallellfärgen till argumentet som ges. Tänk på att denna gången är `Färg` inget **sealed trait**, utan istället en enumeration (**enum**). Klistra in funktionen i REPL.

```
1 scala> parafärg(Färg.Ruter)
2 scala> val xs = Vector.fill(5)(Färg.values((math.random() * 4).toInt))
3 scala> xs.map(parafärg)
```

b) Fundera på skillnader och likheter mellan att utnyttja **sealed trait** ihop med **case**-objekt gentemot att använda sig av **enum** vid mönstermatchning.

**Uppgift 6.** *Betydelsen av små och stora begynnelsebokstäver vid matchning.* För att åstadkomma att namn kan bindas till variabler vid matchning utan att de behöver deklarerats i förväg (som vi såg i uppgift 3) så har identifierare med liten begynnelsebokstav fått speciell betydelse: den tolkas av kompilatorn som att du vill att en variabel binds till ett värde vid matchningen. En identifierare med stor begynnelsebokstav tolkas däremot som ett konstant värde (t.ex. ett case-objekt eller ett case-klass-mönster).

a) *En case-gren som fångar allt.* En case-gren med en identifierare med liten begynnelsebokstav som saknar gard kommer att matcha allt. Prova nedan i REPL, men försök lista ut i förväg vad som kommer att hända. Vad händer?

```
1 scala> val x = "urka"
2 scala> x match
3     case str if str.startsWith("g") => println("kanske gurka")
4     case vadsomhelst => println("ej gurka: " + vadsomhelst)
5 scala> val g = "gurka"
6 scala> g match
7     case str if str.startsWith("g") => println("kanske gurka")
8     case vadsomhelst => println("ej gurka: " + vadsomhelst)
```

b) *Fallgrop med små begynnelsebokstäver.* Innan du provar nedan i REPL, försök gissa vad som kommer att hända. Vad händer? Hur lyder varningarna och vad innebär de?

```
1 scala> val any: Any = "varken tomat eller gurka"
2 scala> case object Gurka
3 scala> case object tomat
4 scala> any match
5     case Gurka => println("gurka")
6     case tomat => println("tomat")
7     case _ => println("allt annat")
```

c) *Använd backticks för att tvinga fram match på konstant värde.* Det finns en utväg om man inte vill att kompilatorn ska skapa en ny lokal variabel: använd specialtecknet *backtick*, som skrivs ``` och kräver speciella tangentbordsstryck.<sup>2</sup> Gör om föregående uppgift men omgärda nu identifieraren `tomat` i `tomat`-case-grenen med backticks, så här:

```
case `tomat` => ...
```

**Uppgift 7.** *Matcha på innehåll i en Vector.* Kör nedan i REPL. Vad skrivs ut? Förklara vad som händer.

```
1 scala> val xss = Vector(Vector("hej"), Vector("på", "dej"), Vector("4", "x", "2"))
2 scala> xss.map(_ match
```

<sup>2</sup>Fråga någon om du inte hittar hur man gör backtick ``` på ditt tangentbord.

```

3  case Vector() => "tom"
4  case Vector(a) => a.reverse
5  case Vector(_, b) => b.reverse
6  case Seq(a, "x", b) => a + b
7  case _ => "ANNARS DETTA"
8  ).foreach(println)

```

**Uppgift 8.** Använda *Option* och *matcha* på värden som kanske saknas. Man behöver ofta skriva kod för att hantera värden som eventuellt saknas, t.ex. saknade telefonnummer i en persondatabas. Denna situation är så pass vanlig att många språk har speciellt stöd för saknande värden.

I Java<sup>3</sup> används värdet **null** för att indikera att en referens saknar värde. Man får då komma ihåg att testa om värdet saknas varje gång sådana värden ska behandlas, t.ex. med **if** (`ref != null`) { ... } **else** { ... }. Ett annat vanligt trick är att låta -1 indikera saknade positiva heltal, till exempel saknade index, som får behandlas med **if** (`i != -1`) { ... } **else** { ... }.

I Scala finns en speciell typ *Option* som möjliggör smidig och typsäker hantering av saknade värden. Om ett kanske saknat värde packas in i en *Option* (eng. *wrapped in an Option*), finns det i en speciell slags samling som bara kan innehålla *inget* eller *något* värde, och alltså har antingen storleken 0 eller 1.

a) Förklara vad som händer nedan.

```

1  scala> var kanske: Option[Int] = None
2  scala> kanske.size
3  scala> kanske = Some(42)
4  scala> kanske.size
5  scala> kanske.isEmpty
6  scala> kanske.isDefined
7  scala> def ökaOmFinns(opt: Option[Int]): Option[Int] = opt match
8           case Some(i) => Some(i + 1)
9           case None    => None
10 scala> val annanKanske = ökaOmFinns(kanske)
11 scala> def öka(i: Int) = i + 1
12 scala> val merKanske = kanske.map(öka)

```

b) Mönstermatchingen ovan är minst lika knölig som en **if**-sats, men tack vare att en *Option* är en slags (liten) samling finns det smidigare sätt. Förklara vad som händer nedan.

```

1  val meningen = Some(42)
2  val ejMeningen = Option.empty[Int]
3  meningen.map(_ + 1)
4  ejMeningen.map(_ + 1)
5  ejMeningen.map(_ + 1).orElse(Some("saknas")).foreach(println)
6  meningen.map(_ + 1).orElse(Some("saknas")).foreach(println)

```

c) *Samlingsmetoder som ger en Option*. Förklara för varje rad nedan vad som händer. En av raderna ger ett felmeddelande; vilken rad och vilket felmeddelande?

```

1  val xs = (42 to 84 by 5).toVector
2  val e = Vector.empty[Int]
3  xs.headOption
4  xs.headOption.get
5  xs.headOption.getOrElse(0)
6  xs.headOption.orElse(Some(0))
7  e.headOption

```

<sup>3</sup>Scala har också **null** men det behövs bara vid samverkan med Java-kod.

```

8 e.headOption.get
9 e.headOption.getOrElse(0)
10 e.headOption.orElse(Some(0))
11 Vector(xs, e, e, e)
12 Vector(xs, e, e, e).map(_._lastOption)
13 Vector(xs, e, e, e).map(_._lastOption).flatten
14 xs.lift(0)
15 xs.lift(1000)
16 e.lift(1000).getOrElse(0)
17 xs.find(_ > 50)
18 xs.find(_ < 42)
19 e.find(_ > 42).foreach(_ => println("HITTAT!"))

```

d) Vilka är fördelarna med Option jämfört med **null** eller -1 om man i sin kod glömmer hantera saknade värden?

**Uppgift 9.** *Kasta undantag.* Om man vill signalera att ett fel eller en onormal situation uppstått så kan man **kasta** (eng. *throw*) ett **undantag** (eng. *exception*). Då avbryts programmet direkt med ett felmeddelande, om man inte väljer att **fånga** (eng. *catch*) undantaget.

a) Vad händer nedan?

```

1 scala> throw new Exception("PANG!")
2 scala> java.lang. // Tryck TAB efter punkten
3 scala> throw new IllegalArgumentException("fel fel fel")
4 scala> val carola =
5     try
6         throw new Exception("stormvind!")
7         42
8     catch
9         case e: Throwable =>
10             println("Fångad av en " + e)
11             -1

```

b) Nämn ett par undantag som finns i paketet `java.lang` som du kan gissa vad de innebär och i vilka situationer de kastas.

c) Vilken typ har variabeln `carola` ovan? Vad hade typen blivit om `catch`-grenen hade returnerat en sträng i stället?

**Uppgift 10.** *Fånga undantag med `scala.util.Try`.* I paketet `scala.util` finns typen `Try` med stort T som är som en slags samling som kan innehålla antingen ett "lyckat" eller "misslyckat" värde. Om beräkningen av värdet lyckades och inga undantag kastas blir värdet inkapslat i en `Success`, annars blir undantaget inkapslat i en `Failure`. Man kan extrahera värdet, respektive undantaget, med mönstermatchning, men det är oftast smidigare att använda samlingsmetoderna `map` och `foreach`, i likhet med hur `Option` används. Det finns även en smidig metod `recover` på objekt av typen `Try` där man kan skicka med kod som körs om det uppstår en undantagssituation.

a) Förklara vad som händer nedan.

```

1 scala> def pang = throw new Exception("PANG!")
2 scala> import scala.util.{Try, Success, Failure}
3 scala> Try{pang}
4 scala> Try{pang}.recover{case e: Throwable => "desarmerad bomb: " + e}
5 scala> Try{"tyst"}.recover{case e: Throwable => "desarmerad bomb: " + e}
6 scala> def kanskePang = if math.random() > 0.5 then "tyst" else pang
7 scala> def kanskeOk = Try{kanskePang}
8 scala> val xs = Vector.fill(100)(kanskeOk)

```

```

9  scala> xs(13) match
10      case Success(x) => ":"
11      case Failure(e) => ":( " + e
12  scala> xs(13).isSuccess
13  scala> xs(13).isFailure
14  scala> xs.count(_.isFailure)
15  scala> xs.find(_.isFailure)
16  scala> val badOpt = xs.find(_.isFailure)
17  scala> val goodOpt = xs.find(_.isSuccess)
18  scala> badOpt
19  scala> badOpt.get
20  scala> badOpt.get.get
21  scala> badOpt.map(_.getOrElse("bomben desarmerad!")).get
22  scala> goodOpt.map(_.getOrElse("bomben desarmerad!")).get
23  scala> xs.map(_.getOrElse("bomben desarmerad!")).foreach(println)
24  scala> xs.map(_.toOption)
25  scala> xs.map(_.toOption).flatten
26  scala> xs.map(_.toOption).flatten.size

```

b) Vad har funktionen `pang` för returtyp?

c) Varför får funktionen `kanskePang` den härledda returtypen `String`?

### 6.2.2 Fördjupningsuppgifter; utmaningar

**Uppgift 11.** *Använda matchning eller dynamisk bindning?* Man kan åstadkomma urskiljningen av de ätbara grönsakerna i uppgift 3 med dynamisk bindning i stället för `match`.

a) Gör en ny variant av ditt program enligt nedan riktlinjer och spara den modifierade koden i filen `vegopoly.scala` och kompilera och kör.

- Ta bort predikatet `ärÄtvärd` i objektet `Main` och inför i stället en abstrakt metod `def ärÄtbar: Boolean` i traiten `Grönsak`.
- Inför konkreta `val`-medlemmar i respektive grönsak som definierar ätbarheten.
- Ändra i huvudprogrammet i enlighet med ovan ändringar så att `ärÄtvärd` anropas som en metod på de skördade grönsaksobjekten när de ätvärda ska filtreras ut.

b) Lägg till en ny grönsak `case class Broccoli` och definiera dess ätbarhet. Ändra i slump-funktionerna så att broccoli blir ovanligare än gurka.

c) Jämför lösningen med `match` i uppgift 3 och lösningen ovan med polymorfism. Vilka är för- och nackdelarna med respektive lösning? Diskutera två olika situationer på ett hypotetiskt företag som utvecklar mjukvara för jordbrukssektorn: 1) att uppsättningen grönsaker inte ändras särskilt ofta medan definitionerna av ätbarhet ändras väldigt ofta och 2) att uppsättningen grönsaker ändras väldigt ofta men att ätbarhetsdefinitionerna inte ändras särskilt ofta.

**Uppgift 12.** *Metoden equals.* Om man överskuggar den befintliga metoden `equals` så kommer metoden `==` att fungera annorlunda. Man kan då själv åstadkomma innehållslighet i stället för referenslikhet. Vi börjar att studera den befintliga `equals` med referenslikhet.

a) Vad händer nedan? Undersök parametertyp och returvärdestyp för `equals`.

```

1  scala> class Gurka(val vikt: Int, val ärÄtbar: Boolean)
2  scala> val g1 = new Gurka(42, true)
3  scala> val g2 = g1
4  scala> val g3 = new Gurka(42, true)
5  scala> g1 == g2

```

```
6 scala> g1 == g3
7 scala> g1.equals // tryck ENTER för att se funktionstyp
```

b) Rita minnessituationen efter rad 4.

c) *Överskugga metoderna equals och hashCode.*

*Bakgrund:* Det visar sig förvånande komplicerat att implementera innehållslighet med metoden equals så att den ger bra resultat under alla speciella omständigheter. Till exempel måste man även överskugga en metod vid namn hashCode om man överskuggar equals, eftersom dessa båda används gemensamt av effektivitetsskäl för att skapa den interna lagringen av objekten i vissa samlingar. Om man missar det kan objekt bli "osynliga" i hashCode-baserade samlingar – men mer om detta i senare kurser. Om objekten ingår i en öppen arvshierarki blir det också mer komplicerat; det är enklare om man har att göra med finala klasser. Dessutom krävs speciella hänsyn om klassen har en typparameter.

Definera klassen nedan i REPL med överskuggade equals och hashCode; den ärver inte något och är final.

```
// fungerar fint om klassen är final och inte ärver något
final class Gurka(val vikt: Int, val ärÄtbar: Boolean):
  override def equals(other: Any): Boolean = other match
    case that: Gurka => vikt == that.vikt && ärÄtbar == that.ärÄtbar
    case _ => false
  override def hashCode: Int = (vikt, ärÄtbar).## //förklaras sen
```

d) Vad händer nu nedan, där Gurka nu har en överskuggad equals med innehållslighet?

```
1 scala> val g1 = new Gurka(42, true)
2 scala> val g2 = g1
3 scala> val g3 = new Gurka(42, true)
4 scala> g1 == g2
5 scala> g1 == g3
```

e) Hur märker man ovan att den överskuggade equals medför att == nu ger innehållslighet? Jämför med deluppgift a.

I uppgift 18 får du prova på att följa det fullständiga receptet i 8 steg för att överskugga en equals enligt konstens alla regler. I efterföljande kurs kommer mer träning i att hantera innehållslighet och hash-koder. I Scala får man ett objekts hash-kod med metoden ##.<sup>4</sup>

**Uppgift 13. Polynom.** Med hjälp av koden nedan, kan man göra följande:

```
1 scala> import polynomial.*
2
3 scala> Const(1) * x
4 res0: polynomial.Term = x
5
6 scala> (x*5)^2
7 res1: polynomial.Prod = 25x^2
8
9 scala> Poly(x*(-5), y^4, (z^2)*3)
10 res2: polynomial.Poly = -5x + y^4 + 3z^2
```

a) Förklara vad som händer ovan genom att studera koden nedan<sup>5</sup>.

```
1 object polynomial:
2
```

<sup>4</sup>Om du är nyfiken på hash-koder, läs mer här: [en.wikipedia.org/wiki/Hash\\_function](https://en.wikipedia.org/wiki/Hash_function)

<sup>5</sup>Koden finns även här:

[github.com/lunduniversity/introprog/tree/master/compendium/examples/polynomial](https://github.com/lunduniversity/introprog/tree/master/compendium/examples/polynomial)

```

3  sealed trait Term:
4    def *(that: Term): Term
5
6  case class Const(value: BigDecimal) extends Term:
7
8    def toSilentString: String = this match
9      case Const.One          => ""
10     case Const.MinusOne     => "-"
11     case _                  => value.toString
12
13    override def toString = value.toString
14
15    override def *(that: Term): Term = that match
16      case Const(d)          => Const(d * value)
17      case v: Var             => Prod(this, Set(v))
18      case Prod(c, vs)       => Prod(Const(c.value * value), vs)
19
20    def *(d: BigDecimal): Const = Const(d * value)
21
22    def ^(e: Int): Const = Const(value.pow(e))
23
24
25  object Const:
26    final val Zero      = Const(BigDecimal(0))
27    final val One       = Const(BigDecimal(1))
28    final val MinusOne  = Const(BigDecimal(-1))
29
30  case class Var(name: Char, exp: Int = 1) extends Term:
31
32    private def silentExpString: String =
33      if exp == 1 then "" else "^"+exp.toString
34
35    override def toString = s"$name$silentExpString"
36
37    def ^(e: Int): Var = Var(name, e * exp)
38
39    def *(c: BigDecimal) = Prod(Const(c), Set(this))
40
41    override def *(that: Term): Term = that match
42      case c: Const => Prod(c, Set(this))
43
44      case v: Var =>
45        if v.name == name then Var(name, v.exp + exp)
46        else Prod(Const.One, Set(this, v))
47
48      case p: Prod => p * this
49
50
51  object Var:
52
53    def apply(d: BigDecimal, name: Char): Prod =
54      Prod(Const(d), Set(Var(name)))
55
56    def apply(d: BigDecimal, name: Char, exp: Int): Prod =
57      Prod(Const(d), Set(Var(name, exp)))
58
59    def addExp(v1: Var, v2: Var): Var = Var(v1.name, v1.exp + v2.exp)
60
61    def multiply(v1: Var, vs: Set[Var]): Set[Var] =

```

```

62     if !vs.contains(v1) then vs + v1
63     else vs.map(v2 => if v1.name == v2.name then addExp(v1, v2) else v2)
64
65     def multiply(vs1: Set[Var], vs2: Set[Var]): Set[Var] =
66         var result = vs2
67         vs1.foreach{ v1 => result = multiply(v1, result) }
68         result
69
70
71     case class Prod(const: Const, vars: Set[Var]) extends Term :
72
73         override def toString = s"${const.toSilentString}${vars.mkString}"
74
75         override def *(that: Term): Term = that match
76             case Const(d) => Prod(Const(d * const.value), vars)
77
78             case v: Var => Prod(const, Var.multiply(v, vars))
79
80             case Prod(Const(d), vs) =>
81                 Prod(Const(const.value * d), Var.multiply(vs, vars))
82
83         def ^(e: Int) = Prod(const ^ e, vars.map(_ ^ e))
84
85     case class Poly(xs: Set[Term]):
86         override def toString = xs.mkString(" + ")
87
88     object Poly:
89         def apply(ts: Term*) : Poly = Poly(ts.toSet)
90
91     val (x, y, z, s, t) = (Var('x'), Var('y'), Var('z'), Var('s'), Var('t'))

```

b) Bygg vidare på **object** polynomial och implementera addition mellan olika termer.

**Uppgift 14.** *Option som en samling.* Studera dokumentationen för Option här och se om du känner igen några av metoderna som också finns på samlingen Vector:

[www.scala-lang.org/api/current/scala/Option.html](http://www.scala-lang.org/api/current/scala/Option.html)

Förklara hur metoden contains på en Option fungerar med hjälp av dokumentationens exempel.

**Uppgift 15.** *Fånga undantag med catch i Java och Scala.* Gör motsvarande program i Scala som visas i uppgift 12, men utnyttja att Scalas **try-catch** är ett uttryck. Kompilera och kör och testa så att de ur användarens synvinkel fungerar precis på samma sätt. Notera de viktigaste skillnaderna mellan de båda programmen.

**Uppgift 16.** *Polynom, fortsättning: reducering.* Bygg vidare på **object** polynomial i uppgift 13 på sidan 218 och implementera metoden **def** reduce: Poly i case-klassen Poly som förenklar polynom om flera Prod-termer kan adderas.

**Uppgift 17.** *Typsäker innehållstest med metoden ===.* Metoderna equals och == tillåter jämförelse med vad som helst. Ibland vill man ha en typsäker innehållsjämförelse som bara tillåter jämförelse av objekt av en mer specifik typ och ger kompileringsfel annars. Man brukar då definiera en metod === som har en parameter that som har en så specifik typ som önskas. Inför nedan abstrakta metod === i traiten polynomial.Term i uppgift 13 på sidan 218 och överskugga den sedan i alla subklasser till Term. Testa så att du får kompileringsfel



om du försöker jämföra en `Term` med något helt annat, t.ex. en `String` eller `Vector`.

```
def ==(that: Term): Boolean
```

**Uppgift 18.** Överskugga `equals` med innehållslikhet även för icke-finala klasser. Nedan visas delar av klassen `Complex` som representerar ett komplext tal med realdel och imaginärdel. I stället för att, som man ofta gör i Scala, använda en case-klass och en `equals`-metod som automatiskt ger innehållslikhet, ska du träna på att implementera en egen `equals`.

```
class Complex(val re: Double, val im: Double):
  def abs: Double = math.hypot(re, im)
  override def toString = s"Complex($re, $im)"
  def canEqual(other: Any): Boolean = ???
  override def hashCode: Int = ???
  override def equals(other: Any): Boolean = ???

case object Complex:
  def apply(re: Double, im: Double): Complex = new Complex(re, im)
```

Följ detta **recept**<sup>6</sup> i 8 steg för att överskugga `equals` med innehållslikhet som fungerar även för klasser som inte är **final**:

1. Inför denna metod: `def canEqual(other: Any): Boolean`  
Observera att typen på parametern ska vara `Any`. Om detta görs i en subklass till en klass som redan implementerat `canEqual`, behövs även **override**.
2. Metoden `canEqual` ska ge **true** om `other` är av samma typ som `this`, alltså till exempel:  
`def canEqual(other: Any): Boolean = other.isInstanceOf[Complex]`
3. Inför metoden `equals` och var noga med att parametern har typen `Any`:  
**override def** `equals(other: Any): Boolean`
4. Implementera metoden `equals` med ett match-uttryck som börjar så här:  
`other match`
5. Match-uttrycket ska ha två grenar. Den första grenen ska ha ett typat mönster för den klass som ska jämföras:  
**case** `that: Complex =>`
6. Om du implementerar `equals` i den klass som inför `canEqual`, börja uttrycket med:  
`(that canEqual this) &&`  
och skapa därefter en fortsättning som baseras på innehållet i klassen, till exempel:  
`this.re == that.re && this.im == that.im`  
Om du överskuggar en *annan* `equals` än den standard-`equals` som finns i `AnyRef`, vill du förmodligen börja det logiska uttrycket med att anropa superklassens `equals`-metod:  
**super**.`equals(that) &&` men du får fundera noga på vad likhet av underklasser egentligen ska innebära i ditt speciella fall.
7. Den andra grenen i matchningen ska vara: **case** `_ => false`

<sup>6</sup>Detta recept bygger på <http://www.artima.com/pinsled/object-equality.html>



8. Överskugga hashCode, till exempel genom att göra en tupel av innehållet i klassen och anropa metoden ## på tupeln så får du i en bra hashCode:

```
override def hashCode: Int = (re, im).##
```

**Uppgift 19.** Överskugga equals vid arv. Bygg vidare på exemplet nedan och överskugga equals vid arv, genom att följa receptet i uppgift 18.

```
trait Number:
  override def equals(other: Any): Boolean = ???

class Complex(re: Double, im: Double) extends Number:
  override def equals(other: Any): Boolean = ???

class Rational(numerator: Int, denominator: Int) extends Number:
  override def equals(other: Any): Boolean = ???
```

**Uppgift 20.** Speciella matchningar. Läs om användning av speciella matchningar här: [dotty.epfl.ch/docs/reference/changed-features/vararg-splices.html](http://dotty.epfl.ch/docs/reference/changed-features/vararg-splices.html)

- a) Prova variabelbinding med @ i en matchning i REPL.
- b) Prova sekvensmönster med \_ och \_\* i en matching i REPL.

**Uppgift 21.** Extraktorer. Läs mer om extraktorer här: [dotty.epfl.ch/docs/reference/changed-features/pattern-matching.html](http://dotty.epfl.ch/docs/reference/changed-features/pattern-matching.html)

Skapa ditt eget extraktor-objekt för http-adresser som i t.ex.:

http://my.host.domain/path/to/this

extraherar my.host.domain och path/to/this med metoden unapply och testa i en matchning.

**Uppgift 22.** Polynom, fortsättning: polynomdivision. Implementera polynomdivision på lämpligt sätt genom att bygga vidare på **object** polynomial i uppgift 13 på sidan 218. Läs mer om polynomdivision här: [sv.wikipedia.org/wiki/Polynomdivision](http://sv.wikipedia.org/wiki/Polynomdivision)

## 6.3 Laboration: blockbattle1

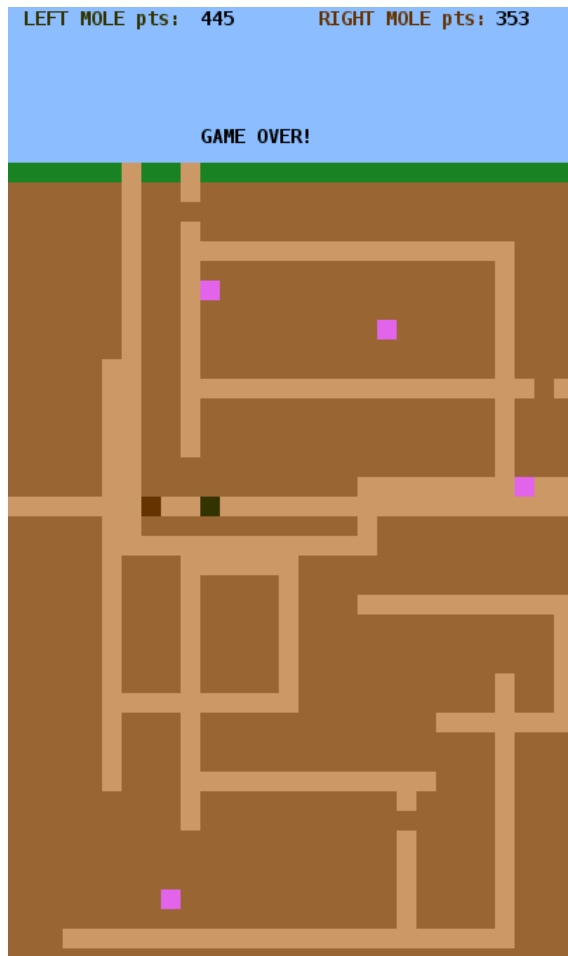
### Mål

- ☐ Kunna förklara skillnader och likheter mellan ett singelobjekt och objekt som är instanser av klasser.
- ☐ Kunna förklara skillnaden mellan förändringsbara och oföränderliga objekt.
- ☐ Kunna definiera och instansiera klasser och case-klasser, samt kunna beskriva när en case-klass är lämpligast och ge några exempel på vad en sådan erbjuder utöver en vanlig klass.
- ☐ Kunna skapa och använda klasser vars instanser innehåller referenser till andra instanser (aggregering).
- ☐ Förstå innebörden av instansreferensen `this`.
- ☐ Kunna skapa enkla match-uttryck.

### Förberedelser

- ☐ Gör övning `classes` i avsnitt 5.2, speciellt uppgift 6.
- ☐ Gör övning `patterns` i avsnitt 6.2.
- ☐ Läs igenom hela laborationen och planera ditt arbete.
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).

### 6.3.1 Bakgrund



**Figur 6.1:** En duell om blockmaskar mellan två lundensiska blockmullvader fångade på bild under intensivt grävanade.

Under denna laboration ska du träna på att deklarera klasser och skapa flera instanser av samma klass. Du tränar även på att bygga ett större program från grunden.

Du ska utveckla ett spel för två spelare som sitter vid samma tangentbord, där den vänstra spelaren styr en blockmullvad med tangenterna A,S,D,W, och den högra spelaren styr en annan blockmullvad med piltangenterna.

I bilden till vänster ser du hur spelet kan se ut. Det finns en ljusbrun och en mörkbrun mullvad. Poängräkningen visas överst i himlen. Det finns fyra rosa blockmaskar (se uppgift 12 i laboration blockmole) som mullvadarna tävlar om att försöka fånga. När en blockmask teleporterar sig till en ny slumpmässig position lämnar den jord efter sig. När en mullvad gräver sig upp till gräsytan blir det hål i gräset. Det ger poäng att gräva tunnlar och att fånga blockmaskar.

Du bestämmer själv hur poängsättningen ska ske och kriteriet för när spelet är slut etc.

### 6.3.2 Obligatoriska krav

Följande funktionella krav ska uppfyllas av ditt program:

- ☐ Varje mullvad rör sig i sin aktuella riktning tills användaren ändrar riktning genom att trycka på "sin" motsvarande knapp, t.ex. W eller pil-upp.
- ☐ Då en mullvad går i mörkbrun jord ska ljusbruna tunnlar grävas.
- ☐ Då en blockmullvad når fönstrets kant eller himlen ska dess riktning reverseras.
- ☐ Det ska ge poäng att gräva tunnlar.
- ☐ Varje spelares poäng ska visas under spelets gång.
- ☐ Ett spel ska avslutas och *Game over* visas när något valfritt kriterium uppfyllts.

Din kod ska utformas enligt dessa design-krav:


- ☐ Ett Game skapas i huvudprogrammet med metoden `start` som kör igång spelet.
- ☐ Konstanter ska namnges och placeras i lämpligt kompanjonsobjekt.
- ☐ Varje klass med ev. tillhörande kompanjonsobjekt ska finnas i en egen kodfil och tillhöra paketet `blockbattle`.
- ☐ Du ska utgå från klasserna som du implementerat i uppgift 6 i övning `classes`.
- ☐ Klassen `BlockWindow` omvandlar till interna fönsterkoordinater. Övriga klasser ska använda block-koordinater.

### 6.3.3 Valbara krav – välj minst ett

Du ska implementera minst ett (gärna flera) av dessa krav:

- ☐ Det ska finnas lagom många blockmaskar (se labb blockmole uppg. 12, sid. 151).
- ☐ Blockmullvadarna ska även ha ett attribut som representerar hälsan, t.ex. ett numeriskt värde mellan 0 och 100. Hälsan ska försvagas något när man gräver tunnlar. Hälsan ska synas i spelfönstret, t.ex. som en sekvens med röda block i himlen som indikerar andelen av maxhälsan för resp. spelare.
- ☐ Att springa på gräset ska påverka poäng och/eller hälsa.
- ☐ Att fånga blockmask ska påverka poäng och/eller hälsa.
- ☐ Det ska finnas gula blockdiamanter som ger många poäng om man tar dem först.
- ☐ Det ska vid spelstart gå att välja namn på respektive blockmullvad och namnet ska synas i spelet vid poängutskriften.
- ☐ Det ska gå fortare att gå i gångar jämfört med att gräva i jord.
- ☐ Om en blockmullvad fångar en blockmask ska dess grävastighet öka.
- ☐ Om en blockmullvad krockar med en annan blockmullvad ska något hända, t.ex. att dess riktning reverseras.
- ☐ Visa *highscore* vid *Game Over*. Highscore sparas med `introprog.IO` i en fil som skapas om den inte finns annars läses in vid uppstart om den finns och uppdateras vid behov. Spara hela highscore-listan eller bara högsta poäng hittills.

### 6.3.4 Förebredelser inför redovisningen

✓  Innan du redovisar din implementation ska du muntligt kunna redogöra för följande:

- ☐ Studera någon annans spel och ge din kamrat minst ett tips om hur kodens läsbarhet kan förbättras. Skriv ner dina tips och beskriv dem vid redovisningen.
- ☐ Beskriv vilka åtgärder du gjort för att din kod ska vara lätt att läsa och förstå.
- ☐ Beskriv hur du stegvis utvecklat ditt program från enklare till mer avancerad funktionalitet, samt vilka buggar du upptäckt och fixat.
- ☐ Beskriv vilket eller vilka valfria krav som din implementation uppfyller.
- ☐ Beskriv hur du hade behövt ändra i klassen `Mole` för att det ska gå att skriva  
`new Mole().move().move().reverseDir().move()`

### 6.3.5 Tips och förslag

1. **Många små steg.** Kör kompilering under ändringsbevakning med `--watch` i ett eget terminalfönster, så att du vid varje ändring kan rätta ev. kompileringsfel. Kör och testa ditt program ett annat terminalfönster.
2. **Inför bra namn.** Din kod blir lättare att läsa och ändra i om du hittar på bra namn på medlemmar och lägger dem på lämpligt ställe. T.ex. kan du samla globala spel-konstanter i kompanjonsobjektet till klassen `Game`. Du kan bygga vidare på nedan kod och lägga till medlemmar allteftersom du upptäcker att de behövs. Nedan finns exempelvis en funktion som ger bakgrundsfärgen för en viss y-koordinat, vilken är användbar när du ska återställa bakgrunden efter att en mullvad har flyttat sig.

```
package blockbattle

object Game:
  val windowSize = (30, 50)
  val windowTitle = "EPIC BLOCK BATTLE"
  val blockSize = 14
  val skyRange    = 0 to 7
```

```

val grassRange = 8 to 8
object Color { ??? }
/** Used with the different ranges and eraseBlocks */
def backgroundColorAtDepth(y: Int): java.awt.Color = ???

class Game(
  val leftPlayerName: String = "LEFT",
  val rightPlayerName: String = "RIGHT"
):
  import Game.* // direkt tillgång till namn på medlemmar i kompanjon

  val window = new BlockWindow(windowSize, windowTitle, blockSize)
  val leftMole: Mole = ???
  val rightMole: Mole = ???

  def drawWorld(): Unit = ???

  /** Use to erase old points, e.g updated score */
  def eraseBlocks(x1: Int, y1: Int, x2: Int, y2: Int): Unit = ???

  def update(mole: Mole): Unit = ??? // update, draw new, erase old

  def gameLoop(): Unit = ???

  def start(): Unit =
    println("Start digging!")
    println(s"$leftPlayerName ${leftMole.keyControl}")
    println(s"$rightPlayerName ${rightMole.keyControl}")
    drawWorld()
    gameLoop()

```

3. **Dela upp din kod i funktioner.** Din kod blir lättare att läsa och ändra i om du delar upp den i många små funktioner med bra namn. I Game-klassen ovan finns exempel på några användbara funktioner. Allteftersom du utvidgar ditt program kan du lägga till fler funktioner som t.ex. heter `showPoints`, `gameOver`, etc.
4. **Tänk igenom den övergripande strukturen.** Programmet du ska skriva i denna laboration är större än det du gjort tidigare. Det är därför viktigt att tänka igenom strukturen på ditt program, vilka klasser som har hand om vad och hur de samarbetar. Diskutera gärna med handledare om du är osäker på hur de koddelar du utvecklat i föregående veckas övning 6, klasserna `Pos`, `KeyControl`, `Mole` och `BlockWindow`, är tänkta att samverka. Var noga med att testa så de olika klasserna och deras metoder fungerar var för sig.
5. **Utformning av `gameLoop()`.** I ett spel behövs en s.k. spel-loop (eng. *game loop*) som upprepar den kod som ska köras vid varje ny skärmbild, ofta kallad *frame*. I varje runda i spel-loopen sker uppdatering av data och ritning i spelfönstret, samt en lämplig fördröjning. En skiss på en typisk spel-loop visas nedan:

```

var quit = false
val delayMillis = 80

def gameLoop(): Unit =
  while !quit do
    val t0 = System.currentTimeMillis
    handleEvents() // ändrar riktning vid tangenttryck etc.
    update(leftMole) // flyttar, ritar, suddar, etc.
    update(rightMole)

```

```

    val elapsedMillis = (System.currentTimeMillis - t0).toInt
    Thread.sleep((delayMillis - elapsedMillis) max 0)
  end while
end gameLoop

```

6. **Hantering av händelser.** Ett BlockWindow, som du implementerade i uppgift 6 i övning classes, kan via anrop av nextEvent ge KeyPressed(key) vid knapptryck och WindowClosed vid fönsterstängning. Om ingen händelse finns att behandla returneras Undefined. Använd en loop som betar av alla händelser tills Undefined påträffas, enligt nedan:

```

def handleEvents(): Unit =
  var e = window.nextEvent()
  while e != BlockWindow.Event.Undefined do
    e match
      case BlockWindow.Event.KeyPressed(key) =>
        ??? // ändra riktning på resp. mullvad

      case BlockWindow.Event.WindowClosed =>
        ??? // avsluta spel-loopen

    e = window.nextEvent()
  end while
end handleEvents

```

7. **Flimmerfri grafik.** För att minska mängden flimmer (eng. *flicker*) är det bäst att i varje iteration i spel-loopen (1) bara rita om det som ändrats för att minimera tiden som spenderas på att rita, och (2) vid ändringar rita nya delar före att gamla delar raderas. För att slippa mullvadsflimmer kan du ”rita först – sudda sen” enligt nedan.<sup>7</sup>

```

window.setBlock(mole.nextPos, mole.color) // draw new
window.setBlock(mole.pos, Color.tunnel)   // erase old
mole.move()                               // update

```

<sup>7</sup>Inom spelutveckling använder man oftast istället så kallad *double buffering* (eller till och med *triple buffering*) för att få helt flimmerfri grafik. Det ligger dock bortom kursen och stöds inte av PixelWindow.

## Kapitel 7

# Sekvenser och enumerationer

Begrepp som ingår i denna veckas studier:

- ☐ översikt av Scalas samlingsbibliotek och samlingsmetoder
- ☐ klasshierarkin i `scala.collection`
- ☐ `Iterable`
- ☐ `Seq`
- ☐ `List`
- ☐ `ListBuffer`
- ☐ `ArrayBuffer`
- ☐ `WrappedArray`
- ☐ sekvensalgoritm
- ☐ algoritm: SEQ-COPY
- ☐ in-place vs copy
- ☐ algoritm: SEQ-REVERSE
- ☐ registrering
- ☐ algoritm: SEQ-REGISTER
- ☐ linjärsökning
- ☐ algoritm: LINEAR-SEARCH
- ☐ tidskomplexitet
- ☐ minneskomplexitet
- ☐ översikt strängmetoder
- ☐ `StringBuilder`
- ☐ ordning
- ☐ inbyggda sökmetoder
- ☐ `find`
- ☐ `indexOf`
- ☐ `indexWhere`
- ☐ inbyggda sorteringsmetoder
- ☐ `sorted`
- ☐ `sortWith`
- ☐ `sortBy`
- ☐ repeterade parametrar

## 7.1 Teori

### 7.1.1 Vad är en sekvens?

- En sekvens är en **följd av element** som
  - har **ordningsnummer** (t.ex. numrerade från noll)
  - är av en viss **typ** (t.ex. heltal).
- En sekvens kan innehålla flera element som är lika.
- En sekvens kan vara **tom** och har då längden noll.
- Exempel på en icke-tom sekvens med dubletter:

```
scala> val xs = Vector(42, 0, 42, -9, 0, 5)
xs: scala.collection.immutable.Vector[Int] =
  Vector(42, 0, 42, -9, 0, 5)
```

- **Indexering** ger ett element via dess ordningsnummer:

```
scala> xs(2)
res0: Int = 42

scala> xs.apply(2)
res1: Int = 42
```

---

### 7.1.2 Exempel: En sträng är en sekvens av tecken

```
scala> "haj po daj"
```

Längd? Vad ligger på första platsen? Elementtyp? Dubletter?

```
scala> "haj po daj".length
res1: Int = 10

scala> "haj po daj".apply(0)
res2: Char = h

scala> "haj po daj"(0)
res3: Char = h

scala> "haj po daj".distinct
res4: String = haj pod
```

---

### 7.1.3 Iterera över element i en sekvens

- Att **iterera** (eng. *iterate*), ä.k. traversera (eng. *traverse*), innebär att **gå igenom** och behandla element i en samling.
- Exempel på iterering med foreach, map, **for**:

```
scala> val xs = Vector(1,2,3)
val xs: Vector[Int] = Vector(1, 2, 3)

scala> xs.foreach(x => println(x + 1))
```



```

2
3
4

scala> xs.map(_ + 1)
val res0: Vector[Int] = Vector(2, 3, 4)

scala> for x <- xs yield x - 1
val res1: Vector[Int] = Vector(0, 1, 2)

```

### 7.1.4 Lägg till i början och i slutet av en sekvens

- Med metoderna `+` och `:+` kan du skapa en ny sekvens med nya element tillagda i början resp. i slutet.
- Minnesregel: **"Colon on the collection side"**

```

scala> val xs = Vector(1,2,3)
scala> 42 +: xs           // ger ny Vector(42, 1, 2, 3)
scala> xs :+ 42           // ger ny Vector(1, 2, 3, 42)

```

- Semantik: operatornotation med operatorer som **slutar med kolon** är **högerassociativa**
- Anropet `42 +: xs` skrivs av kompilatorn om till `xs.+: (42)`

```

1  scala> xs.+: (42)
2  res4: scala.collection.immutable.Vector[Int] = Vector(42, 1, 2, 3)
3

```

- Konkaterering (sammanfogning) av sekvenser: `xs ++ ys`

### 7.1.5 Egenskaper hos några sekvenssamlingar i Scala

- Vector
  - **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
  - Allsidig prestanda: **bra till det mesta**.
- List
  - **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
  - Snabb indexering & uppdatering **i början**.
  - Smidig & snabb vid **rekursiva** algoritmer.
  - Långsam vid upprepad **indexering** på godtyckliga ställen.
- ArrayBuffer
  - **Föränderlig: snabb indexering & uppdatering**.
  - Kan **ändra storlek** efter allokering. Snabb att indexera överallt.
- ListBuffer
  - **Föränderlig**: snabb indexering & uppdatering **i början**.
  - Snabb om du bygger upp sekvens genom många tillägg i början.

- Array eller `scala.collection.mutable.ArraySeq`
  - **Föränderlig: snabb indexering & uppdatering.**
  - Kan **ej ändra storlek**; storlek ges vid allokering.
  - Har särställning i JVM: ger snabb allokering och access.

### 7.1.6 Vilken sekvenssamling ska jag välja?

- Välj Vector om ...
  - a) du vill ha oföränderlighet: **val** `xs = Vector[Int](1,2,3)`
  - b) du behöver föränderlighet (notera **var**):  
**var** `xs = Vector.empty[Int]`
  - c) du ännu inte vet vilken sekvenssamling som är bäst; du kan alltid ändra efter att du mätt prestanda och kollat flaskhalsar vid upprepade körningar.
- Välj List om ...
 

du har en **rekursiv** sekvensalgoritm och/eller **mestadels jobbar i början**.
- Välj ArrayBuffer om ...
 

det behövs av prestandaskäl och du **inte** vet storlek vid allokering:  
**val** `xs = scala.collection.mutable.ArrayBuffer.empty[Int]`
- Välj ListBuffer om ...
 

det behövs av prestandaskäl och du bara behöver lägga till i början:  
**val** `xs = scala.collection.mutable.ListBuffer.empty[Int]`
- Välj Array eller ArraySeq om ...
 

det verkligen behövs av prestandaskäl och du **vet** storlek vid allokering:  
**val** `xs = Array.fill(initSize)(initValue)`

### 7.1.7 Några konstigheter med Array

- **Referenslikhet** (och inte innehållslikhet):

```
scala> Vector(1,2,3) == Vector(1,2,3) //innehållslikhet
val res0: Boolean = true

scala> Array(1,2,3) == Array(1,2,3) // referenslikhet
val res1: Boolean = false // aaargh!!
```

Notera: Metoden `==` mellan två `ArraySeq` ger **innehållslikhet**.

- Special-syntax för allokering **utan** explicit initialisering:  
**val** `xs = new Array[String](1000)` // 1000 null-referenser
- Fungerar inte lika bra med generiska typer:

```
scala> def box[T](x: T) = Vector[T](x) //funkar fint

scala> def abox[T](x: T) = Array[T](x)
error: No ClassTag available for T
```

### 7.1.8 Oföränderlig eller förändringsbar?

- **Oföränderlig**: Kan ej ändra elementreferenserna, men effektiv på att skapa kopia som är (delvis) förändrad **Vector** eller **List**
  - **Förändringsbar**: kan ändra elementreferenserna
    - Kan **ej ändra storlek** efter allokering:  
**Array** eller **ArraySeq**: indexera och uppdatera varsomhelst
    - Kan även ändra storlek efter allokering:  
**ArrayBuffer** eller **ListBuffer**
  - **Ofta funkar oföränderlig sekvenssamling utmärkt**, men om man **efter prestandamätning** upptäcker en flaskhals kan man ändra från **Vector** till t.ex. **ArrayBuffer**.
- 

### 7.1.9 Vad är en sekvensalgoritm?

- En algoritm är en stegvis beskrivning av lösningen på ett problem.
- En **sekvensalgoritm** är en algoritm där **element i sekvens** utgör en viktig del av **problembeskrivningen** och/eller **lösningen**.
- Exempelproblem: sortera en sekvens av personer efter deras ålder.
- **Sju** ofta återkommande programmeringsproblem som löses med en sekvensalgoritm:
  - **Kopiering** av alla element i en sekvens till en **ny** sekvens
  - **Uppdatering** av sekvensen: ta bort, lägga till, ändra **enskilda** element
  - **Transformerings**: applicera en **funktion** på **alla** element
  - **Filtrering**: urval av vissa element som uppfyller ett **villkor**
  - **Sökning** efter ett element som uppfyller ett **sökkriterium**
  - **Sortering** enligt någon **ordning**
  - **Registrering** kategorisera eller **räkna element** med vissa egenskaper

## KUT FSSR

---

### 7.1.10 Använda färdiga sekvenssamlingsmetoder

- Ofta kan man implementera sekvensalgoritmer genom anrop av en eller flera **färdiga** metoder.
  - Dessa färdiga metoder är **optimerade och vältestade** och är att föredra om möjligt.
  - Studera Scalas api-dokumentation och kursens quickref för att se vad man kan göra med färdiga metoder.  
<https://docs.scala-lang.org/overviews/collections-2.13/introduction.html>
  - Det är **lärorikt** att **"uppfinna hjulet"** och implementera några sekvensalgoritmer **själv** för bättre förståelse, även om de redan finns färdiga i Scalas samlingsbibliotek.
-

### 7.1.11 Några användbara samlingsmetoder vid implementation av sekvensalgoritmer

<code>xs.map(f)</code>	transformering, motsv. <b>for</b> <code>x &lt;- xs</code> <b>yield</b> <code>f(x)</code>
<code>xs.map(x =&gt; x)</code>	kopiering, motsv. <b>for</b> <code>x &lt;- xs</code> <b>yield</b> <code>x</code>
<code>xs.filter(p)</code>	filtrering, ta med <code>x</code> om <code>p(x)</code>
<code>xs.filterNot(p)</code>	filtrering, ta med <code>x</code> om <code>!p(x)</code>
<code>xs.distinct</code>	filtrering, ta bort dubletter
<code>xs.take(n)</code>	ny sekvens med de första <code>n</code> elements, resten skippade
<code>xs.drop(n)</code>	ny sekvens där de första <code>n</code> elements är skippade
<code>xs.takeWhile(p)</code>	filtrera, ta med i början så länge <code>p(x)</code>
<code>xs.dropWhile(p)</code>	filtrera, hoppa i början så länge <code>p(x)</code>
<code>xs.find(p)</code>	sök framifrån efter första element <code>x</code> där <code>p(x)</code> är sant
<code>xs.indexOf(x)</code>	sök framifrån efter index för element som är samma som <code>x</code>
<code>xs.lastIndexOf(x)</code>	sök bakifrån efter index för element som är samma som <code>x</code>
<code>xs.sorted</code>	sortera med inbyggd (implicit given) ordning
<code>xs.sorted.reverse</code>	sortera i omvänd ordning
<code>xs.sortBy(f)</code>	sortera i ordning enligt <code>f(x)</code>
<code>xs.sortWith(lt)</code>	sortera enligt "less than"-funktionen <code>lt</code> : <code>(A, A) =&gt; Boolean</code>
<code>xs.count(p)</code>	räkna antalet element där <code>p(x)</code> är sant

Lär dig fler smidiga metoder i [quickref](#)

### 7.1.12 Uppdaterad sekvens med kraftfulla metoden patch

Metoden `patch` kan användas så: `xs.patch(fromPos, ys, nbrReplaced)` för att skapa en **ny** sekvens där **ett** eller **flera** element i `xs` är...

- utbytta (eng. *replaced*)
- borttaga (eng. *removed*)
- tillagda (eng. *inserted*)

.. med nya element ur `ys`

```

1 scala> val xs = Vector(1,2,3)
2
3 scala> xs.patch(2, Vector(-1), 1)    // replaced one elem
4 res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, -1)
5
6 scala> xs.patch(1, Vector(42), 0)    // inserted one elem
7 res1: scala.collection.immutable.Vector[Int] = Vector(1, 42, 2, 3)
8
9 scala> xs.patch(0, Vector(), 2)      // removed two elems
10 res2: scala.collection.immutable.Vector[Int] = Vector(3)

```

### 7.1.13 Använda for-uttryck för filtrering med hjälp av gard

I ett `for`-uttryck kan man ha en **gard** (eng. *guard*) i form av ett booleskt uttryck efter nyckelordet **if**. Då kommer uttrycket efter **yield** bara göras om `gard`-uttrycket är sant.

Syntaxen är så här: (parenteser behövs ej runt `gard`-uttrycket)

```
for x <- xs if uttryck1 yield uttryck2
```

Exempel:

```
scala> val udda = for x <- 1 to 6 if x % 2 == 1 yield x
```

udda blir Vector(1, 3, 5)

#### 7.1.14 Använda samlingsmetoden filter för filtrering

Alla samlingar i `scala.collection` har metoden `filter`. Den har ett predikat som parameter `p: T => Boolean` och ger en ny samling med de element för vilka predikatet är sant.

```
xs.filter(p)
```

Exempel: Antag att `xs` är `(1 to 6).toVector`

```
xs.filter(_ % 2 == 1)
```

uttryckets resultat blir Vector(1, 3, 5), vilket motsvarar:

```
for x <- xs if x % 2 == 1 yield x
```

I själva verket skriver Scala-kompilatorn om `for`-uttryck med `guard` till anrop av metoden `filter` före kodgenerering sker.

#### 7.1.15 Vanliga sekvensproblem som funktionshuvuden

Indata och utdata för några vanliga sekvensproblem:

```
def copy(xs: Vector[Int]): Vector[Int] = ???
```

```
def filter(xs: Vector[Int], p: Int => Boolean): Vector[Int] = ???
```

```
def findIndices(xs: Vector[Int], p: Int => Boolean): Vector[Int] = ???
```

```
def sort(xs: Vector[Int]): Vector[Int] = ???
```

```
def freq(xs: Vector[Int]): Vector[(Int, Int)] = ??? // (heltal, frekvens)
```

Övning: Hur implementera dessa med `for`-uttryck och/eller färdiga samlingsmetoder?

**Tips:** För `sort` & `freq` se `sorted`, `distinct`, `count` i [quickref](#)

#### 7.1.16 Implementation av sekvensproblem med `for`-uttryck och/eller färdiga samlingsmetoder

```
def copy(xs: Vector[Int]): Vector[Int] = for x <- xs yield x
```

```
def filter(xs: Vector[Int], p: Int => Boolean): Vector[Int] =
  for x <- xs if p(x) yield x
```

```
def findIndices(xs: Vector[Int], p: Int => Boolean): Vector[Int] =
```

```
(for i <- xs.indices if p(xs(i)) yield i).toVector

def sort(xs: Vector[Int]): Vector[Int] = xs.sorted // mer om sortering sen

def freq(xs: Vector[Int]): Vector[(Int, Int)] = // mer om registrering snart
  for x <- xs.distinct yield x -> xs.count(_ == x)
```

Övning: Hur implementera dessa med map och filter och/eller andra färdiga samlingsmetoder?

### 7.1.17 Implementation av sekvensproblem med map, filter

```
def copy(xs: Vector[Int]): Vector[Int] = xs.map(x => x)

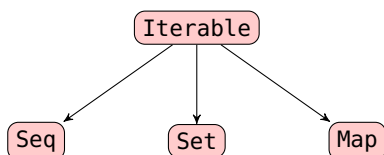
def filter(xs: Vector[Int], p: Int => Boolean): Vector[Int] = xs.filter(p)

def findIndices(xs: Vector[Int], p: Int => Boolean): Vector[Int] =
  xs.indices.filter(i => p(xs(i))).toVector

def sort(xs: Vector[Int]): Vector[Int] = xs.sorted // mer om sortering sen

def freq(xs: Vector[Int]): Vector[(Int, Int)] = // mer om registrering snart
  xs.distinct.map(x => x -> xs.count(_ == x))
```

### 7.1.18 Hierarki av samlings typer i scala.collection v2.13



Iterable har metoder som är implementerade med hjälp av:

```
def foreach[U](f: Elem => U): Unit
def iterator: Iterator[A]
```

Seq: ordnade i sekvens  
 Set: unika element  
 Map: par av (nyckel, värde)

Samlingen **Vector** är en Seq som är en Iterable.  
 De konkreta samlingsarna är uppdelade i dessa paket:

```
scala.collection.immutable
scala.collection.mutable
(undantag: primitiva scala.Array)
```

där flera är **automatiskt** importerade  
 som **måste importeras** explicit

### 7.1.19 Lämna det öppet: använd Seq

Typen **collection.immutable.Seq** är supertyp till alla sekvenssamlingsarna i `collection.immutable`.  
 Exempel: kopiering av sekvens:

- Kopiering av **specifik** heltalssekvens:

```
def copyIntVector(xs: Vector[Int]): Vector[Int] = for x <- xs yield x
```

- Kopiering som fungerar för alla oföränderliga heltalssekvenser:

```
def copyIntSeq(xs: Seq[Int]): Seq[Int] = for x <- xs yield x
```

```
1 scala> val xs = Vector(1,2,3)
2 xs: Vector[Int] = Vector(1, 2, 3)
3
4 scala> val ys = copyIntVector(xs)
5 ys: Vector[Int] = Vector(1, 2, 3)
6
7 scala> val zs = copyIntSeq(xs)
8 val zs: Seq[Int] = Vector(1, 2, 3)
```

### 7.1.20 Implementation med generiska funktioner

Genom att generalisera funktionshuvudena blir våra lösningar användbara för **alla** sekvenser av typen `Seq[T]`, där den obundna **typparametern** `T` vid anrop kan bindas till godtycklig typ. (Mer om typparametrar senare.)

```
def copy[T](xs: Seq[T]): Seq[T] = xs.map(x => x)

def filter[T](xs: Seq[T], p: T => Boolean): Seq[T] = xs.filter(p)

def findIndices[T](xs: Seq[T], p: T => Boolean): Seq[Int] =
  xs.indices.filter(i => p(xs(i))).toVector

def sort[T: Ordering](xs: Seq[T]): Seq[T] = xs.sorted // mer om Ordering sen

def freq[T](xs: Seq[T]): Seq[(T, Int)] =
  xs.distinct.map(_._, xs.count(_ == x))
```

Standardbibliotekets metoder försöker ordna så att det blir samma konkreta typ in som ut, men ibland väljs annan lämplig konkret samling, t.ex. kan en `Array` bli en `ArrayBuffer`.

### 7.1.21 Använda Java-samlingar i Scala med `CollectionConverters`

Med hjälp av **import** `scala.jdk.CollectionConverters.*` får man smidig **interoperabilitet** med Java och dess standardbibliotek, speciellt metoderna **asJava** och **asScala**:

```
1 scala> import scala.jdk.CollectionConverters.*
2
3 scala> Vector(1,2,3).asJava
4 res0: java.util.List[Int] = [1, 2, 3]
5
6 scala> val xs = new java.util.ArrayList[String]()
7 xs: java.util.ArrayList[String] = []
```

```
8
9 scala> xs.add("hej")
10 res1: Boolean = true
11
12 scala> xs.asScala
13 res2: scala.collection.mutable.Buffer[String] = Buffer(hej)
```

Läs mer här:

<https://docs.scala-lang.org/overviews/collections-2.13/conversions-between-java-and-scala-collections.html>

### 7.1.22 Fördjupning: Skapa generisk Array

- I JVM bytekod går det tyvärr **inte** att skapa en primitiv generisk array.
- Maskinkoden måste istället skapa en array av den mest generella referenstypen `Object` och sedan **typtesta och typkonvertera under körtid**.

Se t.ex. Java-implementationen av ArrayList:

<http://developer.classpath.org/doc/java/util/ArrayList-source.html>

- Men det *går* att skapa en generisk array i Scala (men inte i Java). Då behövs en `reflect.ClassTag` som möjliggör typinformation vid körtid för arrayer.

[illegible]

- Kompilatorn skapar då maskinkod som automatiskt gör typkonverteringarna.

### 7.1.23 Repeterade parametrar blir sekvens

Med en asterisk efter parametertypen kan antalet argument variera:

```
def sumSizes(xs: String*): Int = xs.map(_.length).sum
```

```
scala> sumSizes("Zaphod")
res0: Int = 6

scala> sumSizes("Zaphod","Beeblebrox")
res1: Int = 16

scala> sumSizes("Zaphod","Beeblebrox","Ford","Prefect")
res3: Int = 27

scala> sumSizes()
res4: Int = 0
```



Repeterade parametrar (eng. *repeated parameters*) blir en sekvens av typen Seq och som mer specifikt är en ArraySeq

---

### 7.1.24 Sekvenssamling som argument till repeterade parametrar

```
def sumSizes(xs: String*): Int = xs.map(_.size).sum  
  
val veg = Vector("gurka", "tomat")
```

Om du *redan har* en sekvenssamling så kan du applicera den på en funktion som har repeterade parametrar med hjälp av en asterisk \*

Den ska skrivas direkt **efter** den sekvenssamling, som du vill att kompilatorn ska tolka som en sekvens av argument, så här:

```
scala> sumSizes(veg*)  
res5: Int = 10
```

### 7.1.25 Enumerationer har en ordning

En uppräknings av färger i en kortlek med **enum**:

```
enum Suit:  
  case Spade, Heart, Club, Diamond
```

Användbara metoder för att hantera elementens **ordningen**:

```
scala> Suit.Spade.ordinal      // från element till heltal  
val res0: Int = 0  
  
scala> Suit.Club.ordinal  
val res1: Int = 2  
  
scala> Suit.fromOrdinal(3)    // från heltal till element  
val res2: Suit = Diamond  
  
scala> Suit.values            // alla element i ordning  
val res3: Array[Suit] = Array(Spade, Heart, Club, Diamond)  
  
scala> Suit.valueOf("Spade")  // från sträng till element  
val res4: Suit = Spade
```

### 7.1.26 Enumerationer kan ha parametrar och medlemmar

En **enum** kan ha parametrar. Använd **val** för extern synlighet:

```
enum Color(val consoleColor: String):  
  case Black extends Color(Console.BLUE) //Blå färg syns på svart bakgrund  
  case Red   extends Color(Console.RED)
```

I **enum**-kroppen kan du ha medlemmar, tex metoder:

```
enum Suit(val color: Color):
  def show(isConsoleColor: Boolean = true): String =
    if isConsoleColor then color.consoleColor + toString + Console.RESET
    else toString

  case Spade extends Suit(Color.Black)
  case Heart extends Suit(Color.Red)
  case Club extends Suit(Color.Black)
  case Diamond extends Suit(Color.Red)
```

```
scala> println(Suit.Club.show(isConsoleColor = false))
Club
```

### 7.1.27 Enum kan bli fullfjädrade case-klasser

Vill du kunna göra mönster-matching på enum-värden så behövs parametrar på alternativen för att det ska bli motsvarande case-klasser:

```
enum Veg:
  def taste: String
  case Tomato(taste: String)
  case Banana(taste: String)
```

Ovan expanderas automatiskt av kompilatorn till motsvarande detta:

```
sealed trait Veg:
  def taste: String
object Veg:
  case class Tomato(taste: String) extends Veg
  case class Banana(taste: String) extends Veg
```

### 7.1.28 Enum och mönster-matchning

Med parametrar på varje fall och en abstrakt medlem för varje attribut...

```
enum Veg:
  def taste: String
  case Tomato(taste: String)
  case Banana(taste: String)
```

...så gör den automatiska expansionen till case-klasser att detta fungerar fint:

```
scala> val v = Veg.Tomato("nice")
val v: Veg = Tomato(nice)           // notera typen : Veg

scala> v.taste // funkar eftersom Veg har en taste
val res0: String = najs
```

```
scala> val dontLikeBananas = v match:
  case Veg.Tomato(t) => t
  case Veg.Banana(_) => "always bad!"
```

Den abstrakta medlemmen **def** taste: String behövs för att attributet ska synas via referenser som är av den mindre specifika typen Veg.  
(Mer om abstrakta medlemmar i veckan om arv.)

### 7.1.29 Fördelar med enum jämfört med uppräkning med heltal

Varför inte bara så här?

```
val (Spade, Heart, Club, Diamond) = (0, 1, 2, 3)
```

Alla element har samma specifika typ enligt **enum**-deklarationen:

```
1 scala> Suit.Heart           // alla element är av typen Suit
2 val res5: Suit = Heart
```

- Detta är säkrare jämfört med att bara använda heltalsvärden: kompilatorn kan hjälpa dig att skilja på element av olika typ och ge felmeddelande om du använder fel typ oavsiktligt.
- Ej tillåtna värden kan inte representeras (jmf alla möjliga heltal, där bara några är relevanta).

Detta får du prova på veckans labb: först använda heltal sedan **enum**.

### 7.1.30 Registrering

- **Registrering** innefattar algoritmer för att kategorisera eller räkna antalet förekomster av element med vissa specifika egenskaper.
- Exempel:  
Utfallsfrekvens vid kast med en tärning 1000 gånger:

utfall		antal
1	→	178
2	→	187
3	→	167
4	→	148
5	→	155
6	→	165

### 7.1.31 Registrering av tärningskast i Array

Vi låter plats 0 representera antalet ettor, plats 1 representerar antalet tvåor etc. **Övning:** implementera ???

```
scala> def rollDice(): Int = ??? //dra slumpstal 1-6

scala> val reg = ??? //skapa heltalsarray med 6 platser
reg: Array[Int] = Array(0, 0, 0, 0, 0, 0)

scala> for k <- 1 to 1000 do
    ??? //kasta tärning, räkna ut rätt index
    ??? //registrera

scala> for i <- 1 to 6 do println(s"$i: ${reg(i - 1)}")
1: 178
2: 187
3: 167
4: 148
5: 155
6: 165
```

---

### 7.1.32 Registrering av tärningskast i Array

#### Lösning:

```
scala> def rollDice() = scala.util.Random.nextInt(6) + 1

scala> val reg = new Array[Int](6)
reg: Array[Int] = Array(0, 0, 0, 0, 0, 0)

scala> for k <- 1 to 1000 do
    val i = rollDice() - 1
    reg(i) = reg(i) + 1 // eller: reg(i) += 1

scala> for i <- 1 to 6 do println(s"$i: ${reg(i - 1)}")
1: 178
2: 187
3: 167
4: 148
5: 155
6: 165
```

---

### 7.1.33 Skapa lösningar på sekvensproblem från grunden

- Normalt använder man färdiga samlingsmetoder
- Det finns ofta en färdig metod som gör det man vill
- Annars kan man ofta göra det man vill genom att kombinera flera färdiga samlingsmetoder
- Vi ska nu i lärosyfte implementera några egna varianter av uppdatering från grunden.

För problem av typen KUTFSSR ingår det i kursen att kunna 1) lösa dessa med färdiga samlingsmetoder, och 2) implementera egna lösningar med hjälp av sekvens, alternativ, repetition, abstraktion (**SARA**).

---

### 7.1.34 Skapa ny sekvenssamling eller ändra på plats?

Två olika principer vid sekvensalgoritmkonstruktion:

- Skapa **ny sekvens** utan att förändra insekvensen
- Ändra **på plats** (eng. *in-place*) i **förändringsbar** sekvens

Välja mellan att skapa ny sekvens eller ändra på plats?

- Ofta är det **lättast att skapa ny samling** och kopiera över elementen efter eventuella förändringar medan man loopar.
- Om man har mycket stora samlingar kan man behöva ändra på plats för att spara tid/minne.

### 7.1.35 Algoritm: SEQ-COPY

**Pseudokod** för algoritmen SEQ-COPY som kopierar en sekvens, här en Array med heltal:

**Indata** : Heltalsarray *xs*  
**Utdata** : En ny heltalsarray som är en kopia av *xs*.

```

1 result ← en ny array med plats för xs.length element
2 i ← 0
3 while i < xs.length do
4   | result(i) ← xs(i)
5   | i ← i + 1
6 end
7 result
```

### 7.1.36 Implementation av SEQ-COPY med while

```

1 object seqCopy:
2
3   def arrayCopy(xs: Array[Int]): Array[Int] =
4     val result = new Array[Int](xs.length)
5     var i = 0
6     while i < xs.length do
7       result(i) = xs(i)
8       i += 1
9     result
10
11   def test: String =
12     val xs = Array(1,2,3,4,42)
13     val ys = arrayCopy(xs)
14     if xs sameElements ys then "OK!" else "ERROR!"
15
16   def main(args: Array[String]): Unit = println(test)
```

### 7.1.37 Typ-alias för att abstrahera typnamn

Med hjälp av nyckelordet **type** kan man deklarera ett **typ-alias** för att ge ett **alternativt** namn till en viss typ. Exempel:

```
1 scala> type Pt = (Int, Int)           // typalias
2 scala> type Pts = Vector[Pt]         // nästlad typalias
3
4 scala> def distToOrigo(pt: Pt): Double = math.hypot(pt._1, pt._2)
5
6 scala> val xs: Pts = Vector((1,1), (2,2), (3,4))
7 val xs: Pts = Vector((1,1), (2,2), (3,4))
8
9 scala> xs.head
10 val res0: Pt = (1,1)
11
12 scala> xs.map(distToOrigo)
13 val res1: Vector[Double] = Vector(1.4142135623730951, 2.8284271247461903, 5.0)
```

Typ-alias kan vara bra när:

- man har en lång och krånglig typ och vill använda ett kortare namn,
- man vill kunna lätt byta implementation senare (t.ex. om man vill använda en case-klass i stället för en tupel).

### 7.1.38 Exempel: SEQ-INSERT/REMOVE-COPY

Nu ska vi ”uppfinna hjulet” och som träning implementera **insättning** och **borttagning** till en **ny** sekvens utan användning av sekvenssamlingsmetoder (förutom length och apply):

```
object PointSeqUtils:
  type Pt = (Int, Int) // a type alias to make the code more concise

  def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] = ???

  def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] = ???
```

### 7.1.39 Pseudo-kod för SEQ-INSERT-COPY

```

Indata :pts: Array[Pt], pt: Pt, pos: Int
1
Utdata: En kopia av pts men där pt är infogat på plats pos
2
3
4 result ← en ny Array[Pt] med plats för pts.length + 1 element
5 for i ← 0 to pos - 1 do
6   | result(i) ← pts(i)
7 end
8 result(pos) ← pt
9 for i ← pos + 1 to xs.length do
10  | result(i) ← pts(i - 1)
11 end
12 result
13

```

**Övning:** Skriv pseudo-kod för SEQ-REMOVE-COPY

### 7.1.40 Insättning/borttagning i kopia av primitiv Array

```

1 object PointSeqUtils:
2   type Pt = (Int, Int) // a type alias to make the code more concise
3
4   def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] =
5     val result = new Array[Pt](pts.length + 1) // initialized with null
6     for i <- 0 until pos do result(i) = pts(i)
7     result(pos) = pt
8     for i <- pos + 1 to pts.length do result(i) = pts(i - 1)
9     result
10
11   def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] =
12     if pts.length > 0 then
13       val result = new Array[Pt](pts.length - 1) // initialized with null
14       for i <- 0 until pos do result(i) = pts(i)
15       for i <- pos + 1 until pts.length do result(i - 1) = pts(i)
16       result
17     else Array.empty
18
19   // ovan metoder implementerade med hjälp av den kraftfulla metoden patch:
20
21   def insertCopy(pts: Array[Pt], pos: Int, pt: Pt) = pts.patch(pos, Array(pt), 0)
22
23   def removeCopy(pts: Array[Pt], pos: Int) = pts.patch(pos, Array.empty[Pt], 1)

```

Man gör **mycket lätt fel** på gränser/specialfall: +-1, to/until, tom sekvens etc.

### 7.1.41 Exempel: PolygonWindow

- En polygon kan representeras som en punktsekvens, där varje punkt är ett heltalspar.
- PolygonWindow nedan är ett fönster som kan rita en polygon.

```

1 class PolygonWindow(width: Int, height: Int):

```

```

2  val w = new introprog.PixelWindow(width, height, title = "PolygonWindow")
3
4  def draw(pts: Seq[(Int, Int)]): Unit =
5    if pts.size > 0 then
6      for i <- 1 until pts.size do
7        w.line(pts(i - 1)._1, pts(i - 1)._2, pts(i)._1, pts(i)._2)
8      val last = pts.length - 1
9      w.line(pts(last)._1, pts(last)._2, pts(0)._1, pts(0)._2)

```

```

1  object PolygonTest:
2    val star = Array((100,180), (150,100), (180,180), (90,130), (200, 130))
3    val pw = new PolygonWindow(400,400)
4    def main(args: Array[String]): Unit = pw.draw(star.toSeq)

```

### 7.1.42 Implementera Polygon

- En polygon kan representeras som en sekvens av punkter.
- Vi vill kunna lägga till punkter, samt ta bort punkter.
- En polygon kan implementeras på många olika sätt:
  - **Förändringsbar** (eng. *mutable*)
    - \* Med punkterna i en **Array**
    - \* Med punkterna i en **ArrayBuffer**
    - \* Med punkterna i en **ListBuffer**
    - \* Med punkterna i en **Vector**
    - \* Med punkterna i en **List**
  - **Oföränderlig** (eng. *immutable*)
    - \* Som en case-klass med en oföränderlig **Vector** som returnerar nytt objekt vid uppdatering. Vi kan låta datastrukturen vara **publik** eftersom allt är oföränderligt.
    - \* Som en ”vanlig” klass med någon lämplig **privat** datastruktur där vi **inte** möjliggör förändring av efter initialisering och där vi returnerar nytt objekt vid uppdatering.

Val av implementation **beror på** sammanhang & användning!

### 7.1.43 Exempel: PolygonArray, ändring på plats

```

1  class PolygonArray(val maxSize: Int):
2    type Pt = (Int, Int)
3    private val points = new Array[Pt](maxSize) // initialized with null
4    private var n = 0
5    def size = n
6
7    def draw(w: PolygonWindow): Unit = w.draw(points.take(n).toSeq)
8
9    def append(pts: Pt*): Unit =
10      for i <- pts.indices do points(n + i) = pts(i)
11      n += pts.length
12
13    def insert(pos: Int, pt: Pt): Unit = // exercise: change pt to varargs pts
14      for i <- n until pos by -1 do points(i) = points(i - 1)

```



```

15     points(pos) = pt
16     n += 1
17
18     def remove(pos: Int): Unit = // exercise: change pos to fromPos, replaced
19         for i <- pos until n do points(i) = points(i + 1)
20         n -= 1
21
22     override def toString = points.mkString("PolygonArray(", ",", ",")")

```

- Från början är points fylld med null.
- Variabeln n håller reda på hur många som verkligen används.

#### 7.1.44 Exempel: PolygonVector, variabel referens till oföränderlig datastruktur

```

1 class PolygonVector:
2     type Pt = (Int, Int)
3     private var points = Vector.empty[Pt] // note var declaration to allow mutation
4     def size = points.size
5
6     def draw(w: PolygonWindow): Unit = w.draw(points.take(size))
7
8     def append(pts: Pt*): Unit =
9         points += pts.toVector
10
11     def insert(pos: Int, pt: Pt): Unit = // exercise: change pt to varargs pts
12         points = points.patch(pos, Vector(pt), 0)
13
14     def remove(pos: Int): Unit = // exercise: change pos to fromPos, replaced
15         points = points.patch(pos, Vector(), 1)
16
17     override def toString = points.mkString("PrimitivePolygon(", ",", ",")")

```

#### 7.1.45 Exempel: Polygon som oföränderlig case class

```

1 object Polygon:
2     type Pt = (Int, Int)
3     type Pts = Vector[Pt]
4     def apply(pts: Pt*) = new Polygon(pts.toVector)
5
6 case class Polygon(points: Polygon.Pts):
7     import Polygon.Pt
8
9     def size = points.size // for convenience but not really necessary (why?)
10
11     def append(pts: Pt*): Polygon = copy(points ++ pts.toVector)
12
13     def insert(pos: Int, pts: Pt*): Polygon = copy(points.patch(pos, pts, 0))
14
15     def remove(pos: Int, replaced: Int = 1): Polygon =
16         copy(points.patch(pos, Seq(), replaced))
17
18     override def toString = points.mkString("Polygon(", " ", " ,")")

```

### 7.1.46 Att jämföra strängar lexikografiskt

Teckenstandard **UTF-8**: Alla stora bokstäver är ”mindre” än alla små:

```
scala> Array("hej", "Hej", "gurka").sorted
res0: Array[String] = Array(Hej, gurka, hej)
```

- Antag att vi vill lösa detta problem ”från scratch”:  
**att sortera en sekvens med strängar**
- För att göra detta behöver vi lösa dessa delproblemen:
  - **att jämföra strängar**
  - **sökning i sekvenser**
  - **SWAP** (om på-plats-sortering i förändringsbar sekvens)
- Vad betyder det att två strängar är ”lika”?
- Vad betyder det att en sträng är ”mindre” än en annan?

Vi använder här strängjämförelse, sökning och sortering för att illustrera typiska **imperativa algoritmer**. **Normalt** använder man **färdiga lösningar** på dessa problem!

### 7.1.47 Jämföra strängar: likhet

Antag att vi inte kan göra `s1 == s2` utan bara kan jämföra strängar tecken för tecken, t.ex. så här: `s1(i) == s2(i)`. Antag också att vi inte har tillgång till annat än metoderna `length` och `apply` på strängar, samt **while** och variabler av grundtyp. **Lös problemet att avgöra om två strängar är lika.**

- Indata: två strängar
  - Utdata: **true** om lika annars **false**
1. Klura ut din lösningsidé
  2. Formulera algoritmen i pseudokod
  3. Implementera algoritmen i Scala:  
**def** isEqual(s1: String, s2: String): Boolean = ???

### 7.1.48 Algoritmexempel: stränglikhet, pseudokod

```
def isEqual(s1: String, s2: String): Boolean =
  if (/* lika längder */) then
    var foundDiff = false
    var i = /* första index */
    while !foundDiff && /* i inom indexgräns */ do
      if /* tecken på plats i är olika */ then foundDiff = true
      else i = /* nästa index */
    end while
    !foundDiff
  else false
end isEqual
```

Detta är en variant av s.k. **linjärsökning** där vi söker från början i en sekvens till vi hittar det vi söker efter (här söker vi efter tecken som skiljer sig åt).

Hur ser implementationen i exekverbar Scala ut?

### 7.1.49 Algoritmexempel: stränglikhet, implementation

```
def isEqual(s1: String, s2: String): Boolean =
  if s1.length == s2.length then
    var foundDiff = false
    var i = 0
    while !foundDiff && i < s1.length do
      if s1(i) != s2(i) then foundDiff = true
      else i += 1
    end while
    !foundDiff
  else false
end isEqual
```

### 7.1.50 Jämföra strängar: "mindre än"

Med  $s1 < s2$  menar vi att strängen  $s1$  ska sorteras före strängen  $s2$  enligt hur de enskilda tecknen är ordnade med uttrycket  $s1(i) < s2(i)$ .

Antag också att vi inte har tillgång till annat än metoderna `length` och `apply` på strängar, samt **while** och variabler av grundtyp, samt `math.min`

**Lös problemet att avgöra om en sträng är "mindre" än en annan.**

- Indata: två strängar,  $s1$ ,  $s2$
- Utdata: **true** om  $s1$  ska sorteras före  $s2$  annars **false**

1. Klura ut din lösningsidé
2. Formulera algoritmen i pseudokod
3. Implementera algoritmen i Scala:

```
def isLessThan(s1: String, s2: String): Boolean = ???
```

### 7.1.51 Jämföra strängar: "mindre än"

Pseudokod:

```
def isLessThan(s1: String, s2: String): Boolean =
  val minLength = /* minimum av längderna på s1 och s2 */

  def firstDiff(s1: String, s2: String): Int =
```

```

    /* index för första skillnaden (om de börjar lika: minLength) */

    val diffIndex = firstDiff(s1, s2)
    if diffIndex == minLength then /* s1 är kortare än s2 */
    else /* tecknet s1(diffIndex) är mindre än tecknet s2(diffIndex) */

```

### 7.1.52 Jämföra strängar: "mindre än"

```

def isLessThan(s1: String, s2: String): Boolean =
    val minLength = math.min(s1.length, s2.length)

    def firstDiff(s1: String, s2: String): Int =
        var foundDiff = false
        var i = 0
        while !foundDiff && i < minLength do
            if (s1(i) != s2(i)) foundDiff = true
            else i += 1
        end while
        i
    end firstDiff

    val diffIndex = firstDiff(s1, s2)
    if diffIndex == minLength then s1.length < s2.length
    else s1(diffIndex) < s2(diffIndex)
end isLessThan

```

### 7.1.53 Sökning

- **Sökning** återkommer i många skepnader:  
i en datastruktur, vilken det än må vara, vill man ofta kunna **hitta ett element med en viss egenskap**.  
Några färdiga linjärsökningar i Scalas standardbibliotek:

```

1 scala> Vector("gurka","tomat","broccoli").indexOf("tomat")
2 res0: Int = 1
3
4 scala> Vector("gurka","tomat","broccoli").indexWhere(_.contains("o"))
5 res1: Int = 1
6
7 scala> Vector("gurka","tomat","broccoli").find(_.contains("o"))
8 res2: Option[String] = Some(tomat)

```

- Sökning efter ett visst index i en sekvens:
  - Indata: en sekvens och ett **sökkriterium**
  - Utdata: index för första eftersökta element, annars -1

- Två typiska varianter av sökning i en sekvens:
  - Linjärsökning: börja från början och sök tills ett eftersökt element är funnet
  - Binärsökning: antag sorterad sekvensen; börja i mitten, välj rätt halva ...

### 7.1.54 Linjärsökning: hitta index för elementet $x$

Implementera `indexOf`:

```
def indexOf(xs: Vector[Int], x: Int): Int = ???
```

Utdata: index  $i$  där `xs(i) == x`

Om värde saknas. returnera -1

```
def indexOf(xs: Vector[Int], x: Int): Int =
  var i = 0
  var found = false
  while !found && i < xs.length do
    if (xs(i) == x) found = true
    else i += 1
  if (found) i else -1
```

(Är du nyfiken på binärsökning, se kapitel 12: Valfri fördjupning.)

### 7.1.55 Sortering

**Problem:** Vi har en osorterad sekvens med heltal. Vi vill ordna denna osorterade sekvens i en sorterad sekvens från minst till störst.

En *generalisering* av problemet:

Vi har många element av godtycklig typ och en **ordningsrelation** som säger vad vi menar med att ett element är *mindre än* eller *större än* eller *lika med* ett annat element.

Vi vill lösa problemet att ordna elementen i sekvens så att för varje element på plats  $i$  så är efterföljande element på plats  $i + 1$  större eller lika med elementet på plats  $i$ .

- Insättningssortering **lösningssidé:** Ta ett element i taget från den osorterade listan och **sätt in** det på **rätt plats** i den sorterade listan och upprepa till det inte finns fler osorterade element.

### 7.1.56 Det finns många olika sorteringsalgoritmer

- Visualisering av 15 olika sorteringsalgoritmer på 6 min:  
<https://www.youtube.com/watch?v=kPRA0W1kECg>

- Olika sorteringsalgoritmer har olika tids- & minneskomplexitet: i bästa fall, i värsta fall, i medeltal, för nästan sorterad, etc.  
[https://en.wikipedia.org/wiki/Sorting\\_algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm)
- Olika sorteringsalgoritmer lämpar sig olika väl för parallellisering på många kärnor.

### 7.1.57 Bogo sort

```
def bogoSort(xs: Vector[Int]) =
  var result = xs
  while result != result.sorted do
    result = scala.util.Random.shuffle(result)
  result
```

När blir denna färdig?

Antal jämförelser i medeltal vid  $n$  element:  $n \cdot n!$

<https://en.wikipedia.org/wiki/Bogosort>

### 7.1.58 Sortera till ny vektor med insättningssortering: pseudo-kod

Det är nog lättare att förstå **insertion sort** om man sorterar till en ny vektor. Vi ska sedan se hur man sorterar ”på plats” (eng. *in place*) i en array.

**Indata:** en osorterad vektor med heltal

**Utdata:** en ny, sorterad vektor med heltal

```
def insertionSort(xs: Vector[Int]): Vector[Int] =
  val sorted = /* tom ArrayBuffer */
  for /* alla element i xs */ do
    /* linjärsök rätt position i sorted */
    /* sätt in element på rätt plats i sorted */
  end for
  sorted.toVector
```

### 7.1.59 Sortera till ny vektor med insättningssortering: implementation

```
def insertionSort(xs: Vector[Int]): Vector[Int] =
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  for elem <- xs do
    // linjärsök rätt position i sorted:
    var pos = 0
```

```

    while pos < sorted.length && sorted(pos) < elem do
      pos += 1
    end while
    // sätt in element på rätt plats i sorted:
    sorted.insert(pos, elem)
  end for
  sorted.toVector
end insertionSort

```

### 7.1.60 Sorter till ny samling med godtyckligt ordningspredikat

```

def sortWith(xs: Vector[Int])(lt: (Int, Int) => Boolean ): Vector[Int] =
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  for elem <- xs do // insertion sort using lt as "less than"
    var pos = 0
    while pos < sorted.length && lt(sorted(pos), elem) do
      pos += 1
    end while
    sorted.insert(pos, elem)
  end for
  sorted.toVector
end sortWith

```

```

1 scala> val xs = Vector(1,2,1,2,12,42,1)
2
3 scala> sortWith(xs)(_ < _)
4 val res0: Vector[Int] = Vector(1, 1, 1, 2, 2, 12, 42)
5
6 scala> sortWith(xs)(_ > _)
7 val res1: Vector[Int] = Vector(42, 12, 2, 2, 1, 1, 1)

```

### 7.1.61 Insättningssortering på plats – pseudo-kod

**Indata:** en array med heltal

**Utdata:** samma array, men nu sorterad

```

def insertionSortInPlace(xs: Array[Int]): Unit =
  for i <- 1 until xs.length do //från ANDRA till sista
    var j = i
    while j > 0 && xs(j - 1) > xs(j) do
      /* byt plats på xs(j) och xs(j - 1) */
      j -= 1; // stega bakåt
    end while
  end for

```

Se animering här: [Insättningssortering på wikipedia](#)

Gå igenom alla specialfall och kolla så att detta fungerar!

### 7.1.62 Insättningssortering på plats – implementation

```
def insertionSortInPlaceSwap(xs: Array[Int]): Unit =  
  def swap(i: Int, j: Int): Unit =  
    val temp = xs(i)  
    xs(i) = xs(j)  
    xs(j) = temp  
  end swap  
  
  for i <- 1 until xs.length do //från ANDRA till sista  
    var j = i  
    while j > 0 && xs(j - 1) > xs(j) do  
      swap(j, j - 1)  
      j -= 1; // stega bakåt  
    end while  
  end for  
end insertionSortInPlaceSwap
```

---



## 7.2 Övning sequences

### Mål

- ☐ Kunna läsa och skriva pseudokod för sekvensalgoritmer och implementera sekvensalgoritmer enligt pseudokod.
- ☐ Kunna implementera sekvensalgoritmer, både genom kopiering till ny sekvens och genom förändring på plats i befintlig sekvens.
- ☐ Kunna använda inbyggda metoder för uppdatering av, linjärsökning i, och sortering av sekvenssamlingar.
- ☐ Kunna beskriva skillnaden i användningen av föränderliga och oföränderliga sekvenser, speciellt vid uppdatering.
- ☐ Förstå hur sorteringsordningen är definierad för strängar.
- ☐ Kunna sortera sekvenssamlingar innehållande objekt av grundtyper med hjälp av inbyggda och egendefinierade sorteringsordningar med metoderna `sorted`, `sortBy` och `sortWith`.
- ☐ Kunna implementera linjärsökning enligt olika sökkriterier.
- ☐ Kunna beskriva egenskaperna hos sekvenssamlingarna `Vector`, `List`, `Array`, `ArrayBuffer` och `ListBuffer`.
- ☐ Förstå bieffekter av uppdatering av delade referenser till föränderliga element.
- ☐ Kunna använda funktioner med repeterade parametrar.
- ☐ Känna till hur man implementerar funktioner med repeterade parametrar.
- ☐ Kunna implementera heltalsregistrering i en heltalsarray.

### Förberedelser

- ☐ Studera begreppen i kapitel 7

### 7.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** *Para ihop begrepp med beskrivning.*

Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

element	1	A	definierar hur element av en viss typ ska ordnas
samling	2	B	datastruktur med element av samma typ
samlingsbibliotek	3	C	algoritm som ordnar element i en viss ordning
sekvens(samling)	4	D	algoritm som letar upp element enligt sökkriterium
sekvensalgoritm	5	E	hur exekveringstiden växer med problemstorleken
ordning	6	F	sökalgoritm som letar i sekvens tills element hittas
sortering	7	G	objekt i en datastruktur
sökning	8	H	algoritm som räknar element med vissa egenskaper
linjärsökning	9	I	lösning på problem som drar nytta av sekvenssamling
registrering	10	J	många färdiga samlingar med olika egenskaper
tidskomplexitet	11	K	hur minnesåtgången växer med problemstorleken
minneskomplexitet	12	L	noll el. flera element av samma typ i viss ordning

**Uppgift 2.** *Olika sekvenssamlingar.* Koppla varje sekvenssamling med den (förenklade) beskrivning som passar bäst:

Vector	1	A	förändringsbar, snabb indexering, kan ändra storlek
List	2	B	oföränderlig, ger snabbt godtyckligt ändrad samling
Array	3	C	oföränderlig, ger snabbt ny samling ändrad i början
ArrayBuffer	4	D	primitiv, förändringsbar, snabb indexering, fix storlek
ListBuffer	5	E	förändringsbar, snabb att ändra i början

**Uppgift 3.** *Använda sekvenssamlingar.* Antag att nedan variabler finns synliga i aktuell namnrymd:

```
val xs: Vector[Int] = Vector(1, 2, 3)
val x: Int = 0
```

a) Koppla varje uttryck till vänster med motsvarande resultat till höger. Om du är osäker på resultatet, läs i snabbreferensen och testa i REPL.

*Tips: "colon on the collection side".*

<code>x += xs</code>	1	A	<b>true</b>
<code>xs += x</code>	2	B	<code>Vector(2, 2, 3)</code>
<code>xs :+ x</code>	3	C	1
<code>xs ++ xs</code>	4	D	error: value tail is not a member of Int
<code>xs.indices</code>	5	E	(0 until 3)
<code>xs apply 0</code>	6	F	<code>Vector(1, 2, 3)</code>
<code>xs(3)</code>	7	G	<code>Vector(0, 1, 2, 3)</code>
<code>xs.length</code>	8	H	<b>false</b>
<code>xs.take(4)</code>	9	I	<code>java.lang.IndexOutOfBoundsException</code>
<code>xs.drop(2)</code>	10	J	<code>Vector(1, 2, 3, 0)</code>
<code>xs.updated(0, 2)</code>	11	K	<code>Vector(3)</code>
<code>xs.tail.head</code>	12	L	error: value += is not a member of Int
<code>xs.head.tail</code>	13	M	<code>Vector(1, 2, 3, 1, 2, 3)</code>
<code>xs.isEmpty</code>	14	N	2
<code>xs.nonEmpty</code>	15	O	3

b) Vid tre tillfällen blir det fel. Varför? Är det kompilleringsfel eller exekveringsfel?

*Tips inför fortsättningen:* Scalas standardbibliotek har många användbara samlingar med enhetlig metoduppsättning. Om du lär dig de viktigaste samlingsmetoderna får du en kraftfull verktygslåda. Läs mer här:

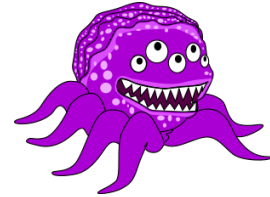
- snabbreferensen (enda tentahjälpmedel):

<http://cs.lth.se/pgk/quickref>

- översikt (av Prof. Martin Odersky, uppfinnare av Scala, m.fl.):  
<http://docs.scala-lang.org/overviews/collections/introduction.html>
- api-dokumentation:  
<https://www.scala-lang.org/api/current/scala/collection/>

**Uppgift 4. Kopiering av sekvenser.** Klassen `Mutant` nedan kan användas för att skapa förändringsbara instanser med heltal.<sup>1</sup>

```
class Mutant(var int: Int = 0)
```



**Figur 7.1:** En instans av klassen `Mutant` där `int` kanske är 5.

Kör nedan i REPL efter studier av detta: <https://youtu.be/dpd0UEe9mm4>

```
1 scala> val fem = new Mutant(5)
2 scala> val xs = Vector(fem, fem, fem)
3 scala> val ys = xs.toArray // kopierar referenserna till ny Array
4 scala> val zs = xs.map(x => new Mutant(x.int)) // djupkopierar till ny Vector
5 scala> xs(0).int = (new Mutant).int
```

a) Fyll i tabellen nedan genom att till höger skriva värdet av varje uttryck till vänster. Förklara vad som händer. *Tips:* Metoden `eq` jämför alltid referenser (ej innehåll).

<code>xs(0)</code>	
<code>ys(0).int</code>	
<code>zs(0).int</code>	
<code>xs(0) eq ys(0)</code>	
<code>xs(0) eq zs(0)</code>	
<code>(ys.toBuffer :+ new Mutant).apply(0).int</code>	

b) Implementera med hjälp av en **while**-sats funktionen `deepCopy` nedan som gör *djup* kopiering, d.v.s skapar en ny array med nya, innehållskopierade mutanter.

```
def deepCopy(xs: Array[Mutant]): Array[Mutant] = ???
```

Använd denna algoritm:

<sup>1</sup>Om den inbyggda grundtypen `Int`, i likhet med `Mutant`, knasigt nog kunnat användas för att skapa förändringsbara instanser hade heltalsmatematiken i Scala omvandlats till ett skrämmande kaos.

**Indata :** En mutantarray *xs*

**Utdata:** En djup kopia av *xs*

```

1 result ← en ny mutantarray med plats för lika många element som i xs
2 i ← 0
3 while i mindre än antalet element do
4   skapa en kopia av elementet xs(i) och lägg kopian i result på platsen i
5   öka i med 1
6 end
7 result

```

c) Testa att din funktion och kolla så att inga läskiga muteringar genom delade referenser går att göra, så som med *xs* och *ys* i första deluppgiften.

d) Är det vanligt att man, för säkerhets skull, gör djupkopiering av alla element i oföränderliga samlingar som enbart innehåller oföränderliga element?

*Tips inför fortsättningen:* Ofta kan du lösa grundläggande delproblem med inbyggda samlingsmetoder ur standardbiblioteket. Till exempel kan ju kopieringen i `deepCopy` i föregående uppgift enkelt göras med hjälp av samlingsmetoden `map`.

Men det är mycket bra för din förståelse om du kan implementera grundläggande sekvensalgoritmer själv även om det normalt är bättre att använda färdiga, vältestade metoder. I kommande uppgifter ska du därför göra egna implementationer av några sekvensalgoritmer som redan finns i standardbiblioteket.

**Uppgift 5.** *Uppdatering av sekvenser.* Deklarera dessa variabler i REPL:

```

val xs = (1 to 4).toVector
val buf = xs.toBuffer

```

a) Uttrycken till vänster evalueras uppifrån och ned. Para ihop med rätt resultat.

{ buf(0) = -1; buf(0) }	1	A	error: value update is not a member
{ xs(0) = -1; xs(0) }	2	B	Vector(5, 2, 3, 4)
buf.update(1, 5)	3	C	ArrayBuffer(-1, 5, 3, 4, 5)
xs.updated(0, 5)	4	D	-1
{ buf += 5; buf }	5	E	Vector(1, -1, 5)
{ xs += 5; xs }	6	F	(): Unit
xs.patch(1, Vector(-1, 5), 3)	7	G	error: value += is not a member
xs	8	H	Vector(1, 2, 3, 4)

*Tips:* Läs om metoderna i snabbreferensen och undersök i REPL. Exempel:

```

1 scala> Vector(1,2,3,4).patch(from = 1, other = Vector(0,0), replaced = 3)
2 val res0: Vector[Int] = Vector(1, 0, 0)

```

b) Implementera funktionen `insert` nedan med hjälp av sekvenssamlingsmetoden `patch`.  
*Tips:* Ge argumentet 0 till parametern `replaced`.

```
/** Skapar kopia av xs men med elem insatt på plats pos. */
def insert(xs: Array[Int], elem: Int, pos: Int): Array[Int] = ???
```

- c) Skriv pseduokod för en algoritm som implementerar insert med hjälp av **while**.
- d) Implementera insert enligt din pseudokod. Testa i REPL och se vad som händer om pos är negativ? Vad händer om pos är precis ett steg bortom sista platsen i xs? Vad händer om pos är flera steg bortom sista platsen?

*Tips inför fortsättningen:* Det är inte lätt att få rätt på alla specialfall även i små algoritmer så som insert ovan. Det är därför viktigt att noga tänka igenom sin sekvensalgoritm med avseende på olika specialfall. Använd denna checklista:

1. Vad händer om sekvensen är tom?
2. Fungerar det för exakt ett element?
3. Kan index bli negativt?
4. Kan index bli mer än längden minus ett?
5. Kan det bli en oändlig loop, t.ex. p.g.a. saknad loopvariabeluppräknings?

Ibland vill man att vettiga undantag ska kastas vid ogiltig indata eller andra feltillstånd och då är require eller assert bra att använda. I andra fall vill man att resultatet t.ex. ska bli en tom sekvenssamling om indata är ogiltigt. Sådana beteenden behöver dokumenteras så att andra som använder dina algoritmer (eller du själv efter att du glömt hur det var) förstår vad som händer i olika fall.

**Uppgift 6.** *Jämföra strängar i Scala.* I Scala kan strängar jämföras med operatorerna ==, !=, <, <=, >, >=, där likhet/olikhet avgörs av om alla tecken i strängen är lika eller inte, medan större/mindre avgörs av sorteringsordningen i enlighet med varje teckens Unicode-värde.<sup>2</sup>

- a) Vad ger följande jämförelser för värde?

```
1 scala> 'a' < 'b'
2 scala> "aaa" < "aaaa"
3 scala> "aaa" < "bbb"
4 scala> "AAA" < "aaa"
5 scala> "ÄÄÄ" < "ÖÖÖ"
6 scala> "ÅÅÅ" < "ÄÄÄ"
```

Tyvärr så följer ordningen av ÄÄÖ inte svenska regler, men det ignorerar vi i fortsättningen för enkelhets skull; om du är intresserad av hur man kan fixa detta, gör uppgift 20.



- b) Vilken av strängarna s1 och s2 kommer först (d.v.s. är "mindre") om s1 utgör början av s2 och s2 innehåller fler tecken än s1?

**Uppgift 7.** *Linjärsökning enligt olika sökkriterier.* Linjärsökning innebär att man letar tills man hittar det man söker efter i en sekvens. Detta delproblem återkommer ofta! Vanligen börjar linjärsökning från början och håller på tills man hittar något element som uppfyller

<sup>2</sup>Överkurs: Alla tecken i en java.lang.String representeras enligt UTF-16-standard (https://en.wikipedia.org/wiki/UTF-16), vilket innebär att varje Unicode-kodpunkt (eng. *code point*) lagras som antingen ett eller två 16-bitars heltal. Strängjämförelse i Scala och Java jämför egentligen inte varje tecken, utan varje 16-bitars heltal. Denna skillnad har ingen betydelse när en sträng bara innehåller tecken som tar upp ett 16-bitars heltal var, och praktiskt nog är nästan alla tecken som används vardagligen av den typen. De flesta tecken som kräver två 16-bitars heltal är sällsynta kinesiska tecken, sällsynta symboler, tecken från utdöda språk och emoji. Vi kommer att bortse från sådana tecken i den här kursen.

kriteriet. Beroende på vad som finns i sekvensen och hur kriteriet ser ut kan det hända att man måste gå igenom alla element utan att hitta det som söks.

a) Linjärsökning med inbyggda sekvenssamlingsmetoder.

```
val xs = ((1 to 5).reverse ++ (0 to 5)).toVector
```

Deklarera ovan variabel i REPL och para ihop uttrycken nedan med rätt värden. Förklara vad som händer.

<code>xs.indexOf(0)</code>	1	A	<code>Vector(1, 1)</code>
<code>xs.indexOf(6)</code>	2	B	<code>-1</code>
<code>xs.indexWhere(_ &lt; 2)</code>	3	C	<code>true</code>
<code>xs.indexWhere(_ != 5)</code>	4	D	<code>Some(1)</code>
<code>xs.find(_ == 1)</code>	5	E	<code>Vector(1, 0, 1)</code>
<code>xs.find(_ == 6)</code>	6	F	<code>5</code>
<code>xs.contains(0)</code>	7	G	<code>Vector(4, 6)</code>
<code>xs.filter(_ == 1)</code>	8	H	<code>4</code>
<code>xs.filterNot(_ &gt; 1)</code>	9	I	<code>1</code>
<code>xs.zipWithIndex.filter(_._1 == 1).map(_._2)</code>	10	J	<code>None</code>

b) Implementera linjärsökning i strängvektor med strängpredikat.

```
/** Returns first index where p is true. Returns -1 if not found. */
def indexOf(xs: Vector[String], p: String => Boolean): Int = ???
```

Ett strängpredikat `p: String => Boolean` är en funktion som tar en sträng som indata och ger ett booleskt värde som resultat. Implementera `indexOf` med hjälp av en **while**-sats. Du kan t.ex. använda en lokal boolesk variabel `found` för att hålla reda på om du har hittat det som eftersöks enligt predikatet.

När element som uppfyller predikatet saknas måste man bestämma vad som ska hända. Kravet på din implementation i detta fall ges av dokumentationskommentaren ovan.

Din funktion ska fungera enligt nedan:

```
1 scala> val xs = Vector("hej", "på", "dej")
2 val xs: Vector[String] = Vector(hej, på, dej)
3
4 scala> indexOf(xs, _.contains('p'))
5 val res0: Int = 1
6
7 scala> indexOf(xs, _.contains('q'))
8 val res1: Int = -1
9
10 scala> indexOf(Vector(), _.contains('q'))
11 val res2: Int = -1
12
13 scala> indexOf(Vector("q"), _.length == 1)
14 val res3: Int = 0
```

**Uppgift 8. Labbförberedelse: Implementera heltalsregistrering i Array.** Registrering innebär att man räknar antalet förekomster av olika värden. Varje gång ett nytt värde förekommer

behöver vi räkna upp en frekvensräknare. Det behövs en räknare för varje värde som ska registreras. Vi ska fortsätta räkna ända tills alla värden är registrerade.

På veckans laboration ska du registrera förekomsten av olika kortkombinationer i kortspelet poker. I denna övning ska du som träning inför laborationen lösa ett liknande registreringsproblem: frekvensanalys av många tärningskast. Vid tärningsregistrering behövs sex olika räknare. Man kan med fördel då använda en sekvenssamling med plats för sex heltal. Man kan t.ex. låta plats 0 hålla reda på antalet ettor, plats 1 hålla reda på antalet tvåor, etc.

a) Implementera nedan algoritm enligt pseudokoden:

```
def registreraTärningskast(xs: Seq[Int]): Vector[Int] =
  val result = ??? /* Array med 6 nollor */
  xs.foreach{ x =>
    require(x >= 1 && x <= 6, "tärningskast ska vara mellan 1 & 6")
    ??? /* räkna förekomsten av x */
  }
  result.toVector
```

b) Använd funktionen kasta nedan när du testar din registreringsalgoritm med en sekvenssamling innehållande minst 1000 tärningskast.

```
def kasta(n: Int) = Vector.fill(n)(util.Random.nextInt(6) + 1)
```

**Uppgift 9.** *Inbyggda metoder för sortering.* Det finns fler olika sätt att ordna sekvenser efter olika kriterier. För grundtyperna Int, Double, String, etc., finns inbyggda ordningar som gör att sekvenssamlingsmetoden sorted fungerar utan vidare argument (om du är nöjd med den inbyggda ordningsdefinitionen). Det finns också metoderna sortBy och sortWith om du vill ordna en sekvens med element av någon grundtyp efter egna ordningsdefinitioner eller om du har egna klasser i din sekvens.

```
val xs = Vector(1, 2, 1, 3, -1)
val ys = Vector("abra", "ka", "dabra").map(_.reverse)
val zs = Vector('a', 'A', 'b', 'c').sorted

case class Person(förnamn: String, efternamn: String)

val ps = Vector(Person("Kim", "Ung"), Person("kamrat", "Clementin"))
```

Deklarera ovan i REPL och para ihop uttryck nedan med rätt resultat.

*Tips:* Stora bokstäver sorteras före små bokstäver i den inbyggda ordningen för grundtyperna String och Char. Dessutom har svenska tecken knasig ordning.<sup>3</sup>

Läs om sorteringsmetoderna i snabbreferensen och prova i REPL.

<sup>3</sup>Ordningen kommer ursprungligen från föråldrade teckenkodningsstandarder: <https://sv.wikipedia.org/wiki/ASCII>



'a' < 'A'	1	A	"ka"
"ÄÖö" < "ÄÖö"	2	B	1
xs.sorted.head	3	C	-1
xs.sorted.reverse.head	4	D	error: ...
ys.sorted.head	5	E	<b>false</b>
zs.indexOf('a')	6	F	0
ps.sorted.head.förnamn.take(2)	7	G	3
ps.sortBy(_.förnamn).apply(1).förnamn.take(2)	8	H	<b>true</b>
xs.sortWith((x1,x2) => x1 > x2).indexOf(3)	9	I	"ak"

Vi ska senare i kursen implementera egna sorteringsalgoritmer som träning, men i normala fall använder man inbyggda sorteringar som är effektiva och vältestade. Dock är det inte ovanligt att man vill definiera egna ordningar för egna klasser, vilket vi ska undersöka senare i kursen.

**Uppgift 10.** *Inbyggd metod för blandning.* På veckans laboration ska du implementera en egen blandningsalgoritm och använda den för att blanda en kortlek. Det finns redan en inbyggd metod `shuffle` i singelobjektet `Random` i paketet `scala.util`.

a) Sök upp dokumentationen för `Random.shuffle` och studera funktionshuvudet. Det står en hel del invecklade saker om `CanBuildFrom` etc. Detta smarta krångel, som vi inte går närmare in på i denna kurs, är till för att metoden ska kunna returnera lämplig typ av samling. När du ser ett sådant funktionshuvud kan du anta att metoden fungerar fint med flera olika typer av lämpliga samlingar i Scalas standardbibliotek.

Klicka på `shuffle`-dokumentationen så att du ser hela texten. Vad säger dokumentationen om resultatet? Är det blandning på plats eller blandning till ny samling?

b) Prova upprepade blandningar av olika typer av sekvenser med olika typer av element i REPL.

**Uppgift 11.** *Repeterade parametrar.* Det går att deklarerera en funktion som tar en argumentsekvens av godtycklig längd, ä.k. *varargs*. Syntaxen består av en asterisk `*` efter typen. Funktion sägs då ha repeterade parametrar (eng. *repeated parameters*). I funktionskroppen får man tillgång till argumenten i en sekvenssamling. Argumenten anges godtyckligt många med komma emellan. Exempel:

```
/** Ger en vektor med stränglängder för godtyckligt antal strängar. */
def stringSizes(xs: String*): Vector[Int] = xs.map(_.size).toVector
```

a) Deklarera och använd `stringSizes` i REPL. Vad händer om du anropar `stringSizes` med en tom argumentlista?

b) Det händer ibland att man redan har en sekvenssamling, t.ex. `xs`, och vill skicka med varje element som argument till en *varargs*-funktion. Syntaxen för detta är `xs: _*` vilket gör att kompilatorn omvandlar sekvenssamlingen till en argumentsekvens av rätt typ.

Prova denna syntax genom att ge en `xs` av typen `Vector[String]` som argument till `stringSizes`. Fungerar det även om `xs` är en sekvens av längden 0?



## 7.2.2 Extrauppgifter; träna mer

**Uppgift 12.** *Registrering av booleska värden. Singla slant.*

a) Implementera en funktion som registrerar många slantsinglingar enligt nedan funktionshuvud. Indata är en sekvens av booleska värden där krona kodas som **true** och klave kodas som **false**. För registreringen ska du använda en lokal `Array[Int]`. I resultatet ska antalet utfall av krona ligga på första platsen i 2-tupeln och på andra platsen ska antalet utfall av klave ligga.

```
def registerCoinFlips(xs: Seq[Boolean]): (Int, Int) = ???
```

b) Skapa en funktion `flips(n)` som ger en boolesk `Vector` med  $n$  stycken slantsinglingar och använd den när du testar din slantsinglingsregistreringsalgoritm.

**Uppgift 13.** *Kopiering och tillägg på slutet.* Skapa funktionen `copyAppend` som implementerar nedan algoritm, efter att du rättat de **två buggarna** nedan:

**Indata:** Heltalsarray  $xs$  och heltalet  $x$   
**Utdata:** En ny heltalsarray som är en kopia av  $xs$  men med  $x$  tillagt på slutet som extra element.

```

1  $ys \leftarrow$  en ny array med plats för ett element mer än i  $xs$ 
2  $i \leftarrow 0$ 
3 while  $i \leq xs.length$  do
4    $ys(i) \leftarrow xs(i)$ 
5 end
6 lägg  $x$  på sista platsen i  $ys$ 
7  $ys$ 
```

Granska din kod enligt checklisten i tidigare tipsruta. Testa din funktion för de olika fallen: tom sekvens, sekvens med exakt ett element, sekvens med många element.

**Uppgift 14.** *Kopiera och reversera sekvens.* Implementera `seqReverseCopy` enligt:

**Indata:** Heltalsarray  $xs$   
**Utdata:** En ny heltalsarray med elementen i  $xs$  i omvänd ordning.

```

1  $n \leftarrow$  antalet element i  $xs$ 
2  $ys \leftarrow$  en ny heltalsarray med plats för  $n$  element
3  $i \leftarrow 0$ 
4 while  $i < n$  do
5    $ys(n - i - 1) \leftarrow xs(i)$ 
6    $i \leftarrow i + 1$ 
7 end
8  $ys$ 
```

a) Använd en **while**-sats på samma sätt som i algoritmen. Prova din implementation i REPL och kolla så att den fungerar i olika fall.

b) Gör en ny implementation som i stället använder en **for**-sats som börjar bakifrån. Kör din implementation i REPL och kolla så att den fungerar i olika fall.

**Uppgift 15.** *Kopiera alla utom ett.* Implementera kopiering av en array *utom* ett element på en viss angiven plats. Skriv först pseudokod innan du implementerar:

```
def removeCopy(xs: Array[Int], pos: Int): Array[Int]
```

**Uppgift 16.** *Borttagning på plats i array.* Ibland vill man ta bort ett element på en viss position i en array utan att kopiera alla element till en ny samling. Ett sätt att göra detta är att flytta alla efterföljande element ett steg mot lägre index och fylla ut sista positionen med ett utfyllnadsvärde, t.ex. 0. Skriv först pseudokod för en sådan algoritm. Implementera sedan algoritmen i en funktion med denna signatur:

```
def removeAndPad(xs: Array[Int], pos: Int, pad: Int = 0): Unit
```

**Uppgift 17.** *Kopiering och insättning.*

a) Implementera en funktion med detta huvud enligt efterföljande algoritm:

```
def insertCopy(xs: Array[Int], x: Int, pos: Int): Array[Int]
```

**Indata :** En sekvens  $xs$  av typen `Array[Int]` och heltalen  $x$  och  $pos$   
**Utdata:** En ny sekvens av typen `Array[Int]` som är en kopia av  $xs$  men där  $x$  är infogat på plats  $pos$

```

1  $n \leftarrow$  antalet element  $xs$ 
2  $ys \leftarrow$  en ny Array[Int] med plats för  $n + 1$  element
3 for  $i \leftarrow 0$  to  $pos - 1$  do
4    $ys(i) \leftarrow xs(i)$ 
5 end
6  $ys(pos) \leftarrow x$ 
7 for  $i \leftarrow pos$  to  $n - 1$  do
8    $ys(i + 1) \leftarrow xs(i)$ 
9 end
10  $ys$ 
```

- b) Vad måste  $pos$  vara för att det ska fungera med en tom array som argument?
- c) Vad händer om din funktion anropas med ett negativt argument för  $pos$ ?
- d) Vad händer om din funktion anropas med  $pos$  lika med  $xs.size$ ?
- e) Vad händer om din funktion anropas med  $pos$  större än  $xs.size$ ?

**Uppgift 18.** *Insättning på plats i array.* Ett sätt att implementera insättning i en array, utan att kopiera alla element till en ny array med en plats extra, är att alla elementen efter  $pos$  flyttas fram ett steg till högre index, så att plats bereds för det nya elementet. Med denna lösning får det sista elementet "försvinna" genom brutal överskrivning eftersom arrayer inte kan ändra storlek.

Skriv först en sådan algoritm i pseudokod och implementera den sedan i en procedur med detta huvud:

```
def insertDropLast(xs: Array[Int], x: Int, pos: Int): Unit
```

**Uppgift 19.** *Fler inbyggda metoder för linjärsökning.*

- a) Läs i snabbreferensen om metoderna `lastIndexOf`, `indexOfSlice`, `segmentLength` och `maxBy` och beskriv vad var och en kan användas till.

- b) Testa metoderna i REPL.

### 7.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 20.** *Fixa svensk sorteringsordning av ÄÅÖ.* Svenska bokstäver kommer i, för svenskar, konstig ordning om man inte vidtar speciella åtgärder. Med hjälp av klassen `java.text.Collator` kan man få en `Comparator` för strängar som följer lokala regler för en massa språk på planeten jorden.

a) Verifiera att sorteringsordningen blir rätt i REPL enligt nedan.

```
1 scala> val fel = Vector("ö", "å", "ä", "z").sorted
2 scala> val svColl = java.text.Collator.getInstance(new java.util.Locale("sv"))
3 scala> val svOrd = Ordering.comparatorToOrdering(svColl)
4 scala> val rätt = Vector("ö", "å", "ä", "z").sorted(svOrd)
```

b) Använd metoden ovan för att skriva ett program som skriver ut raderna i en textfil i korrekt svensk sorteringsordning. Programmet ska kunna köras med kommandot:

```
scala sorted -sv textfil.txt
```

c) Läs mer här:

[stackoverflow.com/questions/24860138/sort-list-of-string-with-localization-in-scala](https://stackoverflow.com/questions/24860138/sort-list-of-string-with-localization-in-scala)

**Uppgift 21.** *Fibonacci-sekvens med ListBuffer.* Samlingen `ListBuffer` är en förändringsbar sekvens som är snabb och minnessnål vid tillägg i början (eng. *prepend*). Undersök vad som händer här:

```
1 scala> val xs = scala.collection.mutable.ListBuffer.empty[Int]
2 scala> xs.prependAll(Vector(1, 1))
3 scala> while xs.head < 100 do {xs.prepend(xs.take(2).sum); println(xs)}
4 scala> xs.reverse.toList
```

Talen i sekvensen som produceras på rad 4 ovan kallas Fibonacci-tal<sup>4</sup> och blir snabbt mycket stora.

a) Definera och testa följande funktion. Den ska internt använda förändringsbara `ListBuffer` men returnera en sekvens av oföränderliga `List`.

```
/** Ger en lista med tal ur Fibonacci-sekvensen 1, 1, 2, 3, 5, 8 ...
 * där det största talet är mindre än max. */
def fib(max: Long): List[Long] = ???
```

b) Hur lång ska en Fibonacci-sekvens vara för att det sista elementet ska vara så nära `Int.MaxValue` som möjligt?

c) Implementera `fibBig` som använder `BigInt` i stället för `Long` och låt din dator få använda sitt stora minne medan planeten värms upp en aning.

**Uppgift 22.** *Omvända sekvens på plats.* Implementera nedan algoritm i funktionen `reverseChars` och testa så att den fungerar för olika fall i REPL.

<sup>4</sup>[sv.wikipedia.org/wiki/Fibonaccital](https://sv.wikipedia.org/wiki/Fibonaccital)

**Indata :** En array *xs* med tecken

**Utdata:** *xs* uppdaterat på plats, med tecknen i omvänd ordning

```

1 n ← antalet element i xs
2 for i ← 0 to  $\frac{n}{2} - 1$  do
3   | temp ← xs(i)
4   | xs(i) ← xs(n - i - 1)
5   | xs(n - i - 1) ← temp
6 end
```

**Uppgift 23.** *Palindrompredikat.* En palindrom<sup>5</sup> är ett ord som förblir oförändrat om man läser det baklänges. Exempel på palindromer: kajak, dallassallad.

Ett sätt att implementera ett palindrompredikat visas nedan:

```
def isPalindrome(s: String): Boolean = s == s.reverse
```

- Implementationen ovan kan innebära att alla tecken i strängen gås igenom två gånger och behöver minnesutrymme för dubbla antalet tecken. Varför?
- Skapa ett palindromtest som går igenom elementen max en gång och som inte behöver extra minnesutrymme för en kopia av strängen. *Lösningssidé:* Jämför parvis första och sista, näst första och näst sista, etc.

**Uppgift 24.** *Fler användbara sekvenssamlingsmetoder.* Sök på webben och läs om dessa metoder och testa dem i REPL:

- `xs.tabulate(n)(f)`
- `xs.forall(p)`
- `xs.exists(p)`
- `xs.count(p)`
- `xs.zipWithIndex`

**Uppgift 25.** *Arrays don't behave, but ArraySeqs do!* Även om `Array` är primitiv så finns smart krångel "under huven" i Scalas samlingsbibliotek för att arrayer ska bete sig nästan som "riktiga" samlingar. Därmed behöver man inte ägna sig åt olika typer av specialhantering, t.ex. s.k. boxning, wrapperklasser och typomvandlingar (eng. *type casting*), vilket man ofta behöver kämpa med som Java-programmerare.

Dock finns fortfarande begränsningar och anomalier vad gäller till exempel likhetstest. Om du vill att en array ska bete sig som andra samlingar kan du enkelt "wrappa" den med metoden `toSeq` som vid anrop på arrayer ger en `ArraySeq`. Denna beter sig som en helt vanlig oföränderlig sekvenssamling utan att offra snabbheten hos en primitiv array.

```
val as = Array(1,2,3)
val xs = as.toSeq
```

- Hur fungerar likhetstest mellan två `ArraySeqs`? Vad har *xs* ovan för typ? Går det att uppdatera en wrappad array?
- Vilken typ av argumentsekvens får du tillgång till i kroppen för en funktion med repeterande parametrar?

<sup>5</sup><https://sv.wikipedia.org/wiki/Palindrom>

- ★ c) Läs här: <http://docs.scala-lang.org/overviews/collections/arrays.html> och ge ett exempel på vad mer man inte kan göra med en array, förutom innehållslighetstest.
- ★ **Uppgift 26.** *List eller Vector?* Jämför tidskomplexitet mellan List och Vector vid hantering i början och i slutet, baserat på efterföljande REPL-session i din egen körmiljö. Körningen nedan gjordes på en AMD Ryzen 7 5800X (16) @ 3.800GHz under Arch Linux 5.12.8-arch1-1 med Scala 3.0.1 och openjdk 11.0.11, men du ska använda det du har på din dator.

Hur snabbt går nedan på din dator? När är List snabbast och när är Vector snabbast? Hur stor är skillnaderna i prestanda? <sup>6</sup>

```
> head -5 /proc/cpuinfo
processor      : 0
vendor_id     : AuthenticAMD
cpu family    : 25
model         : 33
model name    : AMD Ryzen 7 5800X 8-Core Processor

scala> def time(n: Int)(block: => Unit): Double =
  |   def now = System.nanoTime
  |   var timestamp = now
  |   var sum = 0L
  |   var i = 0
  |   while i < n do
  |     block
  |     sum = sum + (now - timestamp)
  |     timestamp = now
  |     i = i + 1
  |   val average = sum.toDouble / n
  |   println("Average time: " + average + " ns")
  |   average

// Exiting paste mode, now interpreting.

time: (n: Int)(block: => Unit)Double

scala> val n = 100000
scala> val l = List.fill(n)(math.random())
scala> val v = Vector.fill(n)(math.random())

scala> (for i <- 1 to 20 yield time(n){l.take(10)}).min
average time: 97.66952 ns
average time: 91.90033 ns
average time: 79.88311 ns
average time: 69.5963 ns
average time: 69.69892 ns
average time: 69.8033 ns
average time: 69.7705 ns
average time: 69.68491 ns
average time: 69.54222 ns
average time: 69.66051 ns
average time: 69.73661 ns
average time: 69.54112 ns
average time: 69.69141 ns
average time: 69.46341 ns
average time: 69.4098 ns
average time: 61.34162 ns
average time: 41.1333 ns
```

<sup>6</sup>Denna typ av mätningar lär du dig mer om i LTH-kursen "Utvärdering av programvarusystem", som ges i slutet av årskurs 1 för Datateknikstudenter.

```
average time: 40.97051 ns
average time: 40.9075 ns
average time: 41.12321 ns
val res0: Double = 40.9075

scala> (for i <- 1 to 20 yield time(n){v.take(10)}).min
average time: 84.56978 ns
average time: 75.20167 ns
average time: 57.16529 ns
average time: 34.84469 ns
average time: 34.38478 ns
average time: 34.77709 ns
average time: 34.77179 ns
average time: 35.0506 ns
average time: 34.7967 ns
average time: 35.04258 ns
average time: 34.82559 ns
average time: 36.3673 ns
average time: 34.91029 ns
average time: 34.87239 ns
average time: 34.51958 ns
average time: 34.83949 ns
average time: 34.56169 ns
average time: 34.80719 ns
average time: 34.84459 ns
average time: 34.89468 ns
val res1: Double = 34.38478

scala> (for i <- 1 to 20 yield time(1000){l.takeRight(10)}).min
average time: 131365.106 ns
average time: 118632.787 ns
average time: 118440.066 ns
average time: 118687.567 ns
average time: 118428.487 ns
average time: 118871.686 ns
average time: 118964.797 ns
average time: 119030.236 ns
average time: 119262.534 ns
average time: 119228.344 ns
average time: 119226.494 ns
average time: 119310.933 ns
average time: 119352.854 ns
average time: 119121.913 ns
average time: 119133.664 ns
average time: 119015.193 ns
average time: 119276.674 ns
average time: 119224.882 ns
average time: 119301.771 ns
average time: 119444.401 ns
val res2: Double = 118428.487

scala> (for i <- 1 to 20 yield time(1000){v.takeRight(10)}).min
average time: 805.989 ns
average time: 365.219 ns
average time: 225.49 ns
average time: 125.92 ns
average time: 124.98 ns
average time: 130.689 ns
average time: 139.86 ns
average time: 128.29 ns
average time: 132.59 ns
average time: 125.729 ns
average time: 125.46 ns
average time: 130.59 ns
```

```
average time: 122.03 ns
average time: 121.9 ns
average time: 119.69 ns
average time: 120.48 ns
average time: 125.239 ns
average time: 126.09 ns
average time: 125.92 ns
average time: 125.91 ns
val res3: Double = 119.69
```

Varför går olika rundor i for-loopen olika snabbt även om varje runda gör samma sak?

★ **Uppgift 27.** *Tidskomplexitet för olika samlingar i Scalas standardbibliotek.*

Studera skillnader i tidskomplexitet mellan olika samlingar här:

[docs.scala-lang.org/overviews/collections/performance-characteristics.html](https://docs.scala-lang.org/overviews/collections/performance-characteristics.html)

Läs även kritiken av förenklingar i ovan beskrivning här:

[www.lihaoyi.com/post/ScalaVectoroperationsarentEffectivelyConstanttime.html](http://www.lihaoyi.com/post/ScalaVectoroperationsarentEffectivelyConstanttime.html)

Läs denna grundliga empirisk genomgång av prestanda i Scalas samlingsbibliotek:

[www.lihaoyi.com/post/BenchmarkingScalaCollections.html](http://www.lihaoyi.com/post/BenchmarkingScalaCollections.html)

Du får lära dig mer om hur man resonerar kring komplexitet i kommande kurser.



## 7.3 Laboration: shuffle

### Mål

- ☐ Kunna skapa och använda sekvenssamlingar.
- ☐ Kunna implementera sekvensalgoritmen SHUFFLE som modifierar innehållet i en array på plats.
- ☐ Kunna registrera antalet förekomster av olika värden i en sekvens.

### Förberedelser

- ☐ Gör övning sequences i avsnitt 7.2
- ☐ Läs igenom hela laborationen och säkerställ att du förstår hur SHUFFLE-algoritmen nedan fungerar.
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).

### 7.3.1 Bakgrund

Denna uppgift handlar om kortblandning. Att blanda kort så att varje möjlig permutation (ordning som korten ligger i) är lika sannolik är icke-trivialt; en osystematisk blandning leder till en skev fördelning.

Givet en bra slumpgenerator går det att blanda en kortlek genom att lägga alla kort i en hög och sedan ta ett slumpvist kort från högen och lägga det överst i leken, tills alla kort ligger i leken. Fisher-Yates-algoritmen<sup>7</sup> (även kallad Knuth-shuffle), fungerar på det sättet. Här benämner vi denna algoritm SHUFFLE. Den återfinns i pseudokod nedan. Notera speciellt att den övre gränsen för  $r$  inkluderar  $i$ .

**Indata** : Array  $xs$  med  $n$  st värden som ska blandas ”på plats”  
**Utdata** :  $xs$  uppdaterad på plats med sina värden omflyttade i slumpmässig ordning

```

1 for  $i \leftarrow (n - 1)$  to 0 do
2   dra slumptal  $r$  så att  $0 \leq r \leq i$ 
3   byt plats på  $xs(i)$  och  $xs(r)$ 
4 end
```

En kortlek (eng. *deck*) har 52 kort, vart och ett med olika valör (eng. *rank*) och färg (eng. *suit*, på svenska även svit). Kortspelet poker handlar om att dra kort och få upp vissa kombinationer av kort, s.k. ”händer”<sup>8</sup>. Dessa är ordnade från bättre till sämre; den spelare som får bäst hand vinner. Det är därför intressant att veta med vilken sannolikhet en viss hand dyker upp vid dragning från en blandad kortlek.

De vanliga pokerhänderna är, i fallande värde, färgstege (straight flush), fyrtal, kåk (full house), färg (flush), stege (straight), triss, tvåpar och par. Dessa finns illustrerade i avsnitt 7.3.4. Det finns ytterligare en hand, s.k. ”royal (straight) flush” som betecknar en färgstege med ess som högsta kort, men dess sannolikhet är för låg för att man vid simulering kan förväntas påträffa den inom rimlig tid.

Under laborationen ska du börja med att göra klar den ofärdiga klassen Deck som visas nedan, och återfinns i workspace på GitHub.

Labbinstruktionerna i avsnitt 7.3.2 ger tips om hur du ska ersätta ??? i givna kodskelett med dina lösningar. Med hjälp av klasserna TestHand och TestDeck kan du testa så att dina implementationer fungerar.

<sup>7</sup>[https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates\\_shuffle#The\\_modern\\_algorithm](https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_modern_algorithm)

<sup>8</sup><https://sv.wikipedia.org/wiki/Pokerhand>

```

1 package cards
2
3 case class Card(rank: Int, suit: Int):
4   import Card._
5
6   require(rankRange.contains(rank), s"rank=$rank, must be in $rankRange")
7   require(suitRange.contains(suit), s"suit=$suit, must be in $suitRange")
8
9   val rankString: String = ranks(rank - 1)
10  val suitChar: Char = suits(suit - 1)
11
12  override def toString() = s"$rankString$suitChar "
13
14 object Card:
15   val suitRange: Range = 1 to 4
16   val rankRange: Range = 1 to 13
17   val suits: Vector[Char] = "♠♥♣♦".toVector
18   val ranks: Vector[String] =
19     "A" +: ((2 to 10).map(_.toString).toVector ++ Vector("J", "Q", "K"))

```

**Figur 7.2:** Den färdigimplementerade, oföränderliga case-klassen Card.

När dina implementationer av metoderna full och shuffle fungerar ska du använda Deck i singelobjektet PokerProbability för att ta reda på sannolikheter för att olika pokerhänder uppkommer när man delar ut 5 kort ur en bra blandad kortlek.

Till din hjälp har du nedan kodfiler, där några har ofärdig kod som du ska färdigställa. All kod ligger i ett paket med namnet cards.<sup>9</sup>

- Card.scala i fig. 7.2 innehåller den färdigimplementerade case-klassen Card som representerar ett kort och har en koncis toString med valör och svit (färg).
- Deck.scala i fig. 7.3 innehåller den förändringsbara klassen Deck, där du ska implementera kortblandning i metoden shuffle. Kompanjonsobjektet har metoder för att skapa kortlekar. Du ska implementera metoden full som skapar en fullständig kortlek med de 52 korten ordnade efter valör och färg.
- Hand.scala i fig. 7.4 innehåller en case-klass Hand som representerar en pokerhand och har metoder för att avgöra vilken hand det är. I kompanjonsobjektet finns fabriksmetoder som kan skapa en ny hand från enskilda kort eller genom att dra kort ur en kortlek. Du ska implementera tally som registrerar antalet kort av en viss valör.
- PokerProbability.scala i fig. 7.5 har en main-metod som räknar ut pokersannolikheter, samt hjälpmetoden register som du ska implementera.
- TestDeck.scala ska du använda för att testa din implementation av shuffle med en kortlek som endast innehåller tre kort. Upprepade blandningar görs och förekomsten av varje möjlig permutation registreras.
- TestHand.scala har en main-metod som testat klassen Hand.

## 7.3.2 Obligatoriska uppgifter

<sup>9</sup>Du kan bläddra bland klasserna i paketet cards här:

[https://github.com/lunduniversity/introprog/tree/master/workspace/w07\\_shuffle/](https://github.com/lunduniversity/introprog/tree/master/workspace/w07_shuffle/)

```
1 package cards
2
3 class Deck private (val initCards: Vector[Card]):
4   private var cards: Array[Card] = initCards.toArray
5
6   def reset(): Unit = cards = initCards.toArray
7   def apply(i: Int): Card = cards(i)
8   def toVector: Vector[Card] = cards.toVector
9   override def toString: String = cards.mkString(" ")
10
11   def peek(n: Int): Vector[Card] = cards.take(n).toVector
12
13   def remove(n: Int): Vector[Card] =
14     val init = peek(n)
15     cards = cards.drop(n)
16     init
17
18   def prepend(moreCards: Card*): Unit = cards = moreCards.toArray ++ cards
19
20   /** Swaps cards at position a and b. */
21   def swap(a: Int, b: Int): Unit = ???
22
23   /** Randomly reorders the cards in this deck. */
24   def shuffle(): Unit = ???
25
26 object Deck:
27   def empty: Deck = new Deck(Vector())
28   def apply(cards: Seq[Card]): Deck = new Deck(cards.toVector)
29
30   /** Creates a new full Deck with 52 cards in rank and suit order. */
31   def full(): Deck = ???
```

**Figur 7.3:** Den ofärdiga klassen Deck med förändringsbar kortlek.

```

1 package cards
2
3 case class Hand(cards: Vector[Card]):
4   import Hand._
5
6   /**
7    * A vector of length 14 with positions 1-13 containing the number of
8    * cards of that rank. Position 0 contains 0.
9    */
10  def tally: Vector[Int] = ???
11
12  def ranksSorted: Vector[Int] = cards.map(_.rank).sorted.toVector
13
14  def isFlush: Boolean = cards.length > 0 && cards.forall(_.suit == cards(0).suit)
15
16  def isStraight: Boolean =
17    def isInSeq(xs: Vector[Int]): Boolean =
18      xs.length > 1 && (0 to xs.length - 2).forall(i => xs(i) == xs(i + 1) - 1)
19
20    isInSeq(ranksSorted) || // special case with ace interpreted as 14:
21      (ranksSorted(0) == 1) && isInSeq(ranksSorted.drop(1) :+ 14)
22
23  def isStraightFlush: Boolean = isStraight && isFlush
24  def isFour: Boolean = tally.contains(4)
25  def isFullHouse: Boolean = tally.contains(3) && tally.contains(2)
26  def isThrees: Boolean = tally.contains(3)
27  def isTwoPair: Boolean = tally.count(_ == 2) == 2
28  def isOnePair: Boolean = tally.contains(2)
29
30  def category: Int = // TODO: add more tests when tally is implemented
31    if isStraight && isFlush then Category.StraightFlush
32    else if isFlush then Category.Flush
33    else if isStraight then Category.Straight
34    else Category.HighCard
35
36 object Hand:
37   def apply(cardSeq: Card*): Hand = new Hand(cardSeq.toVector)
38   def from(deck: Deck): Hand = new Hand(deck.peek(5))
39   def removeFrom(deck: Deck): Hand = new Hand(deck.remove(5))
40
41 object Category:
42   val RoyalFlush = 0
43   val StraightFlush = 1
44   val Fours = 2
45   val FullHouse = 3
46   val Flush = 4
47   val Straight = 5
48   val Threes = 6
49   val TwoPair = 7
50   val OnePair = 8
51   val HighCard = 9
52   val values = RoyalFlush to HighCard
53
54 object Name:
55   val english = Vector("royal flush", "straight flush", "four of a kind", "full house",
56     "flush", "straight", "three of a kind", "two pairs", "pair", "high card")
57   val swedish = Vector("royal flush", "färgstege", "fyrtal", "kåk", "färg",
58     "stege", "tretal", "två par", "par", "högt kort")

```

**Figur 7.4:** Den ofärdiga, oföränderliga klassen Hand som representerar en pokerhand.

```

1 package cards
2
3 object PokerProbability:
4   /**
5    * For a given number of iterations, shuffles a deck, draws a hand and
6    * returns a vector with the frequency of each hand category.
7    */
8   def register(n: Int, deck: Deck): Vector[Int] = ???
9
10  def main(args: Array[String]): Unit =
11    val n = scala.io.StdIn.readLine("number of iterations: ").toInt
12    val deck = Deck.full()
13    val frequencies = register(n, deck)
14    for i <- Hand.Category.values do
15      val name = Hand.Category.Name.english(i).capitalize
16      val percentages = frequencies(i).toDouble / n * 100
17      println(f"$name%16s $percentages%10.6f%")

```

**Figur 7.5:** Det ofärdiga singelobjektet `PokerProbability` som tar reda på sannolikheter för olika pokerhänder.

### Uppgift 1. Implementera algoritmen SHUFFLE.

- Skapa din egen implementation av metoden `shuffle` i klassen `Deck`. Följ den givna algoritmen i stycke 7.3.1 noga. Du kan använda `cards.length` för att få fram längden på kortleken, men du kan gärna istället använda `cards.indices.reverse`. Implementera och använd metoden `swap`.
- Kör `testShuffle` i `TestDeck` som kontrollerar att blandningen är jämnt fördelad genom att blanda en kortlek med tre kort och räkna hur ofta varje möjlig permutation dyker upp. Du bör få en utskrift med sex (3!) procentsatser som ska vara nästan lika.

### Uppgift 2. Skapa en fullständig, ordnad kortlek.

- Implementera metoden `full` som skapar en 52-korts standardlek ordnad efter färg och valör. Använd `Range`-värdena i kompanjonsobjektet `Card`.
- Kör `testCreate` i `TestDeck` och kontrollera så att du får kort av alla fyra färger, samt både ess och kungar.

### Uppgift 3. Gör färdigt och testa `Hand`.

- Implementera `tally` som ska ge en indexerbar sekvens med 14 platser där plats 1-13 innehåller antalet av respektive valör. (Plats 0 ska inte användas.)
- Testa klassen `Hand` med hjälp av `TestHand`.

### Uppgift 4. Ta fram sannolikheterna för "straight flush", "straight" och "flush".

- Implementera metoden `register` i `PokerProbability`. Använd `from` och `category` i `Hand` för att skapa och kategorisera en hand från en kortlek. Lagra frekvenserna i en lokal array som du, när resultatet är färdigt, gör om till en vektor med `toVector`.
- Kör `PokerProbability`, förslagsvis med en miljon iterationer. Du bör få ungefär dessa sannolikheter<sup>10</sup>:

<sup>10</sup><http://www.forum.gpcdata.se/pdf/poker.pdf>

<i>hand</i>	<i>sannolikhet</i>
Straight flush	0.00154%
Flush	0.197%
Straight	0.39%

**Uppgift 5.** Försök tänka ut så många ställen som möjligt i din kod där du skulle kunna använda **enum** och skissa översiktligt med papper och penna hur koden vid ett av dessa ställen skulle kunna se ut. Diskutera med handledare för- och nackdelar med att använda **enum** istället för heltals- eller strängsekvenser.

### 7.3.3 Frivilliga extrauppgifter

**Uppgift 6.** Kopiera hela din lösning till en ny katalog och ändra implementationen så att du drar nytta av uppräknade datatyper med **enum** i stället för heltal och strängsekvenser på alla ställen där det är möjligt och lämpligt. Vilka är för- och nackdelar med de två olika implementationerna? Är det någon skillnad i exekveringstid?

**Uppgift 7.** Förbättra programmet så att simuleringen registrerar alla handkategorier utom Royal Flush. Kör sedan `PokerProbability` igen och notera sannolikheterna.

**Uppgift 8.** Gör om alla metoder i case-klassen `Hand` till **lazy val** och undersök hur det påverkar exekveringstiden. Varför förändras prestanda med denna åtgärd? Hade denna optimering varit lämplig om klassen `Hand` vore förändringsbar? Varför?

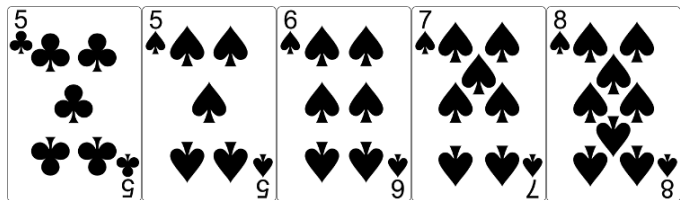
**Uppgift 9.** Gör så att även sannolikheten för Royal Flush kan simuleras. Det krävs i storleksordningen  $10^8$  iterationer för en noggrannhet på 2 värdesiffror. Detta kan ta ca 5 minuter på en någorlunda snabb dator, så det kan vara läge före en paus under simuleringen...

**Uppgift 10.** Implementera ett interaktivt kortspel, t.ex. någon enkel pokervariant. Börja med något mycket enkelt, till exempel högst-kort-vinner, och bygg vidare med sådant som du tycker verkar roligt.

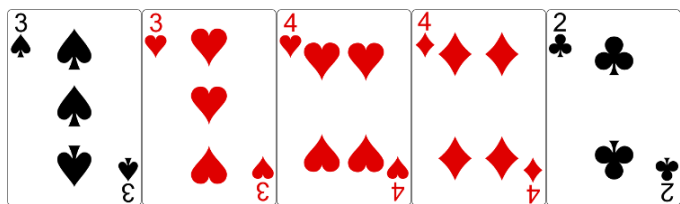
Du kan t.ex. skapa en metod **def** `compareTo(other: Hand): Comparison` i case-klassen `Hand` som ger `Comparison.Worse` om `other` är sämre, `Comparison.Equal` om händerna är lika bra, och `Comparison.Better` om `other` är bättre. Du kan steg för steg göra så att det går att jämföra fler och fler händer enligt de specialregler som gäller för när olika händer anses bättre eller lika. Läs om reglerna här: [https://en.wikipedia.org/wiki/List\\_of\\_poker\\_hands](https://en.wikipedia.org/wiki/List_of_poker_hands)

### 7.3.4 Bilder med exempel på olika pokerhänder

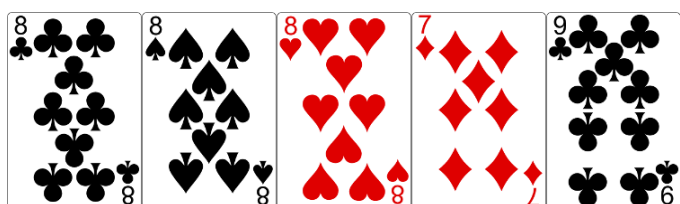
Figurerna 7.6 – 7.14 visar bilder på olika korthänder i poker.



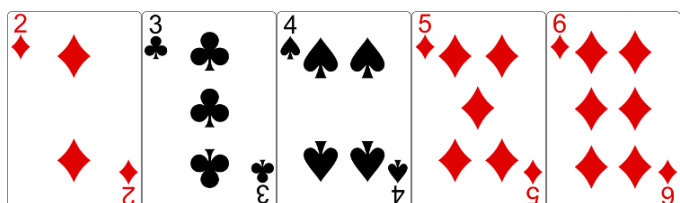
**Figur 7.6:** Par: två kort har samma valör.



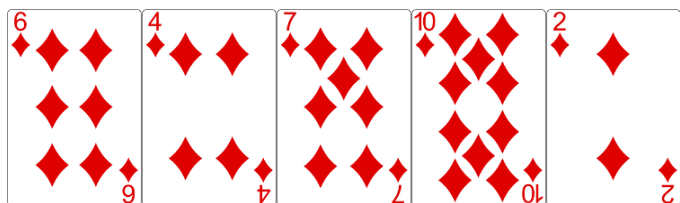
**Figur 7.7:** Två par: handen har två *olika* par.



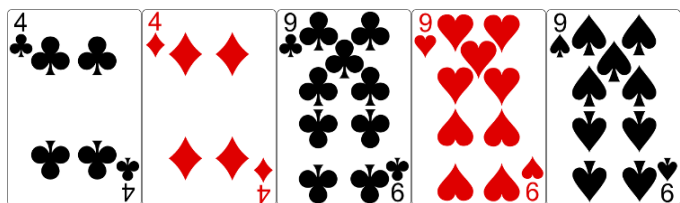
**Figur 7.8:** Tripp: tre kort har samma valör.



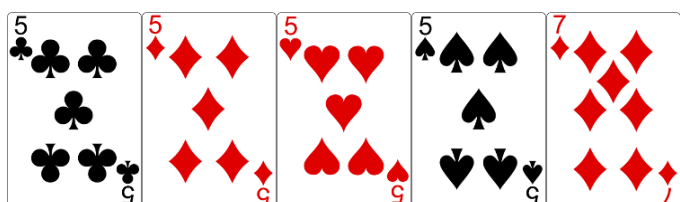
**Figur 7.9:** Stege: kortens valörer bildar en följd, ess kan vara antingen 1 eller 14.



**Figur 7.10:** Färg: alla kort har samma färg.



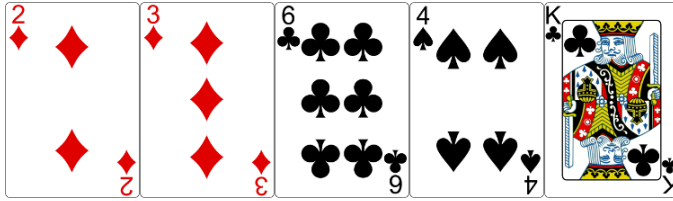
**Figur 7.11:** Kåk: både tripp och par.



**Figur 7.12:** Fyrtalet: fyra kort har samma valör.



**Figur 7.13:** Färgsteg: både steg och färg.



**Figur 7.14:** Högst kort: inget mönster finns.



## Kapitel 8

# Nästlade och generiska strukturer

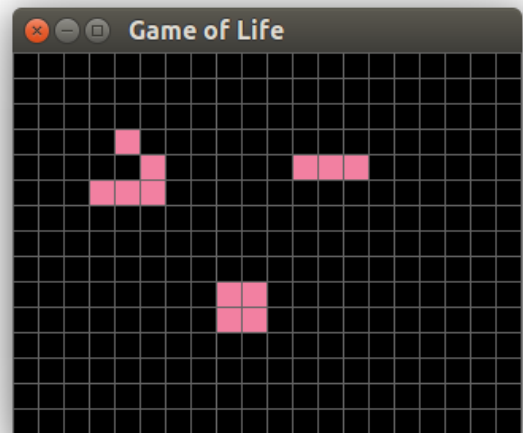
Begrepp som ingår i denna veckas studier:

- ☐ matris
- ☐ nästlad samling
- ☐ nästlad for-sats
- ☐ typparameter
- ☐ generisk funktion
- ☐ generisk klass
- ☐ fri och bunden typparameter
- ☐ generiska datastrukturer
- ☐ generiska samlingar i Scala

## 8.1 Teori

### 8.1.1 Veckans labb: Life

- Universum är en binär matris av **celler** där **levande** celler representeras med **true** och **döda** med **false**.
- Följande regler gäller för **nästa generation** celler i universum:
  - **Fortlevnad**: en levande cell med 2 eller 3 grannar **lever vidare**
  - **Död**: en levande cell med färre än 2 eller fler än 3 grannar **dör**
  - **Födelse**: en död cell med exakt tre grannar föds
- Övning matrices uppgift 5: skapa en generisk **case class** Matrix[T]
- På labben: använd Matrix[Boolean]



- Du ska simulera *Game of Life* i ett introprog.PixelWindow
- Fördjupning:  
[https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)

### 8.1.2 Vad är en matris?

- En **matris** inom **matematiken** innehåller **rader** och **kolumner**<sup>1</sup> med tal.
- I en **matematisk** matris har alla rader **lika många** element och
- även alla kolumner har **lika många** element.
- En matris av dimension  $2 \times 5$  har  $2 \cdot 5 = 10$  stycken element.
- Exempel på en matematisk matris av dimension  $2 \times 5$ :

$$M_{2,5} = \begin{pmatrix} 5 & 2 & 42 & 4 & 5 \\ 3 & 4 & 18 & 6 & 7 \end{pmatrix}$$

### 8.1.3 Indexering i en matris

- En matris av dimension  $m \times n$  har  $m \cdot n$  stycken element.
- En matris  $A_{m,n}$  av dimension  $m \times n$  ritas inom matematiken ofta så här:

<sup>1</sup>även kallade *kolonner*

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

- Matrisindexering inom matematiken sker ofta från 1, men ofta från 0 i datorprogram.
- Vad har talet 42 för index i matrisen  $M_{2,5}$  nedan?
  - Inom matematiken?
  - I Scala och Java och många andra språk?

$$M_{2,5} = \begin{pmatrix} 5 & 2 & 42 & 4 & 5 \\ 3 & 4 & 18 & 6 & 7 \end{pmatrix}$$

#### 8.1.4 Hur skapa matriser?

- Inom programmering används ordet **matris** ofta för att beteckna en **nästlad struktur** i två dimensioner. Exempel:
  - **Oföränderliga** sekvenser, t.ex. `Vector[Vector[Int]]`  
`val xss = Vector(Vector(0, 0, 0), Vector(0, 0, 0))` eller enklare:  
`val xss = Vector.fill(2,3)(0)`
  - **Föränderliga** sekvens, t.ex. `Array[Array[Int]]`  
`val yss = Array(Array(0, 0, 0), Array(0, 0, 0))` eller enklare:  
`val yss = Array.fill(2,3)(0)`

#### 8.1.5 Hur indexera i matriser?

En matris med array av arrayer:

```
1 scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
2 xss: Array[Array[Int]] = Array(Array(5, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
```

Man indexerar i en nästlad sekvens med upprepad apply:

```
1 scala> xss(0)(2)
2 res0: ???
3
4 scala> xss.apply(0).apply(2)
5 res1: ???
6
7 scala> xss(0)
8 res2: ???
```

Övning: Vad är typ och värde vid ??? ovan?

---

### 8.1.6 Hur indexera i matriser?

En matris med array av arrayer:

```
1 scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
2 xss: Array[Array[Int]] = Array(Array(5, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
```

Man indexerar i en nästlad sekvens med upprepad apply:

```
1 scala> xss(0)(2)
2 res0: Int = 42
3
4 scala> xss.apply(0).apply(2)
5 res1: Int = 42
6
7 scala> xss(0)
8 res2: Array[Int] = Array(5, 2, 42, 4, 5)
```

Övning: Rita en bild av minnet som referensen xss refererar till.

---

### 8.1.7 Uppdatering av en förändringsbar nästlad struktur

Man kan förändra en array av arrayer ”på plats” med tilldelning:

```
1 scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
2
3 scala> xss(0)(0) = 100
4
5 scala> xss
6 res0: ???
7
8 scala> xss(0)(2) = xss(0)(2) - 1
9
10 scala> xss
11 res1: ???
12
13 scala> xss(1) = Array.fill(5)(-1)
14
15 scala> xss
16 res2: ???
```

### 8.1.8 Uppdatering av en förändringsbar nästlad struktur

Man kan förändra en array av arrayer ”på plats” med tilldelning:

```
1 scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
2
3 scala> xss(0)(0) = 100
4
5 scala> xss
6 res0: Array[Array[Int]]=Array(Array(100, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
7
8 scala> xss(0)(2) = xss(0)(2) - 1
```

```

9
10 scala> xss
11 res1: Array[Array[Int]]=Array(Array(100, 2, 41, 4, 5), Array(3, 4, 18, 6, 7))
12
13 scala> xss(1) = Array.fill(5)(-1)
14
15 scala> xss
16 res2: Array[Array[Int]]=Array(Array(100, 2, 41, 4, 5), Array(-1,-1,-1,-1,-1))

```

### 8.1.9 Några olika sätt att skapa förändringsbara matriser

Det jobbiga, primitiva sättet:

```

1 scala> val xss = new Array[Array[Int]](2)
2 xss: Array[Array[Int]] = Array(null, null)
3
4 scala> for (i <- xss.indices) {xss(i) = new Array[Int](5)}
5
6 scala> xss
7 res0: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))
8
9 scala> println(xss)
10 [[I@196a99d0

```

Enklare sätt:

```

1 scala> val xss = Array.ofDim[Int](2,5)
2 xss: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))

```

Enklare och tydligare sätt, där initialvärdet anges explicit:

```

1 scala> val xss = Array.fill(2,5)(0)
2 xss: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))

```

### 8.1.10 Exempel på skapande av oföränderlig nästlad struktur

Om du kan beräkna initialvärde direkt, använd `Vector.fill`:

**def** fill[A](n1: Int, n2: Int)(elem: => A): Vector[Vector[A]]

```

1 scala> Vector.fill(2,5)(scala.util.Random.nextInt(6) + 1)
2 res0:
3   typ???
4   värde???

```

Om du kan beräkna initialvärde ur index, använd `Vector.tabulate`:

**def** tabulate[A](n1: Int, n2: Int)(f: (Int, Int) => A): Vector[Vector[A]]

```

1 scala> Vector.tabulate(5,2)((x,y) => x + y + 1)
2 res1:
3   typ???
4   värde???

```

### 8.1.11 Exempel på skapande av oföränderlig nästlad struktur

Om du kan beräkna initialvärde direkt, använd `Vector.fill`:

```
def fill[A](n1: Int, n2: Int)(elem: => A): Vector[Vector[A]]
```

```
1 scala> Vector.fill(2,5)(scala.util.Random.nextInt(6) + 1)
2 res0: Vector[Vector[Int]] =
3   Vector(Vector(1, 2, 6, 2, 1), Vector(1, 4, 3, 3, 2))
```

Om du kan beräkna initialvärde ur index, använd `Vector.tabulate`:

```
def tabulate[A](n1: Int, n2: Int)(f: (Int, Int) => A): Vector[Vector[A]]
```

```
1 scala> Vector.tabulate(5,2)((x,y) => x + y + 1)
2 res1: Vector[Vector[Int]] =
3   Vector(Vector(1,2), Vector(2,3), Vector(3,4), Vector(4,5), Vector(5, 6))
```

### 8.1.12 Uppdatering av en oföränderlig nästlad struktur

Uppdatering av endimensionell struktur med `xs.updated`:

```
def updated[A](index: Int, elem: A): Vector[A]
```

```
1 scala> var xs = Vector.tabulate(5)(x => x + 1)
2 xs: typ??? = värde???
3
4 scala> xs = xs.updated(1, 42)
5 xs: typ??? = värde???
```

Uppdatering av nästlad struktur i två dimensioner:

```
1 scala> var xss = Vector.tabulate(2, 5)((x,y) => x + y + 1)
2 xss:
3   typ??? =
4   värde???
5
6 scala> xss = xss.updated(0, xss(0).updated(1, 42))
7 xss:
8   typ??? =
9   värde???
```

### 8.1.13 Uppdatering av en oföränderlig nästlad struktur

Uppdatering av endimensionell struktur med `xs.updated`:

```
def updated[A](index: Int, elem: A): Vector[A]
```

```
1 scala> var xs = Vector.tabulate(5)(x => x + 1)
2 xs: Vector[Int] = Vector(1, 2, 3, 4, 5)
3
4 scala> xs = xs.updated(1, 42)
5 xs: Vector[Int] = Vector(1, 42, 3, 4, 5)
```

Uppdatering av nästlad struktur i två dimensioner:

```
1 scala> var xss = Vector.tabulate(2, 5)((x,y) => x + y + 1)
2 xss: Vector[Vector[Int]] =
3   Vector(Vector(1, 2, 3, 4, 5), Vector(2, 3, 4, 5, 6))
```

```

4
5 scala> xss = xss.updated(0, xss(0).updated(1, 42))
6 xss:
7   Vector[Vector[Int]] =
8   Vector(Vector(1, 42, 3, 4, 5), Vector(2, 3, 4, 5, 6))

```

#### 8.1.14 Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.

Exempel: iterera med nästlad **for**-sats för utskrift av denna matris

**val** xss = Vector.tabulate(2,5)((x,y) => x + y + 1)

```

1 scala> for ??? do
2     for ??? do
3         print(xss(i)(j))
4         print(" ")
5     println
6
7 1 2 3 4 5
8 2 3 4 5 6

```

Övning:

Vad ska det stå vid ??? för att alla element ska skrivas ut?

#### 8.1.15 Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.

Exempel: iterera med nästlad **for**-sats för utskrift av denna matris

**val** xss = Vector.tabulate(2,5)((x,y) => x + y + 1)

```

1 scala> for xs <- xss do
2     for x <- xs do
3         print(x)
4         print(" ")
5     end for
6     println()
7 end for
8
9 1 2 3 4 5
10 2 3 4 5 6

```

Övning: skriv ut matrisen med nästlad foreach

```

xss.foreach { xs =>
  xs.foreach { x => print(x); print(" ") }
  println()
}

```

### 8.1.16 Övningsexempel: Yatzy

Skapa en funktion `roll` som ger utfallet av `n` st tärningskast:

```
1 scala> import scala.util.Random
2
3 scala> def roll(n: Int): Vector[Int] = ???
```

Skapa en funktion `isYatzy` som ger **true** om alla utfall är lika:

```
1 scala> def isYatzy(xs: Vector[Int]): Boolean = ???
```

Du kan anta att `xs.length > 0`

Tips: använd metoden `xs.forall`:

```
def forall[A](p: A => Boolean): Boolean
```

---

### 8.1.17 Övningsexempel: Yatzy

Skapa en funktion `roll` som ger utfallet av `n` st tärningskast:

```
1 scala> import scala.util.Random
2
3 scala> def roll(n: Int): Vector[Int] = Vector.fill(n)(Random.nextInt(6) + 1)
```

Skapa en funktion `isYatzy` som ger **true** om alla utfall är lika:

```
1 scala> def isYatzy(xs: Vector[Int]): Boolean = xs.forall(x => x == xs(0))
```

Du kan anta att `xs.length > 0`

Tips: använd metoden `xs.forall`:

```
def forall[A](p: A => Boolean): Boolean
```

---

### 8.1.18 Iterera över nästlad struktur: for-sats

Iterera med nästlad for-sats: (vad har `xss` för typ?)

```
1 scala> val xss = Vector.fill(100)(roll(5))
2
3 scala> for (i <- ???) do
4     for (j <- ???) do
5         print(s"($i)($j): ${xss(i)(j)} ")
6         println(s" YATZY: ${isYatzy(xss(i))}")
7
8 (0)(0): 3 (0)(1): 6 (0)(2): 4 (0)(3): 4 (0)(4): 6 YATZY: false
9 (1)(0): 4 (1)(1): 1 (1)(2): 5 (1)(3): 2 (1)(4): 6 YATZY: false
10 (2)(0): 1 (2)(1): 3 (2)(2): 5 (2)(3): 6 (2)(4): 2 YATZY: false
11 (3)(0): 2 (3)(1): 1 (3)(2): 1 (3)(3): 5 (3)(4): 4 YATZY: false
12 (4)(0): 4 (4)(1): 4 (4)(2): 1 (4)(3): 6 (4)(4): 5 YATZY: false
13 (5)(0): 3 (5)(1): 3 (5)(2): 2 (5)(3): 3 (5)(4): 6 YATZY: false
14 (6)(0): 3 (6)(1): 6 (6)(2): 1 (6)(3): 1 (6)(4): 4 YATZY: false
15 (7)(0): 6 (7)(1): 2 (7)(2): 4 (7)(3): 4 (7)(4): 3 YATZY: false
16 (8)(0): 1 (8)(1): 5 (8)(2): 4 (8)(3): 2 (8)(4): 4 YATZY: false
17 (9)(0): 1 (9)(1): 1 (9)(2): 3 (9)(3): 6 (9)(4): 6 YATZY: false
18 (10)(0): 2 (10)(1): 5 (10)(2): 2 (10)(3): 4 (10)(4): 5 YATZY: false
19 (11)(0): 3 (11)(1): 4 (11)(2): 2 (11)(3): 5 (11)(4): 6 YATZY: false
20 ...
```

---



### 8.1.19 Iterera över nästlad struktur: for-sats

Iterera med nästlad for-sats: (xss är en Vector[Vector[Int]])

```

1 scala> val xss = Vector.fill(100)(roll(5))
2
3 scala> for (i <- xss.indices) do
4     for (j <- xss(i).indices) do
5         print(s"($i)($j): ${xss(i)(j)} ")
6         println(s" YATZY: ${isYatzy(xss(i))}")
7
8 (0)(0): 3 (0)(1): 6 (0)(2): 4 (0)(3): 4 (0)(4): 6 YATZY: false
9 (1)(0): 4 (1)(1): 1 (1)(2): 5 (1)(3): 2 (1)(4): 6 YATZY: false
10 (2)(0): 1 (2)(1): 3 (2)(2): 5 (2)(3): 6 (2)(4): 2 YATZY: false
11 (3)(0): 2 (3)(1): 1 (3)(2): 1 (3)(3): 5 (3)(4): 4 YATZY: false
12 (4)(0): 4 (4)(1): 4 (4)(2): 1 (4)(3): 6 (4)(4): 5 YATZY: false
13 (5)(0): 3 (5)(1): 3 (5)(2): 2 (5)(3): 3 (5)(4): 6 YATZY: false
14 (6)(0): 3 (6)(1): 6 (6)(2): 1 (6)(3): 1 (6)(4): 4 YATZY: false
15 (7)(0): 6 (7)(1): 2 (7)(2): 4 (7)(3): 4 (7)(4): 3 YATZY: false
16 (8)(0): 1 (8)(1): 5 (8)(2): 4 (8)(3): 2 (8)(4): 4 YATZY: false
17 (9)(0): 1 (9)(1): 1 (9)(2): 3 (9)(3): 6 (9)(4): 6 YATZY: false
18 (10)(0): 2 (10)(1): 5 (10)(2): 2 (10)(3): 4 (10)(4): 5 YATZY: false
19 (11)(0): 3 (11)(1): 4 (11)(2): 2 (11)(3): 5 (11)(4): 6 YATZY: false
20 ...

```

### 8.1.20 Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```

1 scala> val xss = for (i <- 1 to 2) yield
2     for (j <- 1 to 5) yield i + j + 1
3
4 val xss: IndexedSeq[IndexedSeq[Int]] =
5     ???

```

Om man skriver så här får man en endimensionell struktur:

```

1 scala> val xs = for (i <- 1 to 2; j <- 1 to 5) yield i + j + 1
2 val xs: IndexedSeq[Int] =
3     ???

```

### 8.1.21 Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```

1 scala> val xss = for (i <- 1 to 2) yield {
2     for (j <- 1 to 5) yield i + j + 1
3 }
4 val xss: IndexedSeq[IndexedSeq[Int]] =
5     Vector(Vector(3, 4, 5, 6, 7), Vector(4, 5, 6, 7, 8))

```

Om man skriver så här får man en endimensionell struktur:

```

1 scala> val xs = for (i <- 1 to 2; j <- 1 to 5) yield i + j + 1
2 val xs: IndexedSeq[Int] =
3   Vector(3, 4, 5, 6, 7, 4, 5, 6, 7, 8)

```

### 8.1.22 Nästlade map-uttryck

Iterera med **nästlade map-uttryck**:

```

1 scala> val xss = (1 to 2).map(i => (1 to 5).map(j => i + j + 1))
2 xss: IndexedSeq[IndexedSeq[Int]] =
3   ???

```

### 8.1.23 Nästlade map-uttryck

Iterera med **nästlade map-uttryck**:

```

1 scala> val xss = (1 to 2).map(i => (1 to 5).map(j => i + j + 1))
2 xss: IndexedSeq[IndexedSeq[Int]] =
3   Vector(Vector(3, 4, 5, 6, 7), Vector(4, 5, 6, 7, 8))

```

### 8.1.24 Fallgrop: likhet av array

```

1 scala> Vector.fill(5, 2)(42) == Vector.fill(5, 2)(42)
2 val res0: Boolean = true
3
4 scala> Array.fill(5, 2)(42) == Array.fill(5, 2)(42)
5 val res1: Boolean = false // AAAARRGH!!! :(

```

Primitiva arrayer har en equals-metod som ger referenslikhet, **inte** innehållslikhet. Och det fungerar följaktligen ej heller på nästlade strukturer.

### 8.1.25 Övning: Kolla likhet av array (uppfinner hjulet)

```

def isEqual(xss: Array[Array[Int]], yss: Array[Array[Int]]) =
  var i = 0
  var foundUnequal = false
  while i < xss.length && !foundUnequal do
    var j = 0
    while j < xss(i).length && !foundUnequal do
      if xss(i)(j) != yss(i)(j) then foundUnequal = true
      j += 1
    end while
    i += 1
  end while

```

```

    i += 1
  end while
  !foundUnequal
end isEqual

```

```

1 scala> val (xss, yss) = (Array.fill(5,2)(42), Array.fill(5,2)(42))
2
3 scala> isEqual(xss, yss) // true
4
5 scala> yss(4)(1) = 0
6
7 scala> isEqual(xss, yss) // false

```

### 8.1.26 Använd sameElements för test av innehållslighet men bara på icke-nästlade arrayer

I Scala kan du använda metoden sameElements på arrayer för innehållslighet, men det funkar **INTE** på nästlade strukturer.

```

1 scala> val xs = Array(1,2,3)
2 xs: Array[Int] = Array(1, 2, 3)
3
4 scala> val ys = Array(1,2,3)
5 ys: Array[Int] = Array(1, 2, 3)
6
7 scala> xs.sameElements(ys)
8 res0: Boolean = true
9
10 scala> Array(Array(1)) sameElements Array(Array(1))
11 res1: Boolean = false

```

Använd i stället: `java.util.Arrays.deepEquals(xs, ys)` men det kan då behövas `.asInstanceOf[Array[Object]]` på argumenten om kompilatorn inte klarar typkonverteringen.

### 8.1.27 Om veckans övningar

- Träna på att iterera över nästlade strukturer
- Fortsätt jobba med Yatzy-exemplet
- träna på att skapa **imperativa** algoritmer:  
lös isYatzy med **while**-sats
- Extrauppgift där du ska bygga ett enkelt yatzy-spel i terminalen (kunde varit en tentauppgift...)

### 8.1.28 Exempel: Icke-generisk case-klass med helfalsmatris

En *icke-generisk* datastruktur har inga obundna typparametrar; alla typer är **konkreta** (alltså specifika).

En icke-generisk case-class med en `Vector[Vector[Int]]`:

```
case class Matrix(data: Vector[Vector[Int]]):
  def apply(x: Int, y: Int): Int = data(x)(y)
```

```
1 scala> Matrix(Vector(Vector(5, 2, 42, 4, 5), Vector(3, 4, 18, 6, 7)))
2 res0: Matrix =
3   Matrix(Vector(Vector(5, 2, 42, 4, 5), Vector(3, 4, 18, 6, 7)))
4
```

### 8.1.29 Exempel: Generisk case-klass med generell matris

En *generisk* datastruktur har minst en obunden **typparameter** som kan bindas till ett **konkret typargument**.

```
case class Matrix[T](data: Vector[Vector[T]]):
  def apply(x: Int, y: Int): T = data(x)(y)
```

Matrix i exemplet ovan är en **generisk** case-class där T är obunden, eftersom T är en typparameter deklarerad inom [] **efter** klassens namn men **före** klassparameterlistan.

Användning där T binds till Int via kompilatorns typhärledning:

```
1 scala> Matrix(Vector(Vector(5, 2, 42, 4, 5), Vector(3, 4, 18, 6, 7)))
2 res1: Matrix[Int] =
3   Matrix(Vector(Vector(5, 2, 42, 4, 5), Vector(3, 4, 18, 6, 7)))
4
```

### 8.1.30 Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *parametric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

- En **fri** typparameter kan bindas till vilken typ som helst.
- Bindningen av typargument till typparametrar sker vid **kompileringstid**.
- En typparameter är **fri** om den **inte** fått något värde, annars **bunden**.
- Exempel: **generisk klass** med **generiska metoder**:

```
class Cell[A]( // [A] är fri (måste bindas vid användning)
  var value: A): // A är bunden
  def update(a: A): Unit = value = a // A är bunden
  def replaced[B](b: B): Cell[B] = new Cell(b) // första [B] är fri
```

- **Skuggning kan förekomma:** Om `replaced` i `Cell` hade använt namnet `A` på sin typparameter hade den **skuggat** klassens typparameter och tolkats som en fri typparameter, alltså en godtycklig typ och **inte** klassens typparameter. (jämför namnöverskuggning vid **lokala** namn i nästlade block)

### 8.1.31 Exempel: Generisk funktion

Vad händer här?

```

1
2 scala> def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
3 1 |def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
4   |                               ^
5   |                               Not found: type T
6   |                               ^
7
8 scala> def skrikBaklänges[T](x: T): String = x.toString.toUpperCase.reverse
9
10 scala> skrikBaklänges("gurka är gott")
11 val res0: String = TTOG RÅ AKRUG

```

Om ingen typparameter deklareraras inom hakparenteser efter funktionens namns så vet inte kompilatorn vad `T` är för en typ. Men med en typparameter `[T]` efter funktionsnamnet tolkar kompilatorn funktionen som **generisk** och typen `T` bestäms av argumentets typ **vid anrop** och `T` kan bindas till godtycklig typ.

### 8.1.32 Exempel: Generisk case-klass

En generisk klass har en eller flera typparametrar efter klassnamnet:

```
case class Box[A](value: A)
```

Kompilatorn härleder typparameterarnas typ utifrån givna värden.

```

1 scala> Box("gurka")
2 val res1: Box[String] = Box(gurka)

```

Du kan också ge typparametern en typ explicit:

```

1 scala> Box[Int](42) //
2 val res3: Box[Int] = Box(42)

```

Om typen inte stämmer får du hjälp av kompilatorn att hitta felet:

```

1 scala> Box[String](42)
2 -- Error:
3 1 |Box[String](42)
4   |           ^^
5   |           Found:    (42 : Int)
6   |           Required: String

```

### 8.1.33 Fallgrop: Typradering (eng. *type erasure*)

Informationen om typerna i typparametrar raderas innan kodgenerering för JVM av prestandaskäl och **typparametrar saknas vid runtime** i bytekoden.

```

1 scala> def isIntVector[T](xs: Vector[T]) = xs.isInstanceOf[Vector[Int]]
2 -- Warning:
3 1 |def isIntVector[T](xs: Vector[T]) = xs.isInstanceOf[Vector[Int]]
4   |~~~~~
5   |         the type test for Vector[Int] cannot be checked at runtime
6 def isIntVector[T](xs: Vector[T]): Boolean
7
8 scala> isIntVector(Vector("hej"))
9 res42: Boolean = true // AAAARGHH!! :(

```

Måste ”packa upp” samlingen och typtestar alla element:

```

1 scala> def isIntVector[T](xs: Vector[T]) = xs.forall(_.isInstanceOf[Int])
2
3 scala> isIntVector(Vector("hej"))
4 res43: Boolean = false // FUNKAR :)

```

Typkontroll vid körtid görs oftast hellre med **match**.

---

### 8.1.34 Testning och avlusning

- Läs om testning och avlusning (eng. *debugging*) i Appendix D: ”Fixa buggar”
  - Träna på println-debugging
  - Prova debuggern i VS code
-

## 8.2 Övning matrices

### Mål

- ☐ Kunna skapa och använda matriser med nästlade strukturer av Vector.
- ☐ Kunna iterera över elementen i en matris med nästlade **for**-satser och **for-yield**-uttryck, samt nästlad applicering av map respektive foreach.
- ☐ Kunna skapa och använda funktioner som tar matriser som parametrar.
- ☐ Kunna skapa en enkel generisk klass och enkla generiska funktioner med hjälp av en typparameter.
- ☐ Kunna beskriva skillnader och likheter mellan Scala och Java vad gäller indexering och iterering i matriser implementerade med nästlade arrayer.

### Förberedelser

- ☐ Studera begreppen i kapitel 8

### 8.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** *Para ihop begrepp med beskrivning.*

Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

matris	1	A	indexerbar datastruktur i två dimensioner
radvektor	2	B	har abstrakt typparameter, typen är generell
kolumnvektor	3	C	annat ord för kolumn
kolonn	4	D	konkret typ, binds till typparameter vid kompilering
generisk	5	E	kompilatorn beräknar typ ur sammanhanget
typargument	6	F	matris av dimension $1 \times m$ med $m$ horisontella värden
typhärledning	7	G	matris av dimension $m \times 1$ med $m$ vertikala värden

**Uppgift 2.** *Skapa matriser med hjälp av nästlade samlingar.* Man kan i ett datorprogram, med hjälp av samlingar som innehåller samlingar, skapa nästlade strukturer som kan indexeras i två dimensioner och på så sätt representera en **matris**.<sup>2</sup>

a) Rita minnessituationen efter tilldelningen på rad 1 nedan. Vad har `m` för typ och värde? Vad har `m` för dimensioner? Hur sker indexeringen i ett datorprogram jämfört med i matematiken?

```

1 scala> val m = Vector((1 to 5).toVector, (3 to 7).toVector)
2 scala> m.apply(0).apply(1)
3 scala> m(1)
4 scala> m(1)(4)
```

b) Vad ger uttrycken på raderna 2, 3 och 4 ovan för värden och typ?

c) Man kan i ett datorprogram mycket väl skapa tvådimensionella, nästlade strukturer där raderna *inte* innehåller samma antal element. Det blir då ingen äkta matris i strikt

<sup>2</sup>[sv.wikipedia.org/wiki/Matris](https://sv.wikipedia.org/wiki/Matris)

matematisk mening, men man kallar ofta ändå en sådan struktur för en ”matris”. Vilken typ har variablerna `m2`, `m3`, `m4` och `m5` nedan?

```
1 scala> val m2 = Vector(Vector(1,2,3),Vector(4,5),Vector(42))
2 scala> val m3 = Vector(Vector(1,2), Vector(1.0, 2.0, 3.0))
3 scala> val m4 = m3(1) +: Vector("a") +: m3
4 scala> val m5 = Vector.fill(42){ m2(1).map(e => (e * math.random()).toInt) }
```

d) Vilken av variablerna `m2`, `m3`, `m4` och `m5` ovan representerar en äkta matris i matematisk mening? Vilken är dess dimensioner?

**Uppgift 3.** *Skapa och iterera över matriser.* Du ska skapa matriser där varje rad representerar 5 kast med en tärning i spelet Yatzy.<sup>3</sup>

a) Definiera i REPL en funktion `def throwDie: Int = ???` som returnerar ett slumptal mellan 1 och 6.

b) Skapa nedan heltalsmatris i REPL. Vilken dimension får matrisen?

```
1 scala> val ds1 = for (i <- 1 to 1000) yield
2   for (j <- 1 to 5) yield throwDie
```

c) Man kan också använda nedan varianter för att skapa en heltalsmatris. Vilken av varianterna `ds1` ... `ds6` tycker du är lättast att läsa och förstå? Prova respektive variant i REPL och ange vilken typ på `ds1` ... `ds6` som härleds av kompilatorn.

```
1 val ds2 = (1 to 1000).map(i => (1 to 5).map(j => throwDie))
2 val ds3 = (1 to 1000).map(i => Vector.fill(5)(throwDie))
3 val ds4 = for (i <- 1 to 1000) yield Vector.fill(5)(throwDie)
4 val ds5 = Vector.fill(1000)(Vector.fill(5)(throwDie))
5 val ds6 = Vector.fill(1000, 5)(throwDie)
```

d) Definiera en funktion

```
def roll(n: Int): Vector[Int] = ???
```

som ger en heltalsvektor med  $n$  stycken slumpvisa tärningskast. Kasten ska vara sorterade i växande ordning; använd för detta ändamål samlingsmetoden `sorted`.

e) Definera i REPL en funktion `isYatzy(xs: Vector[Int]): Boolean = ???` som testat om alla elementen i en heltalsvektor är samma. Använd samlingsmetoden `forall`.

f) Skapa en funktion

```
def diceMatrix(m: Int, n: Int): Vector[Vector[Int]] = ???
```

som med hjälp av funktionen `roll` skapar en matris med  $m$  st vektorer med vardera  $n$  slumpvisa tärningskast.

g) Skapa en funktion som returnerar en utskriftsvänlig sträng

```
def diceMatrixToString(xss: Vector[Vector[Int]]): String = ???
```

med hjälp av `map` och `mkString`, som fungerar enligt nedan.

```
1 scala> val dm2s = diceMatrixToString(diceMatrix(4, 5))
2 val dm2s: String = 1 4 4 6 6
3 1 1 2 6 6
4 2 4 4 5 6
5 1 1 5 6 6
6
7 scala> println(dm2s)
8 1 4 4 6 6
9 1 1 2 6 6
```

<sup>3</sup>[sv.wikipedia.org/wiki/Yatzy](http://sv.wikipedia.org/wiki/Yatzy)



```

10 2 4 4 5 6
11 1 1 5 6 6

```

h) Implementera funktionen

```
def filterYatzy(xss: Vector[Vector[Int]]): Vector[Vector[Int]]
```

som filtrerar fram alla yatzy-rader i matrisen xss enligt nedan. Använd din funktion isYatzy och samlingsmetoden filter.

```

1 scala> println(diceMatrixToString(filterYatzy(diceMatrix(10000, 5))))
2 4 4 4 4 4
3 6 6 6 6 6
4 4 4 4 4 4
5 6 6 6 6 6
6 4 4 4 4 4
7 4 4 4 4 4
8 2 2 2 2 2

```

i) Implementera funktionen

```
def yatzyPips(xss: Vector[Vector[Int]]): Vector[Int] = ???
```

som ska ge en vektor med de tärningsvärden som gav yatzy, för kasten i matrisen xss enligt nedan. Använd din funktion filterYatzy.

```

1 scala> val dm = Vector(Vector(1,2,3,4,5),Vector(4,4,4,4,4),Vector(3,3,3,3,3))
2 scala> yatzyPips(dm)
3 val res42: Vector[Int] = Vector(4, 3)

```

**Uppgift 4.** En oföränderlig, generisk matris-klass till veckans laboration [life](#). Under veckans laboration ska du simulera en enkel form av "liv" som består av celler i ett rutnät. För detta ändamål har vi nytta av en matris-klass som du ska implementera steg för steg i denna övning. Skapa case-klassen nedan med en editor i filen `Matrix.scala`. Testa din lösning med hjälp av valfri IDE, t.ex. [scalaide](#) eller [idea](#).

```

case class Matrix(data: Vector[Vector[String]]){
  def apply(row: Int, col: Int): String = data(row)(col)
}
object Matrix {
  def fill(dim: (Int, Int))(value: String): Matrix =
    Matrix(Vector.fill(dim._1, dim._2)(value))
}

```

```

scala> val m = Matrix.fill(3,4)("hej")
scala> val e = m(2, 2)

```

a) Vad får m ovan för typ?

b) Vad får e ovan för typ?

c) På hur många ställen måste du ändra i Matrix ovan för att den i stället ska representera en matris av heltal?

d) Du ska nu med hjälp av en **typparameter** göra Matrix **generisk** (eng. *generic*), så att den blir en mer användbar matrisklass som kan innehålla element av vilken typ som helst. Genomför följande ändringar i `Matrix.scala`:

- Lägg till en typparameter `T` inom klammerparenteser efter namnet `Matrix` på alla ställen där det förekommer *utom* efter namnet på kompanjonsobjektet<sup>4</sup>.
- Byt ut `String` mot `T` på alla ställen där `String` förekommer.
- Lägg till en typparameter `T` inom klammerparenteser efter **def** `fill`.

Testa din generiska klass i REPL genom att skapa en boolesk matris:

```
scala> val bm = Matrix.fill(3,4)(false)
scala> val be = bm(0, 0)
```

- e) Vad får `bm` ovan för typ?
- f) Vad får `be` ovan för typ?
- g) Lägg en kodrad i början av klasskroppen som med hjälp av `require` garanterar att alla rader i matrisen är lika långa.
- h) Lägg till en medlem **val** `dim: (Int, Int)` i klasskroppen efter `require`-satsen som ger ett par (alltså en 2-tupel) med antalet rader resp. kolumner i matrisen.
- i) Lägg till en metod **def** `updated(row: Int, col: Int)(value: T): Matrix[T]` som ger en ny matris där element på platsen (`row`, `col`) har uppdaterats till `value`.
- j) Lägg till en metod **def** `foreachIndex(f: (Int, Int) => Unit): Unit` som för varje index i data applicerar funktionen `f`.
- k) Lägg till en metod **override def** `toString` som så att en instans av `Matrix` visas enligt följande:

```
scala> val dm = Matrix.fill(3,4)(42.0)
val dm: Matrix[Double] =
Matrix of dim (3,4):
42.0 42.0 42.0 42.0
42.0 42.0 42.0 42.0
42.0 42.0 42.0 42.0
```

---

<sup>4</sup>Singelobjekt kan inte ha typparametrar, men deras medlemmar kan.

## 8.2.2 Extrauppgifter; träna mer

**Uppgift 5.** *Imperativa matrisalgoritmer.* Imperativa angreppssätt är nödvändiga att kunna när du stöter på samlingar och/eller språk som saknar funktionella metoder och/eller funktionsprogrammeringsmöjligheter. Genom att studera imperativa lösningar till de ofta mer koncisa funktionella lösningarna, får du träning i att skapa algoritmer som använder förändring genom tilldelning vid iterering.

a) Implementera `isYatzy` från uppgift 3e igen, men nu med ett imperativt angreppssätt som använder en **while**-sats i stället för funktionella `forall`. Ta hjälp av en variabel `i` som håller reda på index och en variabel `foundDiff` som håller reda på om ett avvikande värde upptäcks. Funktionen kräver ca 9 rader, så det kan vara lämpligt att öppna en editor att skriva i medan du klurar ut lösningen. Börja med att skriva pseudokod, gärna med penna på papper. Prova genom att klistra in i REPL.

b) En imperativ implementation av `diceMatrixToString` från uppgift 3g med hjälp av förändringsbara `StringBuilder`<sup>5</sup> visas nedan. Förklara hur nedan kod fungerar. Vad händer om `xss` är tom? Vad händer om `xss` bara innehåller tomma vektorer? Nämn en fördel och en nackdel med att använda **val** `sb: StringBuilder` och `append`, jämfört med en vanlig, oföränderlig **var** `s: String` och `+` för tillägg i slutet.

```
def diceMatrixToString(xss: Vector[Vector[Int]]): String =
  val sb = new StringBuilder()
  for(m <- xss.indices) do
    for(n <- xss(m).indices) do
      sb.append(xss(m)(n).toString)
      if n < xss(m).size - 1 then sb.append(" ")
      else if m < xss.size - 1 then sb.append("\n")
    end for
  end for
  sb.toString
```

c) Gör som träning en imperativ implementation av `filterYatzy` med en **for-do**-sats (alltså utan att använda `filter`, och utan att använda **yield**).

d) Förklara hur nedan funktionella implementation av `filterYatzy` med **for-yield**-uttryck fungerar. Tycker du din imperativa lösning är lättare eller svårare att läsa och förstå jämfört med nedan funktionella lösning?

```
def filterYatzy(xss: Vector[Vector[Int]]): Vector[Vector[Int]] =
  (for i <- xss.indices if isYatzy(xss(i)) yield xss(i)).toVector
```

## Uppgift 6. Strängtabell med kolumnrubriker.

a) Implementera case-klassen `Table` enligt specifikationen nedan. Du kan förutsätta att alla rader har lika många kolumner som antalet element i headings, samt att alla rubrikerna i headings är unika. Parametern `sep` anger det tecken som används för att separera kolumner. Detta förutsätts också gälla för indatafiler som läses in med `fromFile`.

*Tips:*

<sup>5</sup><https://www.scala-lang.org/api/2.12.9/scala/collection/mutable/StringBuilder.html>

- Värdet `indexOfHeading` kan skapas med hjälp av metoden `zipWithIndex` som fungerar på alla sekvenssamlingar, samt metoden `toMap` som fungerar på sekvenser av 2-tupler. Undersök först hur metoderna fungerar i REPL och sök upp deras dokumentation.
- Skapa en indatafil som du kan använda för att testa att `Table` fungerar.

```
case class Table(
  data: Vector[Vector[String]],
  headings: Vector[String],
  sep: Char
):
  /** A 2-tuple with (number of rows, number of columns) in data */
  val dim: (Int, Int) = ???

  /** The element in row r and column c of data, counting from 0 */
  def apply(r: Int, c: Int): String = ???

  /** The row-vector r in data, counting from 0 */
  def row(r: Int): Vector[String] = ???

  /** The column-vector c in data, counting from 0 */
  def col(c: Int): Vector[String] = ???

  /** A map from heading to index counting from 0 */
  lazy val indexOfHeading: Map[String, Int] = ???

  /** The column-vector with heading h in data */
  def col(h: String): Vector[String] = ???

  /** A vector with the distinct, sorted values of col with heading h */
  def values(h: String): Vector[String] = ???

  /** Headings and data with columns separated by sep */
  override lazy val toString: String = ???

object Table:
  /** Creates a new Table from fileName with columns split by sep */
  def fromFile(fileName: String, sep: Char = ';'): Table = ???
```

b) Skapa med hjälp av `Table` ett program som kan köras från terminalen med `scala run infile.csv ';'`  som ger en utskrift av antalet förekomster av olika värden i respektive kolumn (alltså en variant av registrering).

**Uppgift 7.** Skapa ett yatzy-spel för användning i terminalen. Bygg ett förenklat yatzy-spel i terminalen där användaren kan bestämma vilka tärningar som ska slås om. Börja med något riktigt enkelt och bygg sedan vidare på ditt spel genom att införa fler och fler funktioner.

### 8.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 8. Generiska funktioner.** En generisk funktion har (minst) en typparameter inom klammerparenteser efter namnet, till exempel `[T]`. Denna typ förekommer sedan som typ på (någon av) parametrarna i parameterlistan. Kompilatorn härleder en konkret typ vid kompileringstid och ersätter typparametern med denna konkreta typ. På så sätt kan en funktion fungera för många olika typer.

a) Förklara för varje rad nedan vad som händer.

```
1 scala> def tnirp[T](x: T): Unit = println(x.toString.reverse)
2 scala> tnirp(42)
3 scala> tnirp("hej")
4 scala> case class Gurka(vikt: Int)
5 scala> tnirp(Gurka(42))
6 scala> tnirp[String](42)
7 scala> tnirp[Double](42)
```

b) Man kan kombinera generiska funktioner med funktioner som tar funktioner som parametrar. Det är så `map` och `foreach` är implementerade. Förklara för varje rad nedan vad som händer.

```
1 scala> def compose[A, B, C](f: A => B, g: B => C)(x: A): C = g(f(x))
2 scala> def inc(x: Int): Int = x + 1
3 scala> def half(x: Int): Double = x / 2.0
4 scala> compose(inc, half)(42)
5 scala> compose(half, inc)(42)
```

c) Hur lyder felmeddelandet på sista raden ovan? Ändra `inc` och/eller `half` så att typerna passar.

**Uppgift 9. Generiska klasser.** Även klasser kan vara generiska. En generisk klass har (minst) en typparameter inom klammerparenteser efter klassens namn.

a) Testa nedan generiska klass `Cell[T]` i REPL. Skapa instanser av klassen `Cell[T]` där typparametern `T` binds till olika konkreta typer och förklara vad som händer.

```
1 scala> class Cell[T](var value: T):
2     override def toString = "Cell(" + value + ")"
3
4 scala> new Cell(42)
5 scala> new Cell("hej")
6 scala> new Cell(new Cell(math.Pi))
7 scala> new Cell[String](42)
8 scala> new Cell[Double](42)
```

b) Lägg till metoden **def** `concat[U](that: Cell[U]): Cell[String]` i klassen `Cell` som konkatenerar strängrepresentationerna av de båda cellvärdena.

```
1 scala> val a = new Cell("hej")
2 scala> val b = new Cell(42)
3 scala> a concat b
```

c) Vilken sorts celler kan du konkatenera om du tar bort typparameternamnet `U` i `concat` samtidigt som du använder `Cell[T]` som typ på värdeparametern `that`? Vad ger det för konsekvenser för celler av annan typ än `Cell[String]`?

**Uppgift 10.** Implementera fler generiska metoder i `Matrix[T]`. Bygg vidare på uppgift 4 och implementera nedan specifikation. Skapa egna tester som kontrollerar att alla metoder fungerar som förväntat.

**Specification** `Matrix[T]`

```
/** En oföränderlig, generisk Matris-klass. */
case class Matrix[T](data: Vector[Vector[T]]):
  require(???) // garantera att alla rader har lika många kolumner

  /** Ger ett par med antal rader och kolumner. */
  val dim: (Int, Int) = ???

  /** Ger elementet på plats (row, col). */
  def apply(row: Int, col: Int): T = ???

  /** Ger en ny matris där elementet på plats (row, col) har värdet value. */
  def updated(row: Int, col: Int)(value: T): Matrix[T] = ???

  /** Applicerar f på alla element. */
  def foreach(f: T => Unit): Unit = ???

  /** Applicerar f på alla index. */
  def foreachIndex(f: (Int, Int) => Unit): Unit = ???

  /** Ger en ny matris med resultaten av elementvis applicering av f. */
  def map[U](f: T => U): Matrix[U] = ???

  /** Ger en ny matris med resultaten av applicering av f på varje index. */
  def mapIndex[U](f: (Int, Int) => U): Matrix[U] = ???

  /** Ger en utskriftsvänlig strängrepresentation av matrisen. */
  override def toString = ???

object Matrix:
  /** Ger en matris med dimension dim där alla element har värdet value. */
  def fill[T](dim: (Int, Int))(value: T): Matrix[T] = ???
```

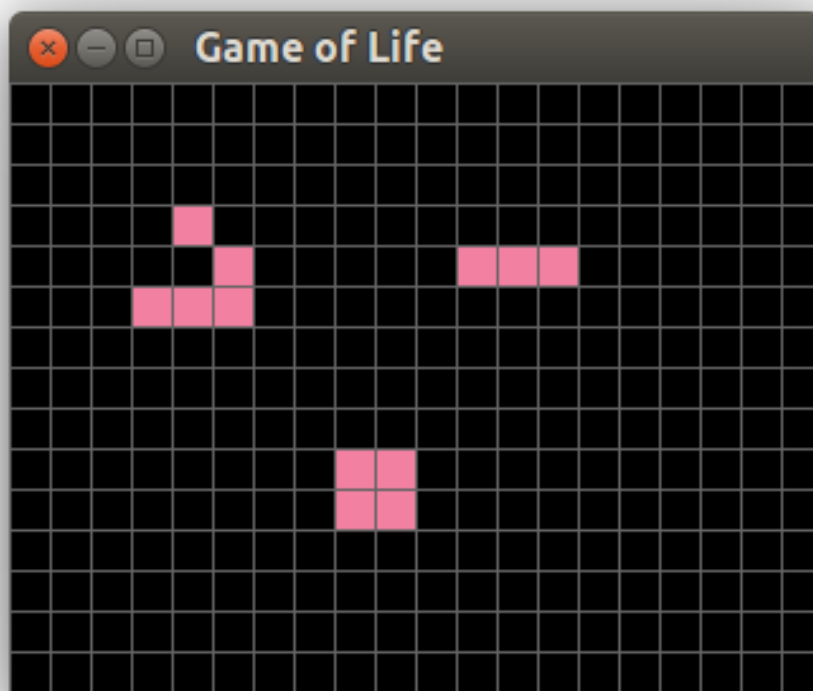
## 8.3 Laboration: life

### Mål

- ☐ Kunna skapa och använda matriser med hjälp av en generisk datatyp.
- ☐ Kunna iterera över alla element i en matris.
- ☐ Träna på algoritmkonstruktion.
- ☐ Träna på hantering av både oföränderliga och förändringsbara objekt.
- ☐ Prova på att använda en avlusare (eng. *debugger*) i en integrerad utvecklingsmiljö (IDE), t.ex. VS code.

### Förberedelser

- ☐ Gör övning matrices i kapitel 8, speciellt uppgift 4.
- ☐ Läs igenom hela laborationen och studera den givna koden<sup>6</sup>.
- ☐ Läs appendix D om avlusning (eng. *debugging*).
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).



**Figur 8.1:** Ett binärt, mörkt datauniversum av dimension  $15 \times 20$ . Cellkolonin innehåller tre cellgrupper: ett rymdskepp av typen *glider*, en *blinker* och ett *block*.

### 8.3.1 Bakgrund

*Game of Life* simulerar en koloni av encelliga organismer som lever, förökar sig och dör i en matris, enligt några enkla men väl valda regler som konstruerades av matematikern John Horton Conway på 1970-talet. Spelet går ut på att simulera flera generationer utifrån en startkonfiguration, även kallad *cellkoloni*, där varje enskild cells överlevnad beror på dess

<sup>6</sup>[https://github.com/lunduniversity/introprog/tree/master/workspace/w08\\_life](https://github.com/lunduniversity/introprog/tree/master/workspace/w08_life)

omgivning. Spelet har inga medvetna spelare och om reglerna följs så kommer slutresultatet enbart bero på startkonfigurationen.

I *Game of Life* består universum av en matris med celler som är antingen levande eller döda. Varje cell har 8 stycken *grannar*, som utgörs av de närmsta omgivande cellerna vertikalt, horisontellt och diagonalt. Varje cells tillstånd i nästa generation bestäms av följande regler:

1. **Fortlevnad.** Om en levande cell har två eller tre grannar så lever den vidare.
2. **Död.** Om en levande cell har färre än två eller mer än tre grannar så dör den av underpopulation respektive överpopulation.
3. **Födelse.** Om cellen är död och har exakt tre grannar så föds den och dess tillstånd ändras till levande, annars fortsätter den vara död.

Flera cellkolonier uppvisar ett "levande" beteende där cellmatrisen koloniserar på intressanta vis när en sekvens av generationer visualiseras. Detta är ett exempel på *emergent* beteende där komplexa, självorganiserade strukturer kan uppstå ur enkla förutsättningar.

Läs mer om *Game of Life* på Wikipedia:

- [https://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway's_Game_of_Life)
- [https://sv.wikipedia.org/wiki/Game\\_of\\_Life](https://sv.wikipedia.org/wiki/Game_of_Life)

### 8.3.2 Obligatoriska krav

Följande funktionella krav ska uppfyllas av ditt program:

- ☐ Levande celler ska ha den vackra rosa<sup>7</sup> RGB-färgen (242, 128, 161).
- ☐ Döda celler ska vara svarta som rymden.
- ☐ Detta mörka universum med binära dataceller ska ritas i ett rutnät bestående av smala, stilfulla linjer, så som visas i fig. 8.1.
- ☐ Tangenttryckningar och musklick ska fungera enligt följande hjälptext, som ska skrivas ut då programmet startas:

```
val help = """
Welcome to GAME OF LIFE!

Click on cell to toggle.
Press ENTER for next generation.
Press SPACE to toggle play/stop.
Press R to create random life.
Press BACKSPACE to clear life.
Close window to exit.
"""
```

Då *play* aktiveras med blankstegstangenten ska en kontinuerlig simulering av universum fortgå där varje ny generation visualiseras med en lagom fördröjning emellan generationer, tills simuleringen stoppas, t.ex. genom tryck ånyo på blankstegstangenten. Vid varje *Enter*-tryck visas *en* efterkommande generation och ev. pågående simulering stoppas. Vid musklick på en cell ska livstillståndet växlas från levande till död eller vice versa. Ett tryck på R ska ge slumpmässigt liv. Ett tryck på backstegstangenten ska rendera alla universums cellers död.

Din kod ska utformas enligt dessa design-krav:

- ☐ Alla klasser och singelobjekt ska ligga i paketet `life`.
- ☐ Det ska finnas en oföränderlig case-klass `Life` som representerar ett celluniversum med hjälp av en `Matrix[Boolean]` från uppgift 4 i veckans övning.

<sup>7</sup><https://www.dsek.se/aktiva/grafiskprofil/farg.php>



- ☐ Det ska finnas en klass `LifeWindow` som visualiserar en instans av klassen `Life` i ett introprog.`PixelWindow` så som i fig. 8.1.

### 8.3.3 Valbara krav – välj minst ett

Du ska implementera minst ett (gärna flera) av dessa krav:

- ☐ Cellerna ska färgläggas i olika färger i enlighet med reglerna för nästa generation. Fortlevnad ska fortfarande vara vackert rosa och fortvarig död svart. Följande färger föreslås men välj andra om du tycker det blir finare:

```
val UnderPopulated = java.awt.Color.cyan // en giftig färg
val OverPopulated   = java.awt.Color.red  // rödklämd av trängsel
val WillBeBorn      = new java.awt.Color(40, 0, 0) // snart levande
```

Ge dessutom `LifeWindow` en klassparameter `isMultiColor` som gör det möjligt att välja om det ska bli mångfärgade celler eller om det bara ska finnas rosa och svart som i grundkraven.

- ☐ Om man trycker på S för *Save* ska `introprog.Dialog.file("Save Life")` visas och, om användaren inte trycker **Cancel**, det aktuella livet sparas med hjälp av `introprog.IO.save` i en textfil via metoden `toString` i `Life`.
- ☐ Om man trycker på O för *Open* ska `introprog.Dialog.file("Open Life")` anropas och ett nytt universum läsas in från textfil enligt lämpligt format. Inläsningen ska ske med hjälp av `introprog.IO.loadString` och tolkas till en `Life`-instans av en metod i kompanjonsobjektet med detta huvud:

```
def fromString(s: String, rowDelim: String="\n", alive: Char='0'): Life
```

Testa med filen `glider-gun.txt` som ska ha följande innehåll på de första 11 raderna och totalt 32 rader där alla rader efter elfte raden innehåller tomt liv:

```
> head -11 glider-gun.txt
-----
-----0-----
-----0-0-----
-----00-----00-----00-----
-----0--0--00-----00-----
-00-----0--0--00-----
-00-----0--0-00--0-0-----
-----0--0--0-----
-----0--0-----
-----00-----
-----
```

- ☐ Universum ska vara cirkulärt, d.v.s grannen vid kanten finns på andra sidan genom att indexeringen börjar om (eng. *wrapped*) enligt modulo-räkning. Inför en klassparameter `isWrapped` i `Life` och en variabel `wrapped: Boolean` i kompanjonsobjektet `Life` som styr om fabriksmetoderna skapar ett universum som är cirkulärt eller ej, så att du lätt kan konfigurera detta. *Tips:* Du har stor nytta av att använda `java.lang.Math.floorMod` i `apply`-metoden i `Life`; metoden `floorMod` räknar på lämpligt sätt med negativa värden, se dokumentationen för `Math`-paketet i JDK8.
- ☐ Läs om varianter till *Game of Life* på Wikipedia och implementera alternativa regler som görs valbara genom konfigureringsparameter i `main`.
- ☐ Skapa en klass `LifeStatistics` som genom väldigt många simuleringar ska ta reda på sannolikheten att en slumpmässig cellkoloni efter  $n$  generationer fortfarande utvecklas, respektive är helt dött. Ingen visualisering med `PixelWindow` ska ske; endast antalet

celler som lever vid generation  $n$  och antalet celler som ändrades sedan generation  $n - 1$  behöver registreras.

### 8.3.4 Tips och förslag

1. Här är ett förslag på hur du kan utforma klassen `Life`:

```
package life

case class Life(cells: Matrix[Boolean]):

  /** Ger true om cellen på plats (row, col) är vid liv annars false.
   * Ger false om indexeringen är utanför universums gränser.
   */
  def apply(row: Int, col: Int): Boolean = ???

  /** Sätter status på cellen på plats (row, col) till value. */
  def updated(row: Int, col: Int, value: Boolean): Life = ???

  /** Växlar status på cellen på plats (row, col). */
  def toggled(row: Int, col: Int): Life = ???

  /** Räknar antalet levande grannar till cellen i (row, col). */
  def nbrOfNeighbours(row: Int, col: Int): Int = ???

  /** Skapar en ny Life-instans med nästa generation av universum.
   * Detta sker genom att applicera funktionen \code{rule} på cellerna.
   */
  def evolved(rule: (Int, Int, Life) => Boolean = Life.defaultRule): Life =
    var nextGeneration = Life.empty(cells.dim)
    cells.foreachIndex( (r,c) =>
      ???
    )
    nextGeneration

  /** Radseparerad text där 0 är levande cell och - är död cell. */
  override def toString = ???

object Life:
  /** Skapar ett universum med döda celler. */
  def empty(dim: (Int, Int)): Life = ???

  /** Skapar ett univversum med slumpmässigt liv. */
  def random(dim: (Int, Int)): Life = ???

  /** Implementerar reglerna enligt Conways Game of Life. */
  def defaultRule(row: Int, col: Int, current: Life): Boolean = ???
```

Du har nytta av metoden `nbrOfNeighbours` när du ska implementera `defaultRule`. Vid implementation av `random` är metoden `foreachIndex` i `Matrix[T]` smidig att använda. Om du som i förslaget ovan låter `evolved` ta uppdateringsregeln som en funktionsparameter blir det lättare att konfigurera vilka regler som ska gälla och därmed blir det även lättare att skapa varianter av *Game of Life* genom att införa nya regler i kompanjonsobjektet (se en av de valfria uppgifterna med vidare hänvisning till Wikipedia).

2. Här är ett förslag på hur du kan utforma klassen `LifeWindow`:

```
package life

import introprog.PixelWindow
import introprog.PixelWindow.Event
```

```

object LifeWindow:
  val EventMaxWait = 1 // milliseconds
  var NextGenerationDelay = 200 // milliseconds
  // lägg till fler användbara konstanter här tex färger etc.

class LifeWindow(rows: Int, cols: Int):
  import LifeWindow.* // importera namn från kompanjon

  var life = Life.empty(rows, cols)
  val window: PixelWindow = ???
  var quit = false
  var play = false

  def drawGrid(): Unit = ???

  def drawCell(row: Int, col: Int): Unit = ???

  def update(newLife: Life): Unit =
    val oldLife = life
    life = newLife
    life.cells.foreachIndex{ ??? }

  def handleKey(key: String): Unit = ???

  def handleClick(pos: (Int, Int)): Unit = ???

  def loopUntilQuit(): Unit = while !quit do
    val t0 = System.currentTimeMillis
    if play then update(life.evolved())
    window.awaitEvent(EventMaxWait)
    while window.lastEventType != PixelWindow.Event.Undefined do
      window.lastEventType match
        case Event.KeyPressed => handleKey(window.lastKey)
        case Event.MousePressed => handleClick(window.lastMousePos)
        case Event.WindowClosed => quit = true
        case _ =>
      window.awaitEvent(EventMaxWait)
    val elapsed = System.currentTimeMillis - t0
    Thread.sleep((NextGenerationDelay - elapsed) max 0)

  def start(): Unit = { drawGrid(); loopUntilQuit() }

```

3. **Dra nytta av din IDE.** Det finns många användbara finesser i en integrerad utvecklingsmiljö (eng. *Integrated Development Environment (IDE)*), så som Microsoft VS Code<sup>8</sup> med tillägget Metals<sup>9</sup> eller JetBrains IntelliJ IDEA<sup>10</sup> med Scala-plugin<sup>11</sup>. Läs på nätet om din IDE och lär dig om sådant du inte kände till som verkar användbart. Sök speciellt upp listan med kortkommandon<sup>12 13</sup> och lär dig några valfria kortkommandon som kan hjälpa dig att snabba upp sådant du gör ofta.

4. Studera dokumentationen om avlusaren (debuggern) in din IDE. <sup>14 15 16</sup>

<sup>8</sup><https://code.visualstudio.com/>

<sup>9</sup><https://scalameta.org/metals/docs/editors/vscode>

<sup>10</sup><https://www.jetbrains.com/idea/>

<sup>11</sup><https://www.jetbrains.com/help/idea/discover-intellij-idea-for-scala.html>

<sup>12</sup><https://code.visualstudio.com/docs/getstarted/keybindings>

<sup>13</sup><https://www.jetbrains.com/idea/resources/>

<sup>14</sup><https://scalameta.org/metals/docs/editors/vscode#running-and-debugging-your-code>

<sup>15</sup><https://code.visualstudio.com/docs/editor/debugging>

<sup>16</sup><https://www.jetbrains.com/help/idea/debugging-code.html>

## Kapitel 9

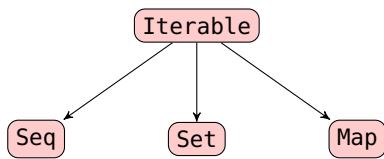
# Mängder och tabeller

Begrepp som ingår i denna veckas studier:

- ☐ innehållstest
- ☐ mängd
- ☐ Set
- ☐ mutable.Set
- ☐ nyckel-värde-tabell
- ☐ Map
- ☐ mutable.Map
- ☐ hash code
- ☐ java.util.HashMap
- ☐ java.util.HashSet
- ☐ persistens
- ☐ serialisering
- ☐ textfiler
- ☐ Source.fromFile
- ☐ java.nio.file

## 9.1 Teori

### 9.1.1 Hierarki av samlingstyper i `scala.collection v2.13`



Iterable har metoder som är implementerade med hjälp av:

```
def foreach[U](f: Elem => U): Unit
def iterator: Iterator[A]
```

Seq: ordnade i sekvens

Set: unika element

Map: par av (nyckel, värde)

Samlingen **Vector** är en Seq som är en Iterable.  
De konkreta samlingarna är uppdelade i dessa paket:

`scala.collection.immutable`

`scala.collection.mutable`

(undantag: primitiva förändringsbara `scala.Array` är automatiskt synlig)

där flera är **automatiskt** importerade  
som **måste importeras** explicit

### 9.1.2 Metoden `iterator` ger en "engångs-iterator"

Med `iterator` kan man iterera med **while**, men endast **en gång**; sedan är iteratorn "förbrukad". (Men man kan be om en ny.) Används "under huven" i samlingsbiblioteket för att implementera andra metoder.

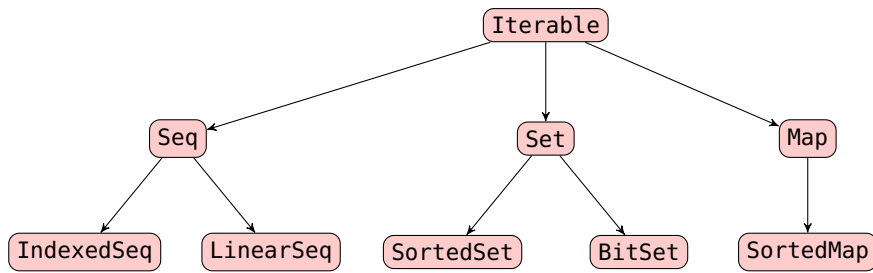
```

1 scala> val xs = Vector(1,2,3,4)
2 val xs: Vector[Int] = Vector(1, 2, 3, 4)
3
4 scala> val it = xs.iterator
5 val it: Iterator[Int] = <iterator>
6
7 scala> while it.hasNext do print(it.next)
8 1234
9
10 scala> it.hasNext
11 val res0: Boolean = false
12
13 scala> it.next
14 java.util.NoSuchElementException: next on empty iterator
  
```

**Normalt** behöver man **inte** använda `iterator`: det finns oftast färdiga metoder som gör det man vill, till exempel `foreach`, `map`, `sum`, `min` etc.

### 9.1.3 Mer specifika samlingstyper i `scala.collection`

Det finns **mer specifika subtyper** av Seq, Set och Map:



**Vector** är en **IndexedSeq** medan **List** är en **LinearSeq**.

[docs.scala-lang.org/overviews/collections-2.13/overview.html](https://docs.scala-lang.org/overviews/collections-2.13/overview.html)

---

#### 9.1.4 Några oföränderliga och förändringsbara sekvenssamlingar

`scala.collection.immutable.Seq.`

IndexedSeq.

**Vector**

**Range**

LinearSeq.

**List**

**Queue**

`scala.collection.mutable.Seq.`

IndexedSeq.

**ArrayBuffer**

**StringBuilder**

LinearSeq.

**ListBuffer**

**Queue**

Fördjupning: Studera samlingars prestanda-egenskaper här:

[docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html](https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html)

---

### 9.1.5 Några användbara metoder på samlingar

---

<b>Iterable</b>	<code>xs.size</code>	antal elementet
	<code>xs.head</code>	första elementet
	<code>xs.last</code>	sista elementet
	<code>xs.take(n)</code>	ny samling med de första n elementet
	<code>xs.drop(n)</code>	ny samling utan de första n elementet
	<code>xs.foreach(f)</code>	gör f på alla element, returtyp Unit
	<code>xs.map(f)</code>	gör f på alla element, ger ny samling
	<code>xs.filter(p)</code>	ny samling med bara de element där p är sant
	<code>xs.groupBy(f)</code>	ger en Map som grupperar värdena enligt f
	<code>xs.mkString(",")</code>	en kommaseparerad sträng med alla element
	<code>xs.zip(ys)</code>	ny samling med par (x, y); "zippa ihop" xs och ys
	<code>xs.zipWithIndex</code>	ger en Map med par (x, index för x)
	<code>xs.sliding(n)</code>	ny samling av samlingar genom glidande "fönster"

---

Prova

<b>Seq</b>	<code>xs.length</code>	samma som <code>xs.size</code>
	<code>xs :+ x</code>	ny samling med x sist efter xs
	<code>x ++ xs</code>	ny samling med x före xs

fler samlingsmetoder ur snabbreferensen: <http://cs.lth.se/quickref>

**Minnesregel** för `++` och `:+` **Colon on the collection side**  
 Digga denna: [https://youtu.be/Lm9JWlEMHjo?si=sNdn\\_ZDa0RlGr3lt](https://youtu.be/Lm9JWlEMHjo?si=sNdn_ZDa0RlGr3lt)

### 9.1.6 Repetition: Vad är en sekvens?

- En sekvens är en **följd av element** som
  - är **numrerade** (t.ex. från noll), och
  - är av en viss **typ** (t.ex. heltal).
- En sekvens kan innehålla **dubbletter**.
- En sekvens kan vara **tom** och ha längden noll.
- Exempel på en icke-tom sekvens med dubletter:

```
scala> val xs = Vector(42, 0, 42, -9, 0, 13, 7)
val xs: Vector[Int] = Vector(42, 0, 42, -9, 0, 13, 7)
```

- **Indexering** ger ett element via dess ordningsnummer:

```
1 scala> xs(2)
2 val res0: Int = 42
3
4 scala> xs.apply(2)
5 val res1: Int = 42
```

### 9.1.7 En sträng är också en IndexedSeq[Char]

Det sker vid behov **implicit konvertering** från String till IndexedSeq[Char].

```
scala> val x: IndexedSeq[Char] = "hej"
val x: IndexedSeq[Char] = hej
```

Detta gör att **alla samlingsmetoder på Seq även funkar på strängar** och även flera andra smidiga strängmetoder erbjuds **utöver** de som finns i `java.lang.String` genom klassen `StringOps`.

```
scala> "hej". //tryck på TAB och se alla strängmetoder
JLine: do you wish to see all 248 possibilities (42 lines)?
```

Detta är en stor fördel med Scala jämfört med många andra språk, som har strängar som inte kan allt som andra sekvenssamlingar kan.

### 9.1.8 Konvertera mellan olika samlingstyper

- För vanligt förekommande konverteringar finns metoderna `toVector`, `toList`, `toArray`, `toBuffer`, `toMap`, `toSeq`, `toIndexedSeq`, `toSet`, `toString`
- Metoden `to` (ny från Scala 2.13) tar ett **kompanjonsobjekt** ur samlingsbiblioteket som argument och kan användas för konvertering till godtycklig samlingstyp.
- Detta kräver kopiering om underliggande representation är olika och samlingen är förändringsbar.
- Kan användas för att t.ex. konvertera mellan oföränderlig och förändringsbar samling:

```
scala> val ms = Set(1,2,3).to(collection.mutable.Set)
val ms: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3)
```

### 9.1.9 Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```



- En mängd är **inte** en sekvens: du kan inte utgå från att elementen ligger i någon viss ordning, t.ex. den ordning som de ges vid konstruktion; en mängd har ej längd, men en **storlek**; metoden `size` ger antalet element men metoden `length` saknas.
- En mängd kan vara **tom** och har då storleken 0.
- Man kan gå igenom element i **någon** ordning (exakt vilken är ej def.), med till exempel `xs.map(f)` eller `for (x <- xs) yield f(x)`
- Det går **inte** att indexera i en mängd med `apply`, som i stället ger **innehållstest**: `Set(1,2,3).apply(3) == true`
- En mängd `Set[T]` med element av typen `T` kan således ses som ett **predikat för innehållstest**: alltså en funktion `T => Boolean` som är **true** om elementet finns annars **false**

### 9.1.10 Exempel: Oföränderlig mängd

- **Skapa:**

```
scala> var xs = Set("gurka", "tomat", "banan", "pingvin")
```

- **Läsa:** avgöra medlemskap

```
scala> xs("gurka")
val res1: Boolean = true
```

- **Uppdatera:** lägg till element (händer inget om redan finns)

```
scala> xs = xs + "jordekorre"
```

- **Ta bort:** (om finns, annars händer inget)

```
scala> xs = xs - "gurka"
```

SLUT = Skapa, Läsa, Uppdatera, Ta bort

CRUD = Create, Read, Update, Delete

### 9.1.11 Mysteriet med de försvunna elementen

Vad händer här?

```
scala> val xs1 = Vector(1,2,3,4,5,6)
scala> xs1.map(_ % 2).count(_ == 0)
val res0: Int = 3           // antalet jämna tal
scala> val xs2 = Set(1,2,3,4,5,6)
scala> xs2.map(_ % 2).count(_ == 0)
val res1: Int = 1           // varför?
```

Mängdegenskaper ger att `xs2.map(_ % 2) == Set(0, 1)`

Fundera alltid noga på om du **riskerar att förlora duplikat** som du egentligen hade velat behålla!

Använd `toSeq` på mängd om du behöver sekvensegenskaper:

```
scala> xs2.toSeq.map(_ % 2).count(_ == 0)
val res1: Int = 3           // med toSeq blir det som vi ville
```

### 9.1.12 Exempel: Förändringsbar mängd

Med en **förändringsbar** mängd kan man stegvis utöka på plats.

```
1 scala> val mängd = scala.collection.mutable.Set.empty[Int]
2
3 scala> for i <- 1 to 1_000_000 do mängd += i
4
5 scala> mängd.contains(-1) // samma som mängd(-1) eller mängd.apply(-1)
```

En **mängd** är **snabb** på att avgöra om ett element **finns eller inte** i mängden. Ingen linjärsökning krävs eftersom den smarta implementationen av datastrukturen medger snabb uppslagning (eng. *lookup*) av ett element.

Men i en sekvens krävs linjärsökning vid innehållstest:

```
1 scala> val sekvens = (1 to 1_000_000).toVector
2
3 scala> sekvens.contains(-1) // kräver linjärsökning ända till slutet
```

Övning: Testa själv att mäta tidsskillnaden med hjälp av:

```
def nanos(b: => Unit) = { val t0 = System.nanoTime; b; System.nanoTime - t0 }
```

### 9.1.13 Vad är en nyckel-värde-tabell?

- En **nyckel-värde-tabell** är en samling element som är **par** med: en **nyckel** av någon typ K och ett **värde** av någon typ V.
- En sådan tabell kan skapas ur en sekvens av par (k, v) där k är en nyckel och v är ett värde:

```
1 scala> val ålder = Map("Björn" -> 42, "Sandra" -> 35, "Kim" -> 19)
2 val ålder: Map[String, Int] = Map(Björn -> 42, Sandra -> 35, Kim -> 19)
```

- Tabellens nycklar utgör en mängd som ges av metoden `keySet`; nycklarna är **unika**.
- Elementen utgör **inte en sekvens** och har ingen speciell ordning; en nyckel-värde-tabell har ej längd, men en **storlek**; metoden `size` ger antalet element.
- En tabell kan ses som en uppslagsfunktion (eng. *dictionary*): alltså en funktion `K => V` som ger ett värde givet en nyckel.

### 9.1.14 Den fantastiska nyckel-värde-tabellen Map

- En **nyckel-värde-tabell** (eng. *key-value table*) är en slags generaliserad vektor där man kan "indexera" med godtycklig typ.
- Kallas även **hashtabell** (eng. *hash table*), **lexikon** (eng. *Dictionary*) eller **mapp** (eng. *Map*) (det blir lätt sammanblandning med metoden `map`).
- Om man vet nyckeln kan man slå upp värdet **snabbt**, på liknande sätt som indexering sker snabbt i en vektor givet heltalsindex.

- Denna datastruktur är **mycket användbar** och fungerar som en slags databas i kombination med filtrering, registrering, etc.

### 9.1.15 Exempel: Oföränderlig nyckel-värde-tabell

- **Skapa:** ge par till metoden apply

```
scala> var födelse = Map("C" -> 1972, "C++" -> 1983, "C#" -> 2000,
  "Scala" -> 2004, "Java" -> 1995, "Javascript" -> 1995, "Python" -> 1991)
```

- **Läsa:** slå upp ett värde med hjälp av en nyckel

```
scala> val year = födelse.apply("Scala")
val year: Int = 2004
```

- **Uppdatera:** lägga till ett par, ersätta ett par

```
scala> födelse = födelse + ("Kotlin" -> 2011)
födelse: Map[String, Int] = HashMap(Scala -> 2004, C# -> 2000, Python -> 1991,
  Javascript -> 1995, C -> 1972, C++ -> 1983, Kotlin -> 2011, Java -> 1995)
```

- **Ta bort** ett par via nyckeln (om finns, annars händer inget)

```
scala> födelse = födelse - "Python"
födelse: Map[String, Int] = HashMap(Scala -> 2004, C# -> 2000,
  Javascript -> 1995, C -> 1972, C++ -> 1983, Kotlin -> 2011, Java -> 1995)
```

### 9.1.16 Fler exempel nyckel-värde-tabell

Några ofta förekommande metoder på tabeller:

- `xs.keySet` ger en mängd av alla nycklar
- `xs.map(f)` kör funktionen `f` på alla par (key, value) i **någon** ordning
- `xs.map((k, v) => k -> f(v))` kör funktionen `f` på alla **värden**

```
scala> val färg = Map("gurka" -> "grön", "tomat" -> "röd", "aubergine" -> "lila")
val färg: Map[String, String] =
  Map(gurka -> grön, tomat -> röd, aubergine -> lila)

scala> färg("gurka")
val res0: String = grön

scala> färg.keySet
val res1: Set[String] = Set(gurka, tomat, aubergine)

scala> val ärGrönSak = färg.map((k,v) => (k, v == "grön"))
val ärGrönSak: Map[String, Boolean] =
  Map(gurka -> true, tomat -> false, aubergine -> false)

scala> val baklängesFärg = färg.map((k, v) => k -> v.reverse)
val baklängesFärg: Map[String, String] =
  Map(gurka -> nörg, tomat -> dör, aubergine -> alil)
```

### 9.1.17 Från sekvens av par till tabell

```

1 scala> val xs = Vector(("Kim",42), ("Pam", 42), ("Kim", 50), ("Pam", 50))
2 val xs: Vector[(String, Int)] =
3   Vector((Kim,42), (Pam,42), (Kim,50), (Pam,50))
4
5 scala> xs.toMap
6 val res0: Map[String, Int] =
7   Map(Kim -> 50, Pam -> 50) // inga dublettnycklar
8
9 scala> val grupperaEfterNamn = xs.groupBy(_._1)
10 grupperaEfterNamn: Map[String,Vector[(String, Int)]] =
11   Map(Kim -> Vector((Kim,42), (Kim,50)), Pam -> Vector((Pam,42), (Pam,50)))
12
13 scala> val grupperaEfterÅlder = xs.groupBy(_._2)
14 grupperaEfterÅlder: Map[Int,Vector[(String, Int)]] =
15   Map(50 -> Vector((Kim,50), (Pam,50)), 42 -> Vector((Kim,42), (Pam,42)))

```

### 9.1.18 Övning: Implementera en Multimap

- Om du lägger till ett värde i en *vanlig* Map så ersätts värdet:

```

scala> val m = Map(1 -> 2, 1 -> 3, 2 -> 1, 2 -> 2)
val m: Map[Int, Int] = Map(1 -> 3, 2 -> 2) //senaste värdet gäller

```

...men ibland vill vi i stället lagra alla tillagda värden.

- En **multimap** är en speciell nyckel-värde-tabell där värdena utgör en samling (ofta en mängd).
- En multimap samlar alla värden som har samma nyckel.

```

1 scala> val mm = Multimap(1 -> 2, 1 -> 3, 2 -> 1, 2 -> 2)
2 val mm: Multimap[Int, Int] = Multimap(1 -> Set(2, 3), 2 -> Set(1, 2))

```

Övning: Implementera en multimap som fungerar som ovan, med hjälp av en case-klass med attributet toMap som är en oföränderlig nyckel-värde-tabell där värdena är en mängd.

Tips: Använd groupBy

### 9.1.19 Lösning: Multimap

```

case class Multimap[K, V] private (toMap: Map[K,Set[V]]):
  def apply(k: K): Set[V] = toMap(k)

  def +(kv: (K, V)): Multimap[K, V] = kv match
    case (k, v) if toMap.isDefinedAt(k) => Multimap(toMap.updated(k, toMap(k) + v))
    case (k, v) => Multimap(toMap + (k -> Set(v)))

  override def toString = toMap.mkString("Multimap(", ", ", ", ")")

object Multimap:
  def apply[K, V](kvs: (K,V)*): Multimap[K, V] =
    new Multimap(kvs.groupBy(_._1).map((k,xs) => k -> xs.map(_._2).toSet))

```

### 9.1.20 Speciella metoder på förändringsbara samlingar

Både Set och Map finns i **förändringsbara** varianter med extra metoder för uppdatering av innehållet "på plats" utan att nya samlingar skapas.

```
scala> import scala.collection.mutable

scala> val ms = mutable.Set.empty[Int]
val ms: scala.collection.mutable.Set[Int] = HashSet()

scala> ms += 42
val res0: scala.collection.mutable.Set[Int] = HashSet(42)

scala> ms ++= Seq(1, 2, 3, 1, 2, 3) // metoden ++= gör samma som addAll
val res1: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 42)

scala> val ms2 = ms diff Set(1, 2)

scala> ms2.mkString("Mängd: ", ", ", " Antal: " + ms2.size)
val res2: String =Mängd: 3, 42 Antal: 2

scala> val ordpar = mutable.Map.empty[String, String]

scala> ordpar ++= Seq("hej" -> "svejs", "abra" -> "kadabra", "ada" -> "lovelace")

scala> ordpar("abra")
val res3: String = kadabra
```

### 9.1.21 Övning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling:  
Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. toSet på en mängd, eller toMap på en hashtabell, eller toVector på en ArrayBuffer eller Array.

Exempel där lösningen har nytta av lokal förändring på plats:

```
def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) = ???
```

```
1 scala> kastaTärningTillsAllaUtfallUtomEtt()
2 val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))
```

### 9.1.22 Övning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling:  
Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. toSet på en mängd, eller toMap på en hashtabell, eller toVector på en ArrayBuffer eller Array.

```
def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) =
  /*
    låt s vara en tom förändringsbar heltalsmängd
    låt n vara noll
```

```

    så länge mängden s är mindre än sidor - 1 gör:
    lägg till ett nytt tärningskast i s
    uppdatera n så att vi räknar hur många slumpstal som dragits
  */
  (n, s.toSet) // notera toSet som ger oföränderlig mängd

```

```

1 scala> kastaTärningTillsAllaUtfallUtomEtt()
2 val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))

```

### 9.1.23 Lösning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling: Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. `toSet` på en mängd, eller `toMap` på en hashtabell, eller `toVector` på en `ArrayBuffer` eller `Array`.

```

def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) =
  val s = scala.collection.mutable.Set.empty[Int] //förändringsbar lokalt
  var n = 0
  while s.size < sidor - 1 do
    s += util.Random.nextInt(sidor) + 1
    n += 1
  (n, s.toSet) // notera toSet som ger oföränderlig mängd

```

```

1 scala> kastaTärningTillsAllaUtfallUtomEtt()
2 val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))

```

I veckans uppgifter används detta i en s.k. **builder**: Först bygga upp en förändringsbar struktur i `FreqMapBuilder` steg för steg, och sedan, då alla tillägg är gjorda, övergå till oföränderlig struktur `Map[String, Int]`.

### 9.1.24 Metoden `sliding`

Metoden `sliding(n)` skapar med ett "glidande fönster" en sekvens av delsekvenser av längd `n` genom att "svepa fönstret" från början till slut:

```

1 scala> val xs = "fem myror är fler än fyra elefanter".split(' ').toVector
2 val xs: Vector[String] = Vector(fem, myror, är, fler, än, fyra, elefanter)
3
4 scala> xs.sliding(2).toVector
5 val res0: Vector[Vector[String]] =
6   Vector(Vector(fem, myror), Vector(myror, är), Vector(är, fler),
7     Vector(fler, än), Vector(än, fyra), Vector(fyra, elefanter))
8
9 scala> xs.sliding(3).toVector
10 val res1: Vector[Vector[String]] =
11   Vector(Vector(fem, myror, är), Vector(myror, är, fler),
12     Vector(är, fler, än), Vector(fler, än, fyra),
13     Vector(än, fyra, elefanter))

```

Denna metod har du nytta av på veckans laboration!  
(se fler exempel på övning)

---

### 9.1.25 Metoderna zipWithIndex, groupBy

```
1 scala> val kort = Vector("Knekt", "Dam", "Kung", "Äss")
2
3 scala> val kortIndex = kort.zipWithIndex.toMap
4 kortIndex: Map[String,Int] = Map(Knekt -> 0, Dam -> 1, Kung -> 2, Äss -> 3)
5
6 scala> kortIndex("Kung") > kortIndex("Knekt")
7 res0: Boolean = true
8
9 scala> kortIndex.map(p => p._1 -> (p._2 + 11))
10
11 scala> val tärningskast = Vector(1,2,3,4,5,6,2,4,6)
12
13 scala> val grupperaStörreÄnFyra = tärningskast.groupBy(_ > 4)
14 grupperaStörreÄnFyra: Map[Boolean,Vector[Int]] =
15   Map(false -> Vector(1, 2, 3, 4, 2, 4), true -> Vector(5, 6, 6))
16
17 scala> val grupperaLika = tärningskast.groupBy(x => x)
18 grupperaLika: Map[Int,Vector[Int]] = Map(5 -> Vector(5), 1 -> Vector(1),
19   6 -> Vector(6, 6), 2 -> Vector(2, 2), 3 -> Vector(3), 4 -> Vector(4, 4))
20
21 scala> val frekvens = tärningskast.groupBy(x => x).map((k,v) => k -> v.size)
22 frekvens: Map[Int,Int] = Map(5 -> 1, 1 -> 1, 6 -> 2, 2 -> 2, 3 -> 1, 4 -> 2)
```

### 9.1.26 Fler användbara samlingsmetoder

Exempel att öva på: räkna bokstäver i ord.  
Undersök vad som händer i REPL:

```
val ord = "sex laxar i en laxask sju sjösjuka sjömän"
val uppdelad = ord.split(' ').toVector
val ordlängd = uppdelad.map(_.length)
val ordlängdMap = uppdelad.map(s => (s, s.size)).toMap
val grupperaEfterFörstaBokstav = uppdelad.groupBy(s => s(0))
val bokstäver = ord.toVector.filter(_ != ' ')
val antalX = bokstäver.count(_ == 'x')
val grupperade = bokstäver.groupBy(ch => ch)
val antal = grupperade.map(p => p._1 -> p._2.size)
//samma som ovan men utnyttjar "parameter untupling":
val antal2 = grupperade.map((k,v) => k -> v.size)
val sorterat = antal.toVector.sortBy(_._2)
val vanligast = antal.maxBy(_._2)
```

### 9.1.27 Serialisering och deserialisering

- Att **serialisera** innebär att **koda objekt** i minnet till en avkodningsbar **sekvens av symboler**, som kan lagras t.ex. i en fil på din hårddisk.
- Att **de-serialisera** innebär att **avkoda en sekvens av symboler**, t.ex. från en fil, och **återskapa objekt** i minnet.

---

### 9.1.28 Läs text från fil och URL

I paketet `scala.io` finns singelobjektet `Source` med metoderna `fromFile` och `fromUrl` för läsning från fil resp. från URL, alltså Universal Resource Locator, som börjar t.ex. med `http://`

```
def läsFrånFil(filnamn: String): String =  
  val s = scala.io.Source.fromFile(filnamn)  
  try s.mkString finally s.close // säkerställ stängning även vid krasch  
  
def läsRaderFrånFil(filnamn: String): Vector[String] =  
  val s = scala.io.Source.fromFile(filnamn)  
  try s.getLines.toVector finally s.close  
  
def läsFrånWebbsida(url: String): String =  
  val s = scala.io.Source.fromURL(url)  
  try s.mkString finally s.close  
  
def läsRaderWebbsida(url: String, kodning: String = "UTF-8"): Vector[String] =  
  val s = scala.io.Source.fromURL(url, kodning) // läs med given teckenkodning  
  try s.getLines.toVector finally s.close
```

Se vidare veckans övning. Exempel på annan teckenkodning: "ISO-8859-1"

---

### 9.1.29 Serialisering i modulen `introprog.IO`

- I kursens kodbibliotek `introprog` finns ett singelobjekt `IO` som samlar smidiga funktioner för serialisering och de-serialisering.
  - Se api-dokumentation här:  
<http://cs.lth.se/pgk/api/>  
Sök på `IO` och klicka på singelobjektet.
  - Se koden här:  
<https://github.com/lunduniversity/introprog-scalalib/blob/master/src/main/scala/introprog/IO.scala>
  - Om du vill får du gärna använda `introprog.IO` istället för `scala.io.Source` på labben.
-



## 9.2 Övning lookup

### Mål

- ☐ Kunna skapa och använda tupler som parametrar och returvärden.
- ☐ Känna till och kunna använda grundläggande metoder på samlingar.
- ☐ Kunna skapa och använda både oföränderliga och föränderliga mängder.
- ☐ Förstå skillnader och likheter mellan en mängd och en sekvens.
- ☐ Kunna beskriva hur algoritmen linjärsökning fungerar.
- ☐ Kunna skapa och använda både oföränderliga och föränderliga nyckel-värde-tabeller.
- ☐ Kunna använda nyckel-värde-tabeller för att implementera registrering.
- ☐ Förstå likheter och skillnader mellan en nyckel-värde-tabell och en sekvens.
- ☐ Kunna spara och läsa data till/från textfiler på disk.

### Förberedelser

- ☐ Studera begreppen i kapitel 9

### 9.2.1 Grunduppgifter; förberedelse inför laboration

#### Uppgift 1. Para ihop begrepp med beskrivning.

Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

mängd	1	A	leta i sekvens tills sökkriteriet är uppfyllt
nyckel-värde-tabell	2	B	avkoda symbolsekvens och återskapa objekt i minnet
mappning	3	C	en unik identifierare
nyckel	4	D	egenskapen att finnas kvar efter programmets avslut
persistens	5	E	koda objekt till avkodningsbar sekvens av symboler
serialisera	6	F	oordnad samling av mappningar med unika nycklar
de-serialisera	7	G	nyckel -> värde
linjärsöka	8	H	oordnad samling med unika element

**Uppgift 2.** Vad är en mängd? Förklara vad som händer nedan. Varför hamnar elementen i en "konstig" ordning? Varför "försvinner" det element?

```

1 scala> val xs = Vector(1,2,3,1,2,3,4,5,7).toSet
2 xs: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 7, 3, 4)
3 scala> xs.foreach(print)
4 512734

```

#### Uppgift 3. Använda mängder.

Para ihop varje uttryck till vänster med ett uttryck till höger som har samma värde:

Set(1, 2) ++ Set(1, 2)	1	A	3
(1 to 3).toSet	2	B	Set(3)
Vector.fill(3)(1).toSet	3	C	6
Set(1, 2, 3) diff Set(1, 2)	4	D	error: ...
(1 to 7).toSet.apply(8)	5	E	<b>true</b>
Set(1, 2, 3).sorted	6	F	Set(1, 2) - 2
Set(2,4) subsetOf (1 to 7).toSet	7	G	Set(1) + 2 + 3
Set(1, -1, 2, -2).map(_ .abs).sum	8	H	<b>false</b>
Set(1, 1, 1, 1, 1, 5).sum	9	I	Set(1, 2)

**Uppgift 4.** *Räkna unika ord med hjälp av en mängd.* På veckans laboration ska vi göra automatisk språkbehandling av långa texter som vi delar upp i ord. Med metoden `s.split(' ').toVector` kan du dela upp en sträng `s` i en sekvens av ord, där `s` blivit uppdelad i många strängar vid varje blanktecken och alla blanktecken är borttagna.

a) Använd metoderna `split` och `toSet` för skapa ett uttryck som beräknar hur många unika ord det finns i strängen `hej` nedan:

```
scala> val hej = "hej hej hemskt mycket hej"
```

b) Mängder är snabba på att kolla om ett element finns i mängden men du kan inte förvänta dig att elementen finns i någon viss ordning. Det finns en sekvenssamlingsmetod som skapar en sekvens med unika element ur en sekvens och behåller den ursprungliga ordningen. Vad heter metoden?

*Tips:* Leta i snabbreferensen eller sök på nätet. Metoden fungerar på alla samlingar som är av typen `Seq` och har ett namn som börjar med bokstäverna `di`.

**Uppgift 5.** *Skapa 2-tupler med metoden `->` som kan uttalas "mappas till".* Vi har tidigare sett hur två olika värden kan samlas i en 2-tupel, till exempel `(0, true)`. Par kan även skapas med hjälp av metoden `->` enligt nedan. Testa detta i REPL:

```
1 scala> ("Skåne", "Lund")           // ett strängpar med vanlig 2-tupel
2 scala> "Skåne" -> "Lund"           // operatornotation med ->
3 scala> "Skåne".->("Lund")          // punktnotation med -> (inte alls vanligt)
```

Metoden `->` fungerar med alla typer och är en fabriksmetod för par. Metodnamnet liknar en högerpil och illustrerar en mappning från första till andra värdet.

a) Fungerar det på par skapade med `->` att använda metoderna `_1` och `_2`?

b) Deklarera en variabel **val** `huvudstad: Vector[(String, String)]` som innehåller mappningar mellan geografiska områden och deras huvudstäder enligt tabellen nedan.

Sverige	Stockholm
Danmark	Köpenhamn
Grönland	Nuuk
Skåne	Lund

c) Skriv ett uttryck som plockar fram "Lund" ur `huvudstad`.

**Uppgift 6.** *Linjärsöka efter nyckel i sekvens av mappningar.*

a) Implementera funktionen `lookupIndex` nedan med hjälp av samlingsmetoden `indexWhere` så att linjärsökning sker efter index för ett par i sekvensen där key finns på första platsen i paret.

```
def lookupIndex(xs: Vector[(String, String)])(key: String): Int = ???
```

b) Testa din funktion i REPL genom att slå upp index för Skånes huvudstad i sekvensen huvudstad från föregående uppgift.

**Uppgift 7.** *Nyckel-värde-tabell.* En nyckel-värde-tabell är en smart datastruktur som gör att du kan slå upp det värde som en nyckel mappar till *utan* att linjärsökning behöver ske. Värdet plockas fram direkt på en konstant tid, d.v.s. tiden att slå upp ett värde beror *inte* på antalet element i samlingen, utan sker med mycket liten fördröjning.

I Scala heter nyckelvärdetabeller `Map` med stort M och är praktiska att använda i många olika sammanhang. `Map` finns i både en oföränderlig och en förändringsbar variant. Det går med metoder på formen `toXXX` lätt att omvandla mellan en `Map` och en sekvens av par av typen `XXX[(Nyckeltyp, Värdetyp)]`.

a) Deklarera mappen `telnr` nedan i REPL och använd `apply` för att ta reda på telefonnumret till Fröken Ur.

b) Vad har `telnr` för typ?

c) Vad har `telnr.toVector` för typ?

```
val telnr = Map(
  "Anna"      -> 46462229812L,
  "Björn"     -> 46462229009L,
  "Sandra"    -> 46462220368L,
  "Fröken Ur" -> 4690510L,
)
```

En uppsättning `Map`-instanser, vid behov nästlade, kan med fördel användas för att bygga upp en i-minnet-databas där inbyggda samlingsmetoder, t.ex. `map`, `filter`, och **for-yield**-uttryck, ger flexibla och effektiva sökmöjligheter. På veckans laboration ska du göra detta.

Samlingen `Map` är en generalisering av en sekvens, där man kan "indexera", inte bara med ett heltal, utan med vilken typ av värde som helst, t.ex. en sträng. Datastrukturen `Map` kallas också *associativ array*<sup>1</sup> och är implementerad som en s.k. *hashtabell*<sup>2</sup>, men du får vänta till fördjupningskursen innan vi går igenom hur en sådan datastruktur implementeras.

**Uppgift 8.** *Använda nyckel-värdetabell.*

a) Skapa nedan variabler i REPL.

```
val follow = for i <- 2 to 16 by 2 yield (i, i + 1)
val xs = follow.toMap
val ys = xs.toVector
```

Hamnar mappningarna i `ys` i samma ordning som `follow`? Varför?

<sup>1</sup>[https://en.wikipedia.org/wiki/Associative\\_array](https://en.wikipedia.org/wiki/Associative_array)

<sup>2</sup>[https://en.wikipedia.org/wiki/Hash\\_table](https://en.wikipedia.org/wiki/Hash_table)

b) Med `xs` och `ys` deklarerade i REPL enligt ovan, para ihop uttryck till vänster med rätt resultat till höger. Om du är osäker på de sammansatta uttrycken, prova enklare uttryck i REPL och undersök värde och typ hos delresultat.

<code>xs(2) + xs(4)</code>	1	A	8
<code>ys(0)</code>	2	B	7
<code>xs(0)</code>	3	C	(10, 11)
<code>(xs + (0 -&gt; 1)).apply(0)</code>	4	D	1
<code>xs.keySet.apply(2)</code>	5	E	(16, 17)
<code>xs.isDefinedAt 0</code>	6	F	-9
<code>xs.getOrElse(0, 7)</code>	7	G	<b>true</b>
<code>xs.maxBy(_._2)</code>	8	H	<b>false</b>
<code>xs.map(p =&gt; p._1 -&gt; -p._2)(8)</code>	9	I	NoSuchElementException

**Uppgift 9.** *Registrering i förändringsbar nyckel-värde-tabell.* I denna uppgift ska du implementera en hjälpklass för registrering i en frekvenstabell som du sedan ska använda på veckans laboration. Klassen ska heta `FreqMapBuilder` som efter upprepade anrop av metoden `add(s: String): Unit` kan skapa frekvenstabeller av typen `Map[String, Int]`, där nyckel-värde-paren i tabellen anger antalet förekomster av en viss sträng. Du ska utgå från koden nedan.

Klassen använder en förändringsbar tabell internt. Efter att man har lagt till många strängar kan man med metoden `toMap` få en oföränderlig tabell för uppslagning av frekvenser för specifika strängar. Läs i snabbreferensen om vilka extra metoder för uppdatering som erbjuds av `mutable.Map[K, V]`.

```
class FreqMapBuilder:
  private val register = collection.mutable.Map.empty[String, Int]
  def toMap: Map[String, Int] = register.toMap
  def add(s: String): Unit = ???

object FreqMapBuilder:
  def apply(xs: String*): FreqMapBuilder = ???
```

Implementera och testa `FreqMapBuilder`. *Tips:* Du kan t.ex. använda metoderna `+=` och `getOrElse`.

**Uppgift 10.** *Metoden sliding.* I veckans laboration kommer du att ha nytta av metoden `sliding`, som ger en iterator för speciella delsekvenser av en sekvens, vilka kan liknas vid "utsikten" i ett "glidande fönster".

a) Kör nedan i REPL och beskriv vad som händer.

```
1 scala> val xs = Vector("fem", "gurkor", "är", "fler", "än", "fyra", "tomater")
2 scala> xs.sliding(2).toVector
3 scala> xs.sliding(3).toVector
4 scala> xs.sliding(10).toVector
```

b) Använd `xs.sliding(2)` och omvandla varje element i resultatet till ett par. Gör sedan om sekvensen av par till en nyckel-värde-tabell. Vad kan tabellen användas till?

**Uppgift 11.** *Läsa text från fil och webbservrar.* På laborationen ska du bygga upp tabeller från data i textformat. Då har du nytta av att kunna läsa text från filer och från webben. Testa detta i REPL:

```
1 scala> val url = "https://fileadmin.cs.lth.se/pgk/europa.txt"
2 scala> val xs = io.Source.fromURL(url, "UTF-8").getLines.toVector
3 scala> val data = xs.map(_.split(';').toVector)
4 scala> data.head
5 scala> data.foreach(println)
```

a) Skapa dessa tabeller ur sekvensen data:

```
val populationOf: Map[String, Int]    = ??? // länders invånarantal
val sizeOf:      Map[String, Int]    = ??? // länders yta i km^2
val capitalOf:   Map[String, String] = ??? // länders huvudstäder
```

Testa tabellerna i REPL.

b) Spara ner data i en textfil `europa.txt`. Läsa in data från filen med metoden `Source.fromFile(filnamn, teckenkodning)` på liknande sätt som med `fromURL` ovan. Om du kör i en Linux-terminal kan du enkelt ladda ner en fil så här (notera att det är stora bokstaven O och inte en nolla i optionen `-sLO`):

```
> curl -sLO https://fileadmin.cs.lth.se/pgk/europa.txt
```

Skriv ut alla raderna i `europa.txt` med hjälp av `Source.fromFile` i REPL.

## 9.2.2 Extrauppgifter; träna mer

**Uppgift 12.** *Skapa ett textspel med hjälp av tabeller.* Gör ett enkelt spel för att träna på olika fakta om Europas länder och huvudstäder genom att läsa data från URL:en:

<https://fileadmin.cs.lth.se/pgk/europa.txt>

Där finns text kodad i UTF-8 med följande innehåll (endast de första raderna visas):

```
Land;Invånarantal;Storlek(km^2);Huvudstad
Albanien;3581655;28748;Tirana
Andorra;71201;468;Andorra la Vella
Belgien;10584534;30528;Bryssel
Bosnien-Hercegovina;4590310;51129;Sarajevo
Bulgarien;7385367;110910;Sofia
Cypern;854000;9250;Nicosia
Danmark;5475791;43094;Köpenhamn
Estland;1324333;45226;Tallinn
Finland;5315280;338145;Helsingfors
Frankrike;61538322;551695;Paris
Färöarna;48344;139574;Torshamn
Grekland;10964021;131940;Aten
// ... etcetera för alla Europas länder.
```

Låt till exempel användaren svara på slumpvisa frågor av typen:

- Har Andorra fler invånare än Cypern?
- Vad heter huvudstaden i Bulgarien?
- Har Danmark större yta än Finland?

Använd oföränderliga tabeller med lämpliga nycklar och värden. Du kan använda en mängd med länder/huvudstäder som användaren hittills svarat rätt på för att kunna förhindra att dessa återkommer igen.

### 9.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 13.** *Registrering med groupBy.* Vi ska nu utnyttja ett riktigt listigt trick för att via en enda kodrad implementera registrering med hjälp av samlingsmetoderna `groupBy` och `map`.

a) Läs om metoden `groupBy` i snabbreferensen. Du hittar den under rubriken *”Methods in trait Iterable[A]”* eftersom `groupBy` fungerar på alla samlingar. Testa `groupBy` enligt nedan och beskriv vad som händer.

```
1 scala> val xs = Vector(1, 1, 2, 2, 4, 4, 4).groupBy(x => x > 2)
2 scala> val ys = Vector(1, 1, 2, 2, 4, 4, 4).groupBy(x => x)
```

b) Skapa en funktion `freq` med nedan funktionshuvud som returnerar en tabell med antalet förekomster av olika heltal i `xs`. Testa `freq` på en sekvens av 1000 slumpvisa tärningskast och förklara hur funktionen `freq` fungerar. *Tips:* Gör först `groupBy(???)` och sedan `map(???)`.

```
def freq(xs: Vector[Int]): Map[Int, Int] = ???

def kasta(n: Int): Vector[Int] =
  Vector.fill(n)(scala.util.Random.nextInt(6) + 1)
```

**Uppgift 14.** *Skriva till fil.* Som hjälp när du skapar egna intressanta applikationer eller bygger vidare på kursens laborationer och övningar med frivilliga extrauppgifter, kan du använda funktionerna i singelobjektet `I0` nedan, som finns i kursens scala-bibliotek `introprog`.<sup>3</sup>

`IO`-modulen använder `scala.io.Source` för att serialisera och de-serialisera strängar till och från vanliga textfiler. `IO`-modulen använder även paketet `java.io` för att erbjuda funktioner som gör det enkelt att serialisera/de-serialisera godtyckliga objekt skapade med hjälp av serialiserbara klasser till/från binärfiler. Case-klasser i Scala blir automatiskt serialiserbara.

I implementationen av `I0` används `try ... finally` för att säkerställa att filer inte lämnas öppnade även om något går fel under den läs/skriv-process som sköts av det underliggande operativsystemet.

a) Kompilera och resta nedan med `introprog` på classpath, t.ex. med hjälp av `sbt`.

```
import introprog.I0

case class Player(name: String)
```

<sup>3</sup>Källkoden finns här och även på sidan ??:

<https://github.com/lunduniversity/introprog-scalalib/blob/master/src/main/scala/introprog/I0.scala>

```
@main def run(): Unit =  
  println("Test of output/input objects to/from disk:")  
  val highscores = Map(Player("Sandra") -> 42, Player("Björn") -> 5)  
  IO.saveObject(highscores, "highscores.ser")  
  val highscores2 = IO.loadObject[Map[Player, Int]]("highscores.ser")  
  val isSameContents = highscores2 == highscores  
  val testResult = if (isSameContents) "SUCCESS :)" else "FAILURE :("  
  println(testResult)
```

b) Använd IO-modulen för att spara användarens poängresultat i ditt spel om Europas länder och städer, i extrauppgiften ovan. Implementationen av `introprog.IO` finns här: <https://github.com/lunduniversity/introprog-scalalib/blob/master/src/main/scala/introprog/IO.scala>



## 9.3 Laboration: words

### Mål

- ☐ Kunna skapa och använda nyckel-värde-tabeller med samlingstypen Map.
- ☐ Kunna skapa och använda mängder med samlingstypen Set.
- ☐ Förstå skillnader och likheter mellan en sekvens och en mängd.
- ☐ Förstå likheter och skillnader mellan en sekvens av par och en nyckel-värde-tabell.
- ☐ Kunna implementera algoritmer som använder nästlade strukturer.

### Förberedelser

- ☐ Gör övning lookup i avsnitt 9.2
- ☐ Läs igenom hela laborationen.
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).

### 9.3.1 Bakgrund

Denna uppgift handlar om analys av naturligt språk (eng. *Natural Language Processing, NLP*). Språkanalys bygger ofta på statistik över förekomsten av olika ord i långa texter. Du ska skriva kod, som utifrån en lång text, till exempel en bok, kan hjälpa dig att svara på denna typ av frågor:

- Hur vanligt är ett visst ord i en given text?
- Vilket är det vanligaste ordet som följer efter ett visst ord?
- Hur kan man generera ordsekvenser som liknar ordföljden i en given text?

För att kunna svara på sådana frågor ska du skapa frekvenstabeller och även så kallade *n-gram*; sekvenser av *n* ord som förekommer i följd i en text. Exempel på några 2-gram (kallas även *bigram*) som finns i föregående mening: (för, att), (att, kunna), (kunna, svara), (svara, på), (på, sådana), och så vidare.<sup>4</sup>

### 9.3.2 Obligatoriska uppgifter

Du ska bygga ditt program med en editor, t.ex. VS code, och kompilera din kod i terminalen med `scalac` eller med hjälp av byggverktyget `sbt`. Medan du steg för steg utvecklar ditt program, ska du parallellt göra experiment i REPL för att undersöka hur du kan använda samlingsmetoder för att lösa uppgifterna.

Kod att utgå ifrån finns på github här: [https://github.com/lunduniversity/introprog/tree/master/workspace/w09\\_words](https://github.com/lunduniversity/introprog/tree/master/workspace/w09_words)

Dessa ofärdiga kodfiler ligger i paketet `nlp`:

- `FreqMapBuilder.scala` innehåller ett skelett till en klass för att, ord för ord, bygga en nyckel-värde-tabell som registrerar antalet förekomster av olika ord. Att implementera denna ingick i övningen du gjorde tidigare i veckan.
- `Text.scala` innehåller ett skelett till en klass som kan göra textbehandling genom att analysera ord i en text.
- `Main.scala` innehåller ett ofärdigt huvudprogram som du kan använda i laborationens senare del.

---

<sup>4</sup>Du kan undersöka olika *n-gram* i en stor mängd böcker med hjälp av Googles *n-gram-viewer*: <https://books.google.com/ngrams/>



För att underlätta ditt arbetsflöde under det att du stegvis bygger upp din kod metod för metod, kan du med fördel använda byggverktyget sbt (se appendix F) så här:

- Med sbt-kommandot `console` startar du REPL innifrån sbt med dina klasser automatiskt tillgängliga på classpath och du kan anropa de metoder som du gjort färdigt hittills medan du gör experiment inför nästa steg. När du ändrat något i din editor och vill experimentera med nya versionen så trycker du Ctrl+D och startar om REPL med `console` (pil-upp) och din kod kompileras om automatiskt.
- Med sbt-kommandot `~run` (notera tilde-tecknet) sker kompilering och körning av main-metoden automatiskt i terminalen varje gång du gör Ctrl+S i din editor.

**Uppgift 1.** *Skapa frekvenstabeller.* Du ska använda `FreqMapBuilder` från veckans övning för att skapa frekvenstabeller av typen `Map[String, Int]`, där nyckel-värde-paren i tabellen anger antalet förekomster av en viss sträng.

a) Lägg klassen `FreqMapBuilder` i ett paket som heter `nlp` och kompilera.

```

1 package nlp
2
3 class FreqMapBuilder:
4   private val register = scala.collection.mutable.Map.empty[String, Int]
5   def toMap: Map[String, Int] = register.toMap
6   def add(s: String): Unit = ???
7
8 object FreqMapBuilder:
9   /** Skapa ny FreqMapBuilder och räkna strängarna i xs */
10  def apply(xs: String*): FreqMapBuilder = ???

```

b) Testa noga så att din `FreqMapBuilder` fungerar korrekt. Exempel på test i REPL:

```

1 scala> import nlp._
2
3 scala> val fmb = FreqMapBuilder("hej", "på", "dej")
4 fmb: nlp.FreqMapBuilder = nlp.FreqMapBuilder@458f85ef
5
6 scala> fmb.add("hej")
7
8 scala> fmb.toMap
9 res0: Map[String,Int] = Map(på -> 1, hej -> 2, dej -> 1)
10
11 scala> (1 to Short.MaxValue).foreach(i => fmb.add(i.toString))
12
13 scala> fmb.toMap.size
14 res1: Int = 32770
15
16 scala> fmb.toMap
17 res2: Map[String,Int] = Map(10292 -> 1, 19125 -> 1, 26985 -> 1, 29301 -> 1, 5451 -> 1, 4018 -> 1, 31

```

I kommande uppgifter ska du steg för steg skapa och testa case-klassen `Text`.

```

1 package nlp
2
3 case class Text(source: String):
4   lazy val words: Vector[String] = ??? // dela upp source i ord
5
6   lazy val distinct: Vector[String] = words.distinct
7
8   lazy val wordSet: Set[String] = words.toSet
9
10  lazy val wordsOfLength: Map[Int, Set[String]] = wordSet.groupBy(_.length)
11
12  lazy val wordFreq: Map[String, Int] = ??? // använd FreqMapBuilder
13
14  def ngrams(n: Int): Vector[Vector[String]] = ??? // använd sliding
15
16  lazy val bigrams: Vector[(String, String)] =
17    ngrams(2).map(xs => (xs(0), xs(1)))
18
19  lazy val followFreq: Map[String, Map[String, Int]] = ??? //nästlad tabell
20
21  lazy val follows: Map[String, String] =
22    followFreq.map( (key, followMap) =>
23      val maxByFreq: (String, Int) = followMap.maxBy(_._2)
24      val mostCommonFollower: String = maxByFreq._1
25      (key, mostCommonFollower)
26    )
27    //eller kortare med samma resultat: (lättare eller svårare att läsa?)
28    // followFreq.map((k, v) => k -> v.maxBy(_._2)._1)
29
30 object Text:
31   def fromFile(fileName: String, encoding: String = "UTF-8"): Text =
32     val source = scala.io.Source.fromFile(fileName, encoding)
33     val txt = try source.mkString finally source.close()
34     Text(txt)
35
36   def fromURL(url: String, encoding: String = "UTF-8"): Text =
37     val source = scala.io.Source.fromURL(url, encoding)
38     val txt = try source.mkString finally source.close()
39     Text(txt)

```

**Uppgift 2.** Dela upp en sträng i ord. Du ska implementera medlemmen `words`. Den ska innehålla en vektor med alla ord i `source`, utan andra tecken än bokstäver. Detta åstadkommer du genom att utgå ifrån strängen `source` och i tur och ordning göra följande:

1. byta ut alla tecken i `source` för vilka `isLetter` är falskt mot ' '
2. dela upp strängen från föregående steg i en array av strängar med `split(' ')`
3. filtrera bort alla tomma strängar
4. gör om alla bokstäver i alla strängar till små bokstäver
5. gör om arrayen till en sekvens av typen `Vector[String]`.

Testa så att `words`, och de värden som använder `words`, fungerar i REPL:

```

1 scala> val t = Text("Gurka är ingen tomat, men gurka är en grönsak.")
2
3 scala> t.words

```

```

4 res1: Vector[String] =
5   Vector(gurka, är, ingen, tomat, men, gurka, är, en, grönsak)
6
7 scala> t.distinct
8 res2: Vector[String] =
9   Vector(gurka, är, ingen, tomat, men, en, grönsak)
10
11 scala> t.wordSet
12 res3: Set[String] = Set(grönsak, är, gurka, men, ingen, tomat, en)
13
14 scala> t.wordsOfLength(5)
15 res4: Set[String] = Set(gurka, ingen, tomat)

```

**Uppgift 3.** Du ska nu skapa ordfrekvenstabellen `wordFreq` genom att registrera ordförekomster med hjälp av `FreqMapBuilder`. Tabellen `wordFreq` ska bestå av nyckelvärdepar `w -> f` där `f` är antalet gånger ordet `w` förekommer i `words`. Testa `wordFreq` genom att ladda ner boken ”Skattkamararön” skriven av Robert Louis Stevenson<sup>5</sup> och undersök frekvensen för olika vanliga ord. Vilket ord är vanligast? Näst vanligast?

```

1 scala> val piratbok = Text.fromURL("https://fileadmin.cs.lth.se/pgk/skattkamararon.txt")
2 val piratbok: nlp.Text = Text(Herr Trelawney, doktor Livesey och de övriga herrarna har bett mig att skriva ner
3
4 scala> piratbok.words.size
5 val res0: Int = 69438
6
7 scala> piratbok.wordFreq("pirat")
8 val res1: Int = 7

```

Länkar till böcker i UTF-8-format som du kan använda i dina tester:

- ”Skattkamararön” av R. L. Stevenson:  
<https://fileadmin.cs.lth.se/pgk/skattkamararon.txt>
- ”Inferno” av August Stringberg:  
<https://fileadmin.cs.lth.se/pgk/inferno.txt>
- ”Pride and Prejudice” av Jane Austen:  
<https://fileadmin.cs.lth.se/pgk/prideandprejudice.txt>
- Projekt Gutenberg med många fritt tillgängliga böcker i textformat:  
<https://www.gutenberg.org/>

**Uppgift 4.** Implementera metoden `ngrams` som ger en sekvens med alla ordföljder i  $n$  steg. *Tips:* På veckans övning ingick att undersöka hur metoden `sliding` fungerar, med vilken du kan skapa  $n$ -gram. Gör `toVector` på resultatet från `sliding`. Testa noga så att `ngrams` och `bigrams` fungerar korrekt innan du går vidare.

```

1 scala> piratbok.ngrams(3).take(2)
2 val res1: Vector[Vector[String]] =
3   Vector(Vector(herr, trelawney, doktor), Vector(trelawney, doktor, livesey))
4
5 scala> piratbok.bigrams.take(2)
6 val res2: Vector[(String, String)] =
7   Vector((herr,trelawney), (trelawney,doktor))

```

<sup>5</sup>Copyright för denna bok har gått ut, så du gör dig inte skyldig till piratkopiering (i juridisk mening).

**Uppgift 5.** Implementera `followFreq`, som ska innehålla en nyckel-värde-tabell där värdet i sin tur är en frekvenstabell över de ord som kommer efter nyckeln.

Genom att analysera alla ordpar kan vi få fram vilket som är det vanligaste ordet som följer efter ett givet ord. Metoden `bigrams` ger oss alla ordpar (`w1`, `w2`) där `w2` följer efter `w1`. Vi kan spara statistiken över efterföljande ord i en nyckelvärde-tabell med mappningarna `w -> f` där nyckeln `w` är ett ord och värdet `f` är en frekvenstabell av typen `Map[String, Int]`. I frekvenstabellen lagrar vi frekvensen för alla de ord som följer efter `w`. Du ska alltså bygga en nästlad tabell av typen `Map[String, Map[String, Int]]`. Rita en bild av den nästlade strukturen.

Implementera metoden `followFreq` genom att utgå från nedan pseudokod:

```
val result = scala.collection.mutable.Map.empty[String, FreqMapBuilder]
for (key, next) <- bigrams do
  if /* key finns redan definierad i result */ then
    /* på "platsen" result(key): lägg till next i frekvenstabellen */
  else
    /* lägg till (key -> ny frekvenstabell med next) i result*/
end for
result.map(p => p._1 -> p._2.toMap).toMap // toMap ger oföränderlig Map
```

Skriv uttryck för att ta reda på följande:

- Vilka ord brukar följa efter *han* respektive *hon* i Stevensons "Skattkammarön"?
- Vilka ord brukar följa efter *han* respektive *hon* i Stringbergs "Inferno"?
- Vilka ord brukar följa efter *he* respektive *she* i Austens "Pride and Prejudice"?

**Uppgift 6.** Skapa ett huvudprogram som rapporterar valfria, intressanta mått om orden i en text. Programmet ska ta textens källa som argument, givet som en URL eller ett filnamn. Skriv huvudprogrammet i filen `Main.scala` i ett singelobjekt med namnet `Main`. Exempel på en rapport som ditt huvudprogram kan generera finns nedan. Här ges även ett heltal som argument som styr topplistornas längd.

```
1 > scala nlp.Main https://fileadmin.cs.lth.se/pgk/skattkammaron.txt 13
2
3 Källa: https://fileadmin.cs.lth.se/pgk/skattkammaron.txt
4
5 *** Antal ord: 69438
6
7 *** De 13 vanligaste orden och deras frekvens:
8 (och,3089), (jag,2007), (att,1594), (det,1382), (en,1262),
9 (i,1244), (som,1132), (på,1068), (han,1063), (var,990),
10 (med,854), (den,774), (av,740)
11
12 *** De 13 längsta orden och deras längd:
13 (besättningsmedlemmarnas,23), (befästningsanordningar,22),
14 (temperamentsuppvissning,22), (undsättningsexpedition,22),
15 (besättningsmedlemmarna,22), (försiktighetsåtgärder,21),
16 (undsättningsfartyget,20), (sjukdomsframkallande,20),
17 (husföreståndarinnans,20), (sjömansterminologin,19),
18 (parlamentärsflaggan,19), (begravningsplatsen,19),
19 (tidvattenströmmarna,19)
```

Exempel på huvudprogram som kan skapa ovan utskrift:

```
1 package nlp
2
```

```

3 object Main:
4   val defaultUrl = "https://fileadmin.cs.lth.se/pgk/skattkammaron.txt"
5   val defaultN = 10
6
7   def top(n: Int, freqMap: Map[String, Int]): Vector[(String, Int)] = ???
8
9   def report(text: Text, from: String, n: Int): String =
10    val longestWordsWithLength =
11      top(n, text.distinct.map(w => (w, w.length)).toMap).mkString(", ")
12    s"""
13    |Källa: $from
14    |
15    |*** Antal ord: ${text.words.size}
16    |
17    |*** De $n vanligaste orden och deras frekvens:
18    |${top(n, text.wordFreq).mkString(", ")}
19    |
20    |*** De $n längsta orden och deras längd:
21    |$longestWordsWithLength
22    """.stripMargin
23
24  def main(args: Array[String]): Unit =
25    val location = if args.isEmpty then defaultUrl else args(0)
26    val n = if args.length < 2 then defaultN else args(1).toInt
27    val text =
28      if location.startsWith("http") then Text.fromURL(location)
29      else Text.fromFile(location)
30
31    println(report(text, location, n))

```

### 9.3.3 Kontrollfrågor

1. I vilken ordning hamnar elementen om man anropar `distinct` på en sekvens?
2. Om man itererar över en mängd, i vilken ordning behandlas elementen?
3. Ge exempel på när är det lämpligt att använda en mängd i stället för en sekvens av distinkta värden?
4. Är alla nycklar i en nyckel-värde-tabell garanterat unika?
5. Är alla värden i en nyckel-värde-tabell garanterat unika?
6. LTH-teknologen Oddput Clementin vill summera längden på varje sträng i en mängd och skriver:

```

1 scala> Set("hej", "på", "dej").map(_.length).sum
2 res0: Int = 5

```

Varför blir det fel? Hur kan Oddput åtgärda problemet?

### 9.3.4 Frivilliga uppgifter

**Uppgift 7.** Bygg vidare på klassen `Text` och implementera nedan metod som ska ge ett slumpmässigt ord ur `wordSet`. Varje ord ska förekomma med lika stor sannolikhet.

```
def randomWord: String = ???
```

**Uppgift 8.** Med NLP kan man generera slumpmässiga meningar som statistiskt sett liknar ”riktiga”, människoskapade meningar.

Implementera metoden `randomSeq(firstWord, n)` nedan i klassen `Text`. Den ska ge en sekvens  $w_1, w_2, \dots, w_n$  där  $w_1$  är `firstWord` och  $w_{i+1}$  är något slumpmässigt ord som är draget bland de ord som följer efter  $w_i$ . Detta kan du åstadkomma genom att varje efterföljande ord  $w_{i+1}$  väljs ur `keys.toVector` för den `followFreq`-tabell som hör till  $w_i$ . Orden ska dras ur efterföljandemängden, med lika stor sannolikhet.

```
def randomSeq(firstWord: String, n: Int): Vector[String] = ???
```

**Uppgift 9.** För att dina datorgenererade meningar verkligen ska likna mänskligt språk kan vi skapa de mest sannolika meningarna av olika längder ur vår analys av ordfrekvenser.

Lägg till metoden `mostCommonSeq` i klassen `Text` enligt nedan:

```
def mostCommonSeq(firstWord: String, n: Int): Vector[String] = ???
```

a) Implementera metoden så att resultatet blir en sekvens med  $n$  ord. Sekvensen ska börja med `firstWord` och därefter följas av det ord som är det *vanligaste* efterföljande ordet efter `firstWord`, och därpå det vanligaste efterföljande ordet efter det, etc. *Tips:* Använd en lokal variabel **val** `result` som är en `ArrayBuffer` till vilken du i en **while**-loop lägger de efterföljande orden.

b) Jämför de slumpmässiga sekvenserna med sekvenser genererade med `randomSeq` i uppgift 8. Vilka sekvenser liknar mest "riktiga" meningar?

**Uppgift 10.** Använd befintliga samlingsmetoder i stället för `FreqMapBuilder` för att registrera efterföljande ord.

a) Undersök i REPL hur metoden `groupBy(x => x)` fungerar då den appliceras på en samling med strängar. Sök efter och studera dokumentationen för `groupBy`.

b) Inför värdet **lazy val** `wordFreq2`. Den ska ge samma resultat som `wordFreq` men implementeras med hjälp av `groupBy` och `map` i stället för `FreqMapBuilder`.

★ c) Jämför prestanda mellan `wordFreq2` och `wordFreq`. Vilken är snabbast för stora texter? Är skillnaden stor?

d) Inför värdet **lazy val** `followsFreq2`. Den ska ge samma resultat som `followsFreq` men implementeras med hjälp av `groupBy` och `map` i stället för `FreqMapBuilder`. Denna uppgift är ganska knepig. Experimentera dig fram i REPL, och bygg upp en lösning steg för steg. *Tips:*

```
bigrams
  .groupBy(???)
  .map(p => p._1 -> p._2.map(???) .groupBy(???) .map(???) )
```

★ e) Jämför prestanda mellan `followsFreq2` och `followsFreq`. Vilken är snabbast för stora texter? Är skillnaden stor?

**Uppgift 11.** Gör `FreqMapBuilder` generisk. Generiska strukturer, alltså sådana som har typparametrar, är ofta väsentligt mycket mer användbara. Om du gör `FreqMapBuilder` generisk genom att införa en typparameter i stället för att hårdkoda typen till `String` så kan du använda `FreqMapBuilder` med godtycklig elementtyp.

a) Studera `FreqMapBuilder` och identifiera allt i den klassen som är specifikt för typen `String`.

- b) Inför en typparameter `A` inom hakparenteser efter klassnamnet och använd sedan `A` i stället för `String` i alla metoder.
- c) Testa så att din generiska frekvenstabellbyggare fungerar på sekvenser som innehåller annat än strängar.

Detta funkar eftersom inget i `FreqMapBuilder` egentligen förutsätter att elementen som ska räknas är av sträng-typ (det räcker att det finns en vettig `equals` och `hashCode`).

# Kapitel 10

## Arv och komposition

Begrepp som ingår i denna veckas studier:

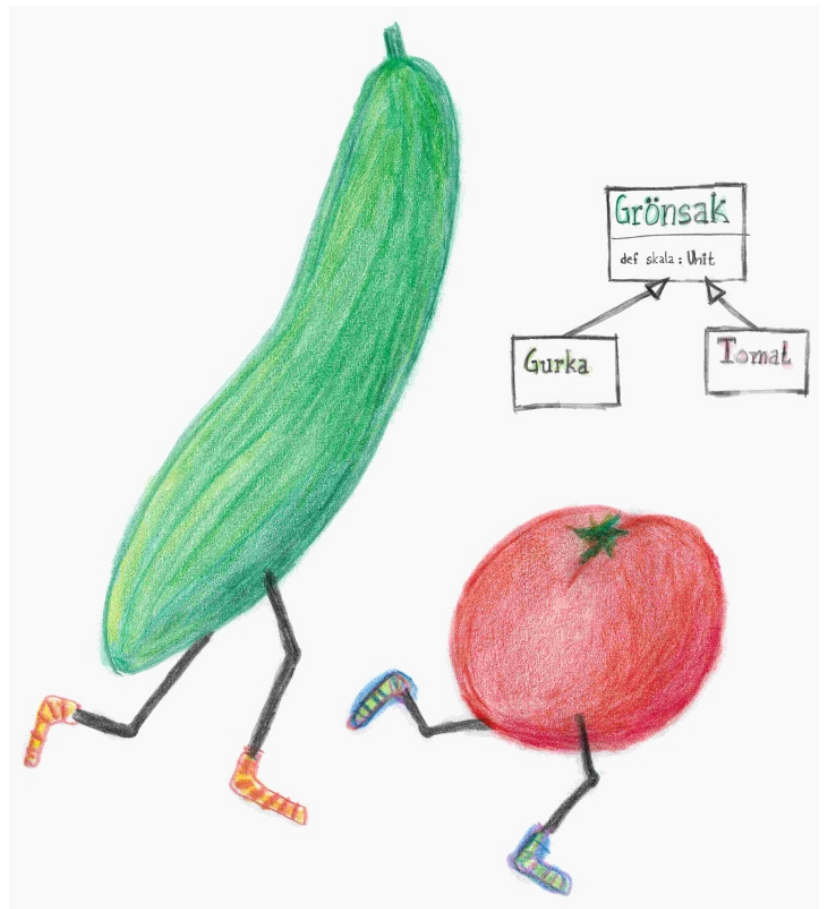
- ☐ arv
- ☐ komposition
- ☐ polymorfism
- ☐ trait
- ☐ extends
- ☐ asInstanceOf
- ☐ with
- ☐ inmixning supertyp
- ☐ subtyp
- ☐ bastyp
- ☐ override
- ☐ Scalas typhierarki
- ☐ Any
- ☐ AnyRef
- ☐ Object
- ☐ AnyVal
- ☐ Null
- ☐ Nothing
- ☐ topptyp
- ☐ bottentyp
- ☐ referenstyper
- ☐ värdetyper
- ☐ accessregler vid arv
- ☐ protected
- ☐ final
- ☐ trait
- ☐ abstrakt klass



## 10.1 Teori

### 10.1.1 Vad är arv?

Arv (eng. *inheritance*) beskriver relationen  
**X är en Y**

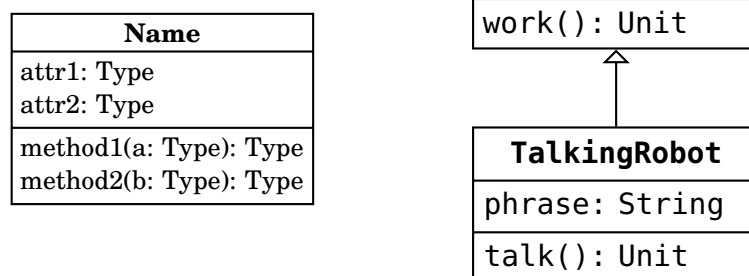


### 10.1.2 Varför behövs arv?

- Man kan använda arv för att dela upp kod i:
  - **generella** (gemensamma) delar och
  - **specifika** (specialanpassade) delar.
- Man kan åstadkomma **kontrollerad flexibilitet**:
  - Klientkod kan **utvidga** (eng. *extend*) ett givet API med egna specifika tillägg.
- Man kan använda arv för att deklarera en gemensam **bastyp** så att generiska samlingar kan ges en mer specifik elementtyp.
  - Det räcker att man vet bastypen för att kunna anropa gemensamma metoder på alla element i samlingen.

### 10.1.3 Klassdiagram med UML (Unified Modeling Language)

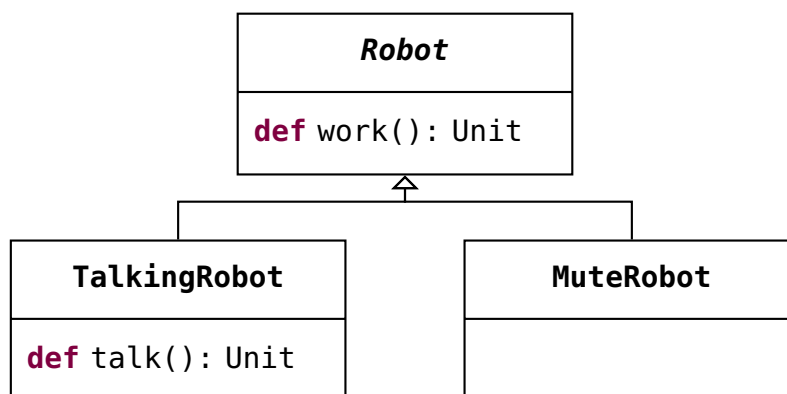
UML Klassdiagram:



Mer om UML-diagram i pfk, OMD.

[en.wikipedia.org/wiki/Class\\_diagram](https://en.wikipedia.org/wiki/Class_diagram)

### 10.1.4 Exempel: Robot som bastyp för två subtyper

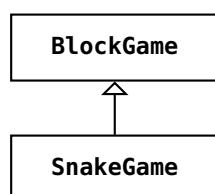


(Ibland utelämnar man attribut och/eller metoder i ett UML-diagram för att minska antalet detaljer i modellen och ge en överblick.)

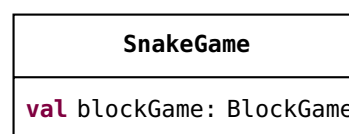
### 10.1.5 Alternativ till arv: komposition

- Som alternativ till att klassen X ärver klassen Y kan man i stället använda **komposition** (eng. *composition*), som innebär att klassen X **har ett attribut** som **refererar** till klassen Y.

Exempel på arv i snake-labben:  
SnakeGame **är** ett BlockGame



Ett alternativ vore **komposition**:  
SnakeGame **har** ett BlockGame

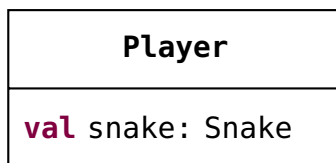


- Komposition där **X har en Y** är ofta ett bättre alternativ än arv om:
  - det *inte* finns en tydlig **X-är-en-Y**-relation

2. det *inte* behövs en **gemensam bastyp** i en hierarki.
3. det *inte* är önskvärt ärva och exponera *alla* Y:s medlemmar via X

### 10.1.6 Exempel på komposition i snake-labben

En Player **har en**<sup>1</sup> snake.



```
class Player(val snake: Snake)
```

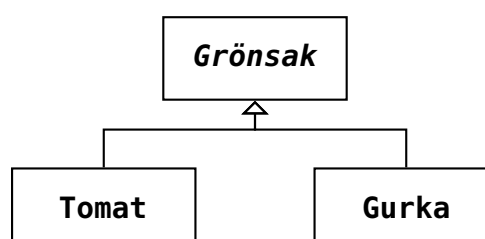
### 10.1.7 Behovet av gemensam bastyp

```
1 scala> class Gurka(val vikt: Int)
2
3 scala> class Tomat(val vikt: Int)
4
5 scala> val gurkor = Vector(new Gurka(200), new Gurka(300))
6 gurkor: scala.collection.immutable.Vector[Gurka] =
7   Vector(Gurka@60856961, Gurka@2fd953a6)
8
9 scala> gurkor.map(_.vikt)
10 res0: scala.collection.immutable.Vector[Int] = Vector(200, 300)
11
12 scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
13 grönsaker: scala.collection.immutable.Vector[Object] =
14   Vector(Gurka@669253b7, Tomat@5305c37d)
15
16 scala> grönsaker.map(_.vikt)
17 <console>:15: error: value vikt is not a member of Object
18   grönsaker.map(_.vikt)
```

Hur ordna en mer specifik typ än Vector[Object]? → Skapa en **bastyp**!

### 10.1.8 Skapa en gemensam bastyp

Typen **Grönsak** är en **bastyp** i nedan arvshierarki:



<sup>1</sup>Läs mer om komposition och de relaterade begreppen aggregering, association, ägarskap etc. här: [https://en.wikipedia.org/wiki/Object\\_composition](https://en.wikipedia.org/wiki/Object_composition)

Pilen  $\uparrow$  betecknar **arv** och utläses ”**är en**”

Typerna Tomat och Gurka är **subtyper** till den **abstrakta** typen Grönsak.  
En typ kallas abstrakt om den inte kan instansieras.

### 10.1.9 Skapa en gemensam bas typ med trait och extends

Med **trait** Grönsak kan klasserna Gurka och Tomat få en gemensam **bastyp** genom att båda **subtyperna** gör **extends** Grönsak:

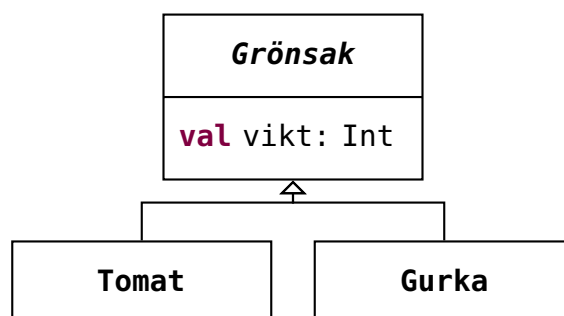
```
1 scala> trait Grönsak
2
3 scala> class Gurka(val vikt: Int) extends Grönsak
4
5 scala> class Tomat(val vikt: Int) extends Grönsak
6
7 scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
8 grönsaker: scala.collection.immutable.Vector[Grönsak] =
9   Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)
```

Men det är fortfarande inte som vi vill ha det:

```
scala> grönsaker.map(_.vikt)
<console>:15: error: value vikt is not a member of Grönsak
  grönsaker.map(_.vikt)
```

### 10.1.10 En gemensam bas typ med gemensamma delar

Placera gemensamma medlemmar i bas typen:



- Alla grönsaker har attributet **val** vikt.
- Det specifika värdet på vikten definieras **inte** i bas typen.
- Medlemmen vikt kallas **abstrakt** eftersom den **saknar implementation**.

### 10.1.11 Placera gemensamma delar i basstypen

Vi inkluderar det gemensamma attributet **val** vikt som en **abstrakt medlem** i basstypen:

```
trait Grönsak:  
  val vikt: Int    // implementation saknas, inget =  
  
class Gurka(val vikt: Int) extends Grönsak  
  
class Tomat(val vikt: Int) extends Grönsak
```

Nu vet kompilatorn att alla grönsaker har en vikt:

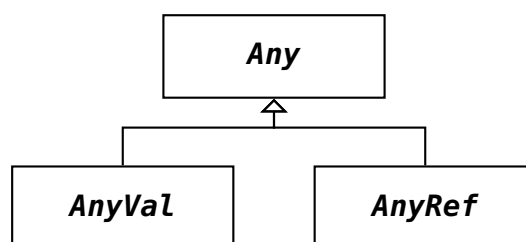
```
scala> val grönsaker = Vector(new Gurka(200), new Tomat(42)) //funkar även utan new  
val grönsaker: Vector[Grönsak] =  
  Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)  
  
scala> grönsaker.map(_.vikt)  
val res0: Vector[Int] = Vector(200, 42)
```

Den abstrakta medlemmen vikt i den abstrakta typen Grönsak **implementeras** i en konkret subklass, här som en klassparameter.

---

### 10.1.12 Scalas typhierarki

En förenklad bild av den översta delen av typhierarkin i Scala:

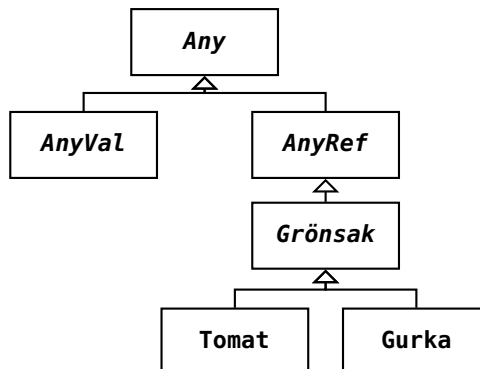


- De numeriska typerna Int, Double, etc är subtyper till **AnyVal** och kallas **värdetyper** och lagras på ett speciellt, effektivt sätt i minnet.
- Alla dina egna klasser är subtyper till **AnyRef** och kallas **referenstyper** och kan (direkt eller indirekt) konstrueras med **new**.
- AnyRef motsvaras av **java.lang.Object** i JVM.
- (För att skilja ut s.k. opaka typer (ingår ej i denna kurs), som inte kan undersökas med mönstermatchning, så är finns även typen Matchable som är subtyp till Any och supertyp till AnyRef och AnyVal.)

---

### 10.1.13 Implicita supertyper till dina egna klasser

Alla dina egna typer ingår underförstått i Scalas typhierarki:



### 10.1.14 Vad är en trait?

- **Trait** betyder **egenskap**.
- En trait liknar en klass, **men** speciella regler gäller:
  - den **kan** innehålla delar som **saknar implementation**
  - den **kan mixas** med flera andra traits så att olika koddelar kan återanvändas på flexibla sätt.
  - den **kan inte** instansieras direkt eftersom den är abstrakt.
  - Från Scala 3 **kan** den ha **parametrar** (precis som klasser).

### 10.1.15 Vad används en trait till?

En **trait** används för att skapa en bas typ som kan vara hemvist för gemensamma delar hos subtyper:

```

trait Bastyp { val x = 42 }           // Bastyp har medlemmen x
class Subtyp1 extends Bastyp { val y = 43 } // Subtyp1 ärver x, har även y
class Subtyp2 extends Bastyp { val z = 44 } // Subtyp2 ärver x, har även z
  
```

```

scala> val a = new Subtyp1
val a: Subtyp1 = Subtyp1@51016012

scala> a.x
val res0: Int = 42

scala> a.y
val res1: Int = 43

scala> a.z
-- Error:
   value z is not a member of Subtyp1

scala> new Bastyp
--Error:
   Bastyp is a trait; it cannot be instantiated
  
```

### 10.1.16 En trait kan ha abstrakta medlemmar

```
trait X { val x: Int }    // x är abstrakt, d.v.s. saknar implementation
class A extends X { val x = 42 }    // x ges en implementation
class B extends X { val x = 43 }    // x ges en annan implementation
```

```
1 scala> val a = new A
2 val a: A = A@5faeada1
3
4 scala> val b = B()    // fungerar utan new men då behövs ()
5 val b: B = B@cb51256
6
7 scala> val xs = Vector(a,b)
8 val xs: Vector[X] = Vector(A@5faeada1, B@cb51256)
9
10 scala> xs.map(_._x)
11 val res0: Vector[Int] = Vector(42, 43)
12
13 scala> class Y { val y: Int }
14 -- Error:
15   class Y needs to be abstract, since val y: Int in class Y is not defined
```

### 10.1.17 En trait kan ha parametrar

```
trait X(val x: Int)
class A extends X(42)
class B(y: Int) extends X(y) // värdet av y blir argument till x i X
```

```
1 scala> val a = A()
2 val a: A = A@5faeada1
3
4 scala> val b = B(43)
5 val b: B = B@cb51256
6
7 scala> val xs = Vector(a,b)
8 val xs: Vector[X] = Vector(A@5faeada1, B@cb51256)
9
10 scala> xs.map(_._x)
11 val res0: Vector[Int] = Vector(42, 43)
12
13 scala> b.y
14 -- Error:
15   value y cannot be accessed as a member of (b : B)
```

Hur kan vi göra medlemmen y synlig? Gör det till en **val**-parameter.

### 10.1.18 Abstrakta och konkreta medlemmar

```
1 object exempelVego1:
2
3   trait Grönsak:
4     var vikt: Double    // abstrakt medlem, saknar implementation
5     var ärSkalad: Boolean = false // konkret medlem, har implementation
```

```

6
7     def skala(): Unit                // abstrakt medlem, saknar implementation
8
9     class Gurka(var vikt: Double) extends Grönsak:
10         def skala(): Unit =          // implementation specifik för Gurka
11             if !ärSkalad then
12                 println("Skalas med skalare.")
13                 vikt = 0.99 * vikt
14                 ärSkalad = true
15
16     class Tomat(var vikt: Double) extends Grönsak:
17         def skala(): Unit =          // implementation specifik för Tomat
18             if !ärSkalad then
19                 println("Skållas.")
20                 vikt = 0.99 * vikt
21                 ärSkalad = true

```

### 10.1.19 Undvika kodduplicering med hjälp av arv

```

1     object exempelVego2:
2
3     trait Grönsak: // innehåller gemensamma delar; hjälper oss undvika upprepning
4         val skalningsmetod: String // abstrakt
5         val skalfaktor = 0.99      // konkret
6         var vikt: Double           // abstrakt
7         var ärSkalad: Boolean = false // konkret
8
9         def skala(): Unit = if !ärSkalad then // konkret
10             println(skalningsmetod)
11             vikt = skalfaktor * vikt
12             ärSkalad = true
13
14     class Gurka(var vikt: Double) extends Grönsak: // det som är speciellt för gurkor
15         val skalningsmetod = "Skalas med skalare."
16
17     class Tomat(var vikt: Double) extends Grönsak: // det som är speciellt för tomater
18         val skalningsmetod = "Skållas."

```

### 10.1.20 Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg
- Fler kodrader att underhålla:
  - Om man rättar en bug på ett ställe måste man komma ihåg att göra **exakt samma ändring** på alla de ställen där kodduplicering förekommer → **risk för nya buggar**
- Principen på engelska: **def** DRY = "Don't Repeat Yourself!"



- Men det finns tillfällen när **kodduplicering faktiskt är att föredra**: t.ex. om man vill att olika delar av koden ska vara **helt oberoende** av varandra.

### 10.1.21 Överskuggning

```

1 object exempelVego3:
2
3   trait Grönsak:
4     val skalningsmetod: String      // abstrakt
5     val skalfaktor = 0.99           // konkret
6     var vikt: Double                // abstrakt
7     var ärSkalad: Boolean = false  // konkret
8
9     def skala(): Unit = if !ärSkalad then
10      println(skalningsmetod)
11      vikt = skalfaktor * vikt
12      ärSkalad = true
13
14   class Gurka(var vikt: Double) extends Grönsak:
15     //nyckelordet override behövs ej om abstrakt medlem i supertyp
16     val skalningsmetod = "Skalas med skalare."
17
18   class Tomat(var vikt: Double) extends Grönsak:
19     //nyckelordet override behövs ej om abstrakt medlem i supertyp men tillåtet:
20     override val skalningsmetod = "Skållas."
21     // override val skalningmetod = "Skållas." //kompilatorn hittar stavfelet!
22     override val skalfaktor = 0.95 // override måste anges vid ändrad impl.

```

### 10.1.22 En final medlem kan ej överskuggas

```

1 object exempelVego4:
2
3   trait Grönsak:
4     val skalningsmetod: String
5     final val skalfaktor = 0.99 // en final medlem kan ej överskuggas
6     var vikt: Double
7     var ärSkalad: Boolean = false
8
9     def skala(): Unit = if !ärSkalad then
10      println(skalningsmetod)
11      vikt = skalfaktor * vikt
12      ärSkalad = true
13
14   class Gurka(var vikt: Double) extends Grönsak:
15     val skalningsmetod = "Skalas med skalare."
16
17   class Tomat(var vikt: Double) extends Grönsak:
18     val skalningsmetod = "Skållas."
19     // override val skalfaktor = 0.95
20     // ger KOMPILERINGSFEL: "cannot override final member"

```

### 10.1.23 Protected ger synlighet begränsad till subtyper

```
scala> trait Super:
  private val minHemlis = 42
  protected val vårHemlis = 42

scala> class Sub extends Super { def avslöjad = minHemlis }
- Error: Not found: minHemlis

scala> class Sub extends Super { def avslöjad = vårHemlis }

scala> val s = Sub()
val s: Sub = Sub@2eee9593

scala> s.avslöjad
val res0: Int = 42

scala> s.minHemlis
-- Error:
value minHemlis is not a member of Sub - did you mean s.vårHemlis?

scala> s.vårHemlis
-- Error:
Access to protected value vårHemlis not permitted because enclosing object
is not a subclass of trait Super where target is defined
```

### 10.1.24 Filnamnsregler och -konventioner

- I flera språk, t.ex. Java, gäller dessa regler (men **inte** i Scala):
  - Det går bara ha **en enda** publik klass per kodfil.
  - Kodfilen måste ha **samma namn** som den publika klassen, t.ex. `KlassensNamn.java`
- Scala är friare:
  - I Scala får man ha **många** klasser/traits/singelobjekt i samma kodfil.
  - I Scala får man döpa kodfilerna **oberoende** av deras innehåll.
- Dessa **konventioner** brukar användas i Scala:
  - Om en kodfil bara innehåller **en enda** klass/trait/singelobjekt ge filen samma namn som innehållet, t.ex. `KlassensNamn.scala`
  - Om en kodfil innehåller **flera** saker, döp filen till något som återspeglar hela innehållet och använd **liten begynnelsebokstav**, t.ex. `drawing-utils.scala` eller `bastypensNamn.scala`

### 10.1.25 Klasser, arv och klassparametrar

Klasser kan ärva andra typer (klasser och traits). Om supertypen har parametrar så **måste** subtypen ge argument efter **extends**.

```
1 object personExample1:
2
3   class Person(val namn: String)
4
5   class Akademiker(namn: String,
6                     val universitet: String) extends Person(namn)
```

```

7
8  class Student(namn: String,
9              universitet: String,
10             val program: String) extends Akademiker(namn, universitet)
11
12  class Forskare(namn: String,
13              universitet: String,
14             val titel: String) extends Akademiker(namn, universitet)
15
16  def main(args: Array[String]): Unit =
17      val kim = new Student("Kim Robinson", "Lund", "Data")
18      println(s"${kim.namn} ${kim.universitet} ${kim.program}")

```

### 10.1.26 Statisk och dynamisk typ

```
var p: Person = Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är Person vilket gör att vi sedan kan låta p referera till andra instanser som är av typen Person.

```
p = Student("Kim Robinson", "Lund", "Data")
```

- Med ”statisk typ” menas den typinformation som kompilatorn känner till vid kompileringstid.
- Den **dynamiska typen**, även kallad **körtidstypen**, som gäller under körning är här mer specifik och mångfaceterad: p är efter tilldelning nu Student, Person och Akademiker.
- Man kan undersöka om den dynamiska typen för p är EnVissTyp med `p.isInstanceOf[EnVissTyp]`
- Man kan säga åt kompilatorn: **”jag garanterar att p är av typen EnVissTyp så du kan omforma den statiska typen till EnVissTyp och så får jag stå ut med körtidsfel om jag ljugar”** genom att göra typomvandling (eng. *type casting*) med `p.asInstanceOf[EnVissTyp]` (detta är mycket ovanligt i normal Scala-kod)

### 10.1.27 Inmixning

Man kan ärva flera traits. Detta kallas **inmixning** (eng. *mix-in*) och görs med en komma-separerad typ-lista efter **extends**.

```

1  object personExample2:
2
3      trait Person      { val namn: String }
4      trait Akademiker { val universitet: String }
5      trait Examinerad { val titel: String }
6
7      class Student(val namn: String,
8                  val universitet: String,
9                  val program: String) extends Person, Akademiker
10

```

```

11  class Forskare(val namn: String,
12                val universitet: String,
13                val titel: String) extends Person, Akademiker, Examinerad
14
15  def main(args: Array[String]): Unit =
16    val f = new Forskare("B. Regnell", "Lunds universitet", "Professor Dr")
17    println(s"${f.titel} ${f.namn}, ${f.universitet}")

```

### 10.1.28 isInstanceOf och asInstanceOf

Testa körtidstyp med `isInstanceOf[Typ]`. Lova kompilatorn (och ta själv ansvar för) att det är en viss körtidstyp med `asInstanceOf[Typ]`. OBS! Använd hellre **match**.

```

1  object personExample3:
2
3    trait Person { val namn: String }
4    trait Akademiker { val universitet: String }
5    trait Examinerad { val titel: String }
6
7    class Student(val namn: String,
8                  val universitet: String,
9                  val program: String) extends Person, Akademiker
10
11    class Forskare(val namn: String,
12                  val universitet: String,
13                  val titel: String) extends Person, Akademiker, Examinerad
14
15    def main(args: Array[String]): Unit =
16      var p: Person = new Forskare("B. Regnell", "Lunds universitet", "Professor Dr")
17      if p.isInstanceOf[Akademiker] then println(p.namn + " är akademiker")
18      p = new Student("Kim Robinson", "Lund", "Data") // går det att göra p.program?
19      if p.isInstanceOf[Student] then println(p.asInstanceOf[Student].program)
20      // Ovan görs hellre med match

```

### 10.1.29 Terminologi: Polymorfism och dynamisk bindning

```

trait Robot { def work(): Unit }

case class CleaningBot(name: String) extends Robot:
  override def work(): Unit = println(" Städa Städa")

case class TalkingBot(name: String) extends Robot:
  override def work(): Unit = println(" Prata Prata")

```

**Polymorfism** betyder ”många former”. Referenserna `r` och `bot` nedan kan ha olika ”former”, d.v.s de kan referera till olika sorters robotar.

**Dynamisk bindning** innebär att körtidstypen avgör vilken metod som körs.

```

1  scala> def robotDoWork(bot: Robot) = { print(bot); bot.work() }
2
3  scala> var r: Robot = new CleaningBot("Wall-E")

```

```

4
5 scala> robotDoWork(r)
6 CleaningBot(Wall-E) Städa Städa
7
8 scala> r = new TalkingBot("C3PO")
9
10 scala> robotDoWork(r)
11 TalkingBot(C3PO) Prata Prata

```

### 10.1.30 Anonym klass

Om man har en abstrakt typ med saknade implementationer kan man fylla i det som fattas i dessa i ett extra block som ”hängs på” vid instansiering:

```

1 scala> trait Grönsak { val vikt: Int }
2 // defined trait Grönsak
3
4 scala> new Grönsak
5 -- Error:
6 1 |new Grönsak
7   |   ^^^^^^^
8   |   Grönsak is a trait; it cannot be instantiated
9
10 scala> new Grönsak { val vikt = 42 }
11 val res0: Grönsak = anon1@4e3f2908

```

Man får då vad som kallas en **anonym klass**. (I detta fall en ganska konstig grönsak som inte är någon speciell sorts grönsak, men som ändå har en vikt.)

Den allra enklaste (och mest meningslösa) anonyma klassen är:

```

scala> new {}
val res0: Object = anon1@5bb37371

```

### 10.1.31 Hur förhindra subtypning?

Du kan förhindra subtypning på dessa sätt:

- Med **final** skapar du en final klass som ej går att ärva:

```
final class Person(name: String)
```

```

scala> object Björn extends Person("Björn")
-- Error:
1 |object Björn extends Person("Björn")
  |      ^
  |      object Björn cannot extend final class Person

```

- Med **sealed** kan du försegla en hel typhierarki:

```

scala> sealed trait T // tryck Alt+TAB för att vänta med evaluering
  | final class A extends T
  | final class B extends T

```

```
|
scala> new T{}
-- Error:
1 |new T{}
  |^
  |Cannot extend sealed trait T in a different source file
```

Med en förseglad typhierarki kan du bara ärva bastypen inom samma kodfil.

### 10.1.32 Förseglade typer med sealed

Med en **sealed** kan du skapa en **förseglad** uppräknings:

```
sealed trait Färg(val toInt: Int)
object Färg:
  val values = Vector(Spader, Hjärter, Ruter, Klöver)

  case object Spader extends Färg(0)
  case object Hjärter extends Färg(1)
  case object Ruter   extends Färg(2)
  case object Klöver  extends Färg(3)
```

Nyckelordet **sealed förhindrar** vidare subtypning av Färg i **annan kodfil** och **ger varning** om matchning är ofullständig – tips för att undvika körtidsfel.

```
scala> Färg.values(0) match { case Färg.Spader => "hej" }
-- Warning:
1 |Färg.values(0) match { case Färg.Spader => "hej" }
  |^^^^^^^^^^^^^^^^
  |match may not be exhaustive.
  |
  |It would fail on pattern case: Hjärter, Ruter, Klöver
val res0: String = hej
```

### 10.1.33 Öppen klass signalerar uppmuntrad subtypning

- Om man ärver en klass får man tillgång till alla medlemmar som inte är privata och kan byta till godtycklig implementation om typerna stämmer.
- Detta kan vara riskabelt om den som skrivit klassen inte planerat för detta och noga dokumenterat hur klassen är tänkt att användas vid arv.
- Genom att skapa **öppna klasser** med nyckelordet **open** signalerar du att klassen är tänkt att vara en supertyp vid arv.

```
open class Gurka(val vikt: Int, val pris: Double):
  /** Gör override på denna om du vill ha annat alternativ. */
  def alternativ: String = s"Det går precis lika bra med selleri!"
```

```
scala> object StorGurkan extends Gurka(1000, 1_000_000): // ärv på bäst du vill!
  override def alternativ = "kan kanske också funka med en betongpelare"
```

- **open** krävs om du vill tysta varning vid arv från en annan kodfil.

- Du kan även komma undan varningen med **import** `scala.language.adhocExtensions`  
<https://youtu.be/aFmIS5qeetA?t=221>

### 10.1.34 Trait eller abstrakt klass?

Nyckelordet **abstract** behövs framför **class** om abstrakta medlemmar:

```
scala> class X { val x: Int }
1 |class X { val x: Int }
  |      ^
  |      class X needs to be abstract, since val x: Int in class X is not defined

scala> abstract class X { val x: Int } // fungerar!
```

Men går det inte lika bra med en trait? Det går ofta [precis lika bra med en trait](#).

Använd en **trait** om...

- ...du är osäker på vilket som är bäst. (Du kan alltid ändra till en abstrakt klass senare.)
- ...du vill kunna mixa in din trait tillsammans med andra traits.
- ...du vill göra din trait, om inmixad, transparent vid typhärledning.

Använd en **abstrakt klass** om...

- ...du vill begränsa inmixning.
- ...du vill ärva supertypen från klasser skrivna i Java.
- ...du vill minimera tid för omkompilering vid ändringar (spar tid vid stora projekt).
- ...du vill vidarebefordra klassparametrar till supertyp.

### 10.1.35 En trait får ej vidarebefordra parametrar

En trait får inte skicka vidare parametrar till en supertyp (det skulle bli knepiga problem annars vid inmixning):

```
scala> trait X(x: Int)
// defined trait X

scala> trait Y(y: Int) extends X(y)
-- Error:
1 |trait Y(y: Int) extends X(y)
  |                      ^^^^
  | trait Y may not call constructor of trait X
```

Men det funkar fint med en **abstract class** eller en **class** (som ju **inte** får vara inmixningar):

```
scala> abstract class Y(y: Int) extends X(y)
// defined class Y
```

Mer detaljer här: [dotty.epfl.ch/docs/reference/other-new-features/trait-parameters.html](http://dotty.epfl.ch/docs/reference/other-new-features/trait-parameters.html)

### 10.1.36 Medlemmar, arv och överskuggning

Olika sorters överskuggningsbara medlemmar i **Scala**:

- **def**
- **val**
- **lazy val**
- **var**

- När man överskuggar (eng. *override*) en medlemmen med en annan medlem med samma namn i en subtyp, får denna medlem en (ny) implementation.
- Singelobjekt kan ej ärvas (och medlemmar i singelobjekt kan därmed ej överskuggas).
- När man konstruerar ett objektorienterat språk gäller det att man definierar sunda överskuggningsregler vid arv. Detta är förvånansvärt knepigt.
- Alla knepiga regler för överskuggning är svåra att lära sig utantill, men som tur är så hjälper kompilatorns felmeddelande dig att följa reglerna :)

Olika sorters överskuggningsbara instansmedlemmar i **Java**:

- variabel
- metod

Medlemmar som är **static** kan ej överskuggas (men döljas) vid arv.

### 10.1.37 Fördjupning: Regler för överskuggning i Scala

En medlem M1 i en supertyp får överskuggas av en medlem M2 i en subtyp, enligt dessa regler: (se även <https://stackoverflow.com/questions/40057337>)

1. M1 och M2 ska ha samma namn och typerna ska matcha.
2. **def** får bytas ut mot: **def**, **val**, och **lazy val**, och under speciella förutsättningar **var** (mer om att byta **def** mot **var** snart)
3. **val** får bytas ut mot **val**. Om medlemmen i M1 som överskuggas är abstrakt så får en **val** även bytas mot en **lazy val**.
4. En abstrakt **var** får bara bytas ut mot en **var**. En konkret **var** får ej bytas ut.
5. **lazy val** får bara bytas ut mot en **lazy val**.
6. Om en medlem i en supertyp är abstrakt *behöver* man inte använda nyckelordet **override** i subtypen. (Men det är bra att göra det ändå så att kompilatorn hjälper dig att kolla att du verkligen överskuggar något.)
7. Om en medlem i en supertyp är konkret *måste* man använda nyckelordet **override** i subtypen, annars ges kompileringsfel.
8. M1 får inte vara **final**.
9. M1 får inte vara **private**, men kan vara **private**[X] om M2 också är **private**[X], eller **private**[Y] om X är (en del av) Y.
10. Om M1 är **protected** måste även M2 vara det.

### 10.1.38 Fördjupning: Överskugga var med var

Det krävs speciella förutsättningar för att överskugga en **var** med en **var**.

- En konkret medlem får **inte** bytas ut mot en **var** – inte ens en konkret **var**:

```
scala> trait ConcreteVar { var x = 42 }
scala> trait Sub extends ConcreteVar { override var x = 43 }
-- Error:
```



```
1 | trait Sub extends ConcreteVar { override var x = 43 }
  |                               ^
  |                               error overriding variable x in trait ConcreteVar of type Int;
  |                               variable x of type Int cannot override a mutable variable
```

- En abstrakt **var**-medlem får bytas ut mot en **var** om du **inte** skriver **override**:

```
scala> trait AbstractVar { var x: Int }
scala> trait Sub extends AbstractVar { override var x = 43 }
-- Error:
1 | trait Sub extends AbstractVar { override var x = 43 }
  |                               ^
  |                               error overriding variable x in trait AbstractVar of type Int;
  |                               variable x of type Int cannot override a mutable variable
scala> trait Sub extends AbstractVar { var x = 43 } // funkar utan override
// defined trait Sub
```

Undvik abstrakta **var**-medlemmar – oftast bättre med abstrakt **def**.

### 10.1.39 Fördjupning: Överskugga def med var

- En abstrakt **def**-medlem får bytas ut mot en **var** om du **inte** skriver **override**:

```
scala> trait AbstractDef { def x: Int }
scala> trait Sub extends AbstractDef { override var x = 43 }
-- Error:
1 | trait Sub extends AbstractDef { override var x = 43 }
  |                               ^
  |                               setter x_= overrides nothing
scala> trait Sub extends AbstractVar { var x = 43 } // funkar om ej override
```

Den abstrakta **def**-medlemmen blir då implementerad av en konkret getter.

- Egentligen är en publik **var**-medlem en kombination av en getter och en setter. Du kan skapa konkret getter+setter och överskugga gettern explicit med **override** (notera att settern inte kan göra **override**, eftersom superklassen inte har någon motsvarande metod att byta ut – jämför felmeddelande ovan):

```
scala> trait Sub2 extends AbstractDef:
  private var myPrivateValue = 42
  override def x: Int = myPrivateValue
  def x_(newValue: Int): Unit = myPrivateValue = newValue
```

Ovan fungerar fint eftersom vi nu har en giltig kombination av getter+setter.

### 10.1.40 Att skilja på mitt och ditt med super

```
1 | scala> class X { def gurka = "super pepino" }
2 |
3 | scala> class Y extends X {
4 |     override val gurka = ":(("
5 |     val sg = super.gurka
6 | }
7 |
8 | scala> val y = new Y
9 | y: Y = Y@26ba2a48
10 |
```

```
11 scala> y.gurka
12 res0: String = :(
```

Super Pepinos to the rescue:

```
scala> y.sg
res1: String = super pepino
```



<https://youtu.be/NPhjiXskz34>

### 10.1.41 Fördjupning: Intersektionstyp

Vid inmixning blir supertypen en intersektionstyp (eng. *intersection type*), vilket indikeras med typoperatoren & i exempel nedan:

```
trait Grönsak(val vikt: Int)
trait HarSmak(val smak: String)

object Gurka extends Grönsak(42), HarSmak("vattning")
object Tomat extends Grönsak(43), HarSmak("syrlig")
```

```
scala> Vector(Gurka, Tomat)
val res0: Vector[Grönsak & HarSmak] =
  Vector(Gurka@560742f7, Tomat@3cc82e23)
```

Det går att skapa egna intersektionstyper med **with**:

```
scala> class X { val x = 42 }
scala> trait Y { val y = "hej" }
scala> val a: X & Y = new X           // -- Error: Found: X Required: X & Y
scala> val b: X & Y = new X with Y    // Funkar! En anonym klass av typen X & Y
```

### 10.1.42 Fördjupning: Typunioner med eller-operator

En mycket enkel summa-typ kan skapas med typoperatoren |

```
scala> var x: Int | String = 42
var x: Int | String = 42

scala> x = "hej"
x: Int | String = hej

scala> type IntOrErr = Int | String
// defined alias type IntOrErr = Int | String

scala> def div(nom: Int, denom: Int): IntOrErr =
  if denom != 0 then nom / denom else "div. by zero"
```

Fördel jämfört med klass: rudimentärt enkelt.

Nackdelar: kan inte deklarerera parametrar, medelemtar och allt annat som en klass kan; i viss mån kan detta kompenseras med **extension** och **match**.

Läs mer här: <https://dotty.epfl.ch/docs/reference/new-types/union-types.html>

### 10.1.43 Fördjupning: Transparent trait

Du kan göra din trait **genomskinlig** vid typhärledning av supertypen för inmixningen med nyckelordet **transparent**:

```
trait Grönsak(val vikt: Int)
transparent trait HarSmak(val smak: String)

object Gurka extends Grönsak(42), HarSmak("vattning")
object Tomat extends Grönsak(43), HarSmak("syrlig")
```

```
scala> Vector(Gurka, Tomat)
val res0: Vector[Grönsak] =
  Vector(Gurka@560742f7, Tomat@3cc82e23)
```

Vilken typ hade visats utan **transparent**?  
Vector[Grönsak & HarSmak]

---

### 10.1.44 Terminologi och nyckelord vid arv

<b>subtyp</b>	en typ som ärver en supertyp
<b>supertyp</b>	en typ som ärvs av en subtyp
<b>bastyp</b>	en typ som är rot i ett arvsträd
<b>abstrakt medlem</b>	en medlem som saknar implementation
<b>konkret medlem</b>	en medlem som ej saknar implementation
<b>abstrakt typ</b>	en typ som kan ha abstrakta medlemmar; kan ej instansieras
<b>konkret typ</b>	en typ som ej har abstrakta medlemmar; kan instansieras
<b>class</b>	en konkret typ som <b>kan ej ha abstrakta medlemmar</b>
<b>abstract class</b>	en abstrakt typ som <b>kan ha abstrakta medlemmar</b>
<b>trait</b>	är en abstrakt typ som <b>kan mixas in</b>
<b>extends</b>	står före en supertyp, medför arv av supertypens medlemmar
<b>override</b>	en medlem överskuggar (byter ut) en medlem i en supertyp
<b>protected</b>	gör en medlem synlig i subtyper till denna typ (jmf <b>private</b> )
<b>final</b> gurka	gör medlemmen gurka final: förhindrar överskuggning
<b>final class</b>	gör klassen final: förhindrar vidare subtypning
<b>sealed</b>	förseglad trait/klass: enbart subtyper i denna kodfil, koll av match
<b>open class</b>	berätta att den är tänkt att ärvas, <b>open</b> krävs för arv i annan kodfil
<b>transparent trait</b>	gör typen osynlig vid typhärledning
<b>super</b> .gurka	refererar till supertypens medlem gurka (jmf <b>this</b> )

### 10.1.45 Vad är en algebraisk datatyp?

Algebraisk datatyp (eng. *algebraic data type*), förk. ADT:

- En datatyp som består av (en kombination av):
- **Produkt**-typer, **Summa**-typer:
  - En produkt-typ är en ”och”-typ (ä.k. record, struct), exempel:

**case class** Person(namn: String, ålder: Int)

består av attributen namn **OCH** ålder.

- En summa-typ är en 'Eller'-typ (ä.k. disjunkt union, co-produkt).
- Exempel: **enum** Färg { **case** Röd, Svart}  
Färg kan vara antingen Röd **ELLER** Svart.  
En Grönsak är antingen en Gurka **ELLER** en Tomat

```
sealed trait Grönsak { val vikt: Int }
case class Gurka(vikt: Int) extends Grönsak
case class Tomat(vikt: Int) extends Grönsak
```

[https://en.wikipedia.org/wiki/Algebraic\\_data\\_type](https://en.wikipedia.org/wiki/Algebraic_data_type)

#### 10.1.46 En case-klass är en produkt.

Klassen Person nedan har ett namn **och** en ålder.

```
1 scala> case class Person(namn: String, ålder: Int)
2
3 scala> Person("Kim",42)
4 val res0: Person = Person(Kim,42)
5
6 scala> res0.isInstanceOf[Product]
7 val res1: Boolean = true
8
9 scala> res0.product      // Tryck TAB
10 productArity            productElement    productElementName
11 productElementNames     productIterator   productPrefix
12
13 scala> res0.productElementNames.toVector
14 val res2: Vector[String] = Vector(namn, ålder)
15
16 scala> res0.productElement
17 val res3: Int => Any = Lambda$1981/0x00000008408f2840@44498af0
18
19 scala> res0.productElement(0)
20 val res4: Any = björn
```

#### 10.1.47 Algebraisk datatyp, kombinerad produkt och summa

Med trait, case-klass och objekt:

```
sealed trait PersonId // summan av två subtyper (Number eller Missing)
object PersonId:
  case class Number(n: Long) extends PersonId // produkt med ett element
  case object Missing extends PersonId        // ett enkelt värde
```

Motsvarande med **enum**:

```
enum PersonId:
  case Number(n: Long)
  case Missing
```

---

### 10.1.48 Algebraisk datatyp med typparameter

Denna ADT har ett fall som är en **generisk** case-klass:

```
sealed trait Kanske[+A]    // +A ger s.k. covarians, mer om det senare
object Kanske:
  case class Någon[A](a: A) extends Kanske[A]
  case object Ingen extends Kanske[Nothing]
```

Motsvarande med **enum**: (kompilatorn fyller i **extends** ... automatiskt)

```
enum Kanske[+A]:
  case Någon(a: A)
  case Ingen
```

Känner du igen denna? Jämför inbyggda typen Option

Implementation av getOrElse med extensionsmetod, tips: använd **match**

```
extension [A](k: Kanske[A]) def getOrElse(default: A):A = k match
  case Kanske.Någon(a) => a
  case Kanske.Ingen => default
```

---

## 10.2 Övning inheritance

### Mål

- ☐ Kunna deklarerera och använda en arvshierarki i flera nivåer med nyckelordet **extends**.
- ☐ Känna till synlighetsregler vid arv och nyttan med privata och skyddade medlemmar och nyckelorden **private** och **protected**.
- ☐ Kunna deklarerera och använda överskuggade medlemmar och nyckelordet **override**.
- ☐ Kunna deklarerera och använda en hierarki av klasser där konstruktorparametrar överförs till superklasser.
- ☐ Kunna deklarerera och använda uppräknade värden med case-objekt och gemensam bas typ.
- ☐ Känna till reglerna som gäller vid överskuggning av olika sorters medlemmar.
- ☐ Känna till nyttan med finala klasser och finala attribut och nyckelordet **final**.
- ☐ Kännedom om dessa begrepp: bas typ, supertyp, subtyp, körtidstyp, dynamisk bindning, polymorfism, trait, inmixning, överskuggad medlem, anonym klass, skyddad medlem, abstrakt medlem, abstrakt klass, referenstyp, värdetyp.

### Förberedelser

- ☐ Studera begreppen i kapitel 10

### 10.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** Para ihop begrepp med beskrivning.

Koppla varje begrepp med den (förenklade) beskrivning som passar bäst:

bas typ	1	A	har supertypen AnyRef, allokeras i heapen via referens
supertyp	2	B	kan ha många former, t.ex. en av flera subtyper
subtyp	3	C	klass utan namn, utvidgad med extra implementation
körtidstyp	4	D	en typ som är mer specifik
dynamisk bindning	5	E	kan ha parametrar, kan ej instansieras, kan ej mixas in
polymorfism	6	F	saknar implementation
trait	7	G	har supertypen AnyVal, lagras direkt på stacken
inmixning	8	H	tillföra egenskaper med <b>with</b> och en trait
överskuggad medlem	9	I	är abstrakt, kan mixas in, kan ha parametrar
anonym klass	10	J	kan vara mer specifik än den statiska typen
skyddad medlem	11	K	är endast synlig i subtyper
abstrakt medlem	12	L	körtidstypen avgör vilken metod som körs
abstrakt klass	13	M	medlem i subtyp ersätter medlem i supertyp
förseglad typ	14	N	den mest generella typen i en arvshierarki
referenstyp	15	O	en typ som är mer generell
värdetyp	16	P	subtypning utanför denna kodfil är förhindrad

**Uppgift 2. Gemensam bas typ.** Man vill ofta lägga in objekt av olika typ i samma samling.

```
1 scala> class Gurka(val vikt: Int)
2 scala> class Tomat(val vikt: Int)
3 scala> val gurkor = Vector(Gurka(100), Gurka(200))
4 scala> val grönsaker = Vector(Gurka(300), Tomat(42))
```

a) Om en samling innehåller objekt av flera olika typer försöker kompilatorn härleda den mest specifika typen som objekten har gemensamt. Vad blir det för typ på värdet grönsaker ovan?

b) Försök ta reda på summan av vikterna enligt nedan. Vad ger andra raden för felmeddelande? Varför?

```
1 scala> gurkor.map(_.vikt).sum // fungerar
2 scala> grönsaker.map(_.vikt).sum // fungerar inte
```

c) Du ska nu göra så att du kan komma åt vikten på alla grönsaker genom att ge gurkor och tomater en gemensam bas typ som de olika konkreta grönsakstyperna utvidgar med nyckelordet **extends**. Det heter att subtyperna Gurka och Tomat **ärver** egenskaperna hos supertypen Grönsak.

Skapa en bas typ Grönsak med ett abstrakt attribut vikt. Låt sedan de konkreta grönsakerna ärva bas typen:

```
1 scala> trait Grönsak { val vikt: Int }
2 scala> class Gurka(val vikt: Int) extends Grönsak
3 scala> class Tomat(val vikt: Int) extends Grönsak
4 scala> val gurkor = Vector(Gurka(100), Gurka(200))
5 scala> val grönsaker = Vector(Gurka(300), Tomat(42))
```

När sker initialisering av attributet vikt?

d) Vad blir det nu för typ på variabeln grönsaker ovan?

e) Går det nu att summera av vikterna i grönsaker med uttrycket nedan? Varför?  
grönsaker.map(\_.vikt).sum

f) En trait liknar en klass, men man kan inte instansiera den direkt. Vad blir det för felmeddelande om du försöker skapa en instans av en trait enligt nedan?

```
1 scala> trait Grönsak { val vikt: Int }
2 scala> new Grönsak
```

g) Traiten Grönsak har en abstrakt medlem vikt. Den sägs vara abstrakt eftersom den saknar definition – medlemmen har bara ett namn och en typ men inget värde. Du kan instansiera den abstrakta traiten Grönsak om du fyller i det som "fattas", nämligen ett värde på vikt. Man kan fylla på det som fattas i genom att "hänga på" ett block efter typens namn vid instansiering. Man får då vad som kallas en **anonym klass**, i detta fall en ganska konstig grönsak som inte är någon speciell sorts grönsak med som ändå har en vikt.

Vad får anonymGrönsak nedan för typ och strängrepresentation?

```
1 scala> val anonymGrönsak = new Grönsak { val vikt = 42 }
```

h) Vad blir felmeddelandet om du skapar en anonym klass Grönsak med en kropp som saknar definition av vikt?



**Uppgift 3.** *Polymorfism vid arv, s.k. subtypspolymorfism.* Polymorfism betyder ”många skepnader”. I samband med arv innebär det att flera subtyper, till exempel Ko och Gris, kan hanteras gemensamt som om de vore instanser av samma supertyp, så som Djur. Subklasser kan implementera en metod med samma namn på olika sätt. Vilken metod som exekveras bestäms vid körtid beroende på vilken subtyp som instansieras. På så sätt kan djur komma i många skepnader.

a) Implementera funktionen skapaDjur nedan så att den returnerar antingen en ny Ko eller en ny Gris med lika sannolikhet.

```
1 scala> trait Djur { def väsnas: Unit }
2 scala> class Ko extends Djur { def väsnas = println("Muuuuuuu") }
3 scala> class Gris extends Djur { def väsnas = println("Nöffnöff") }
4 scala> def skapaDjur(): Djur = ???
5 scala> val bondgård = Vector.fill(42)(skapaDjur())
6 scala> bondgård.foreach(_ .väsnas)
```

b) Lägg till ett djur av typen Häst som väsnas på lämpligt sätt och modifiera skapaDjur så att det skapas kor, grisar och hästar med lika sannolikhet.

**Uppgift 4.** *Olika typer av heltalspar till laborationen [snake0](#).* Under veckans laboration ska du använda olika typer av par som representerar riktning och position på en tvådimensionell spelplan, samt spelplanens storlek. I stället för att använda en vanlig 2-tupel till dessa tre olika typer av par ska du skapa egna, specifika typer som alla ärver basypen Pair[T]. Dessa typer ska alla ligga i filen pairs.scala i **package** snake.

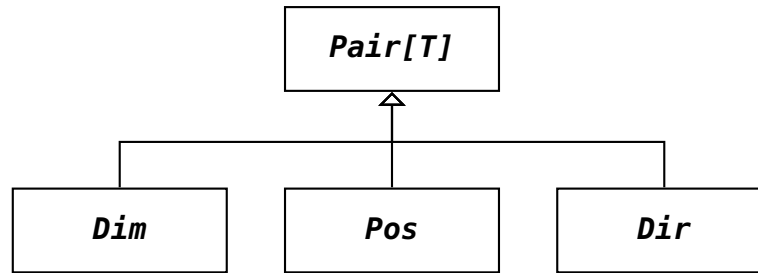
```
// detta är en skiss på filen pairs.scala
package snake

trait Pair[T]:
  def x: T
  def y: T
  // uppgift a) lägg till den konkreta metoden tuple

// efterföljande deluppgifterna implementerar dessa subtyper till Pair:
//   case klass Dim beskriver en 2-dimensionell ytas storlek
//   case klass Pos beskriver en position på en yta av Dim storlek
//   enum Dir beskriver förflyttning mot North, South, East, West
```

Skillnaden mellan Pair[T] och en vanlig 2-tupel är att medlemmarna x och y garanterat är av *samma* typ, medan en 2-tupel kan innehålla element av olika typ.

I fig. 10.1 visas en bild av klasshierarkin som du steg-för-steg ska utveckla i efterföljande uppgifter. Fördelen med att ha olika typer av par är att det är mer typsäkert (eng. *type safe*): vi får hjälp av kompilatorn att upptäcka om vi av misstag förväxlar t.ex. en position med en riktning.



**Figur 10.1:** Arvshierarki med Pair[T] som bastyp.

a) Öppna en editor och koda **trait** Pair[T] i en fil pairs.scala. Lägg dessutom till en konkret metod tuple i Pair[T] som returnerar en 2-tupel med de båda elementen i paret, så att det vid behov går att omvandla Pair-instanser till 2-tupler. Använd REPL via sbt console för att testa att detta fungerar:

```
scala> val p = new Pair[Int] { override val x = 10; override val y = 20 }
p: Pair[Int]{val x: Int; val y: Int} = $anon$1@784223e9

scala> p.tuple
val res0: (Int, Int) = (10,20)
```

b) Skapa en case-klass Dim som ärver Pair[Int]. Instanser av denna klass kommer du att använda under veckans laboration för att representera en spelplans storlek genom att låta x ange antalet horisontella positioner och y antalet vertikala positioner.

Lägg även till ett kompanjonsobjekt Dim med en apply-metod som kan skapa Dim-instanser givet en 2-tupel. Testa i REPL enligt nedan.

```
scala> Dim(50, 60)
val res1: Dim = Dim(50,60)

scala> Dim((60, 50))
val res2: Dim = Dim(60,50)

scala> res2.tuple
val res3: (Int, Int) = (60,50)
```

c) Lägg till en case-klass Pos som ärver Pair[Int] som representerar en position med en x-koordinat och en y-koordinat, båda klassparametrar. Koordinaterna ska hållas inom en spelplansstorlek som ges av klassparametern dim av typen Dim. Koordinatpositionerna är heltal och räknas från 0 till (men inte med) dim.x resp. dim.y.

Gör primärkonstruktorn i case-klassen Pos **privat**, genom att skriva nyckelordet **private** efter klassnamnet men före klassparameterlistan, så att det inte går att skapa instanser via primärkonstruktorn utanför klasskroppen och kompanjonsobjektet.

Implementera metoderna + och - i case-klassen Pos. Båda metoderna ska ta en parameter p av typen Pair[Int] och returnera en ny Pos, där p.x resp. p.y är adderat resp. subtraherat från aktuell position. Observera att du inte ska skriva **new** när du skapar en ny instans, eftersom dessa alltid ska skapas via kompanjonsobjektets apply-metod, som är en "smart" fabriksmetod som garanterar håller koordinaterna inom spelplanen.

Lägg till ett kompanjonsobjekt Pos med en apply-metod som skapar en ny Pos-instans som ser till att koordinaterna alltid är inom dim. Aritmetiken ska ske modulo storleken dim, d.v.s en position ska aldrig kunna hamna utanför spelplanen; i stället så börjar man om på andra sidan (se exempel i REPL nedan).

*Tips:* Använd `java.lang.Math.floorMod` som hanterar negativa argument så att resultatet blir positivt (till skillnad från modulo-operatören `%`).

Lägg även till fabriksmetoden `random` som kan skapa nya slumpmässiga positioner inom `dim`. *Tips:* Använd `scala.util.Random.nextInt`.

Testa att det fungerar enligt nedan:

```
scala> Pos(-1,20,Dim(10,20))
val res4: Pos = Pos(9,0,Dim(10,20))

scala> new Pos(-1,20,Dim(10,20)) // förbjuds med privat primärkonstruktor
-- Error:
1 |new Pos(-1,20,Dim(10,20))
  |    ^^^
  |constructor Pos cannot be accessed as a member of Pos

scala> Pos(0,0,Dim(5,5)) + Pos(6,12, Dim(5,5))
val res5: Pos = Pos(1,2,Dim(5,5))

scala> Pos(0,0,Dim(5,5)) - Pos(1,2, Dim(5,5))
val res6: Pos = Pos(4,3,Dim(5,5))

scala> for (_ <- 1 to 3) yield Pos.random(Dim(10,10))
val res7: IndexedSeq[Pos] =
  Vector(Pos(8,8,Dim(10,10)), Pos(2,6,Dim(10,10)), Pos(3,7,Dim(10,10)))
```

d) Vad händer om du glömmer skriva **new** när du anropar den privata konstruktorn i din `apply`-metod? Varför finns inte detta problem i `apply`-metoden för `Dim`?

e) Lägg till en **enum** `Dir` som ärver `Pair[Int]` och har två **val**-parametrar `x` och `y`. Lägg också till fyra fall med **case** som alla ärver `Dir` och som representerar en enstegsförflyttning i de fyra väderstrecken, genom att ge parametrarna `x` resp. `y` något av värden 1, -1 eller 0. Norrut ska anges med `x`-koordinaten 0 och `y`-koordinaten -1, etc. Verifiera i REPL att enumerationen fungerar.

Lägg till en **export** som gör så att det räcker att importera `snake.*` för att få alla fyra riktningar synliga direkt (annars behövs även `import Dir.*` på alla ställen där riktning används i och utanför paketet `snake`)

**Uppgift 5.** *Supertyp med parameter.* Utbildningsdepartementet vill med sitt nya datasystem hålla koll på vissa personer och skapar därför en klasshierarki enligt nedan. Skriv in koden i en editor och testa i REPL med `sbt`.

```
class Person(val namn: String)

class Akademiker(
  namn: String,
  val universitet: String) extends Person(namn)

class Student(
  namn: String,
  universitet: String,
  program: String) extends Akademiker(namn, universitet)

class Forskare(
  namn: String,
  universitet: String,
```

```
titel: String) extends Akademiker(namn, universitet)
```

- Deklarera fyra olika **val**-variabler med lämpliga namn som refererar till olika instanser av alla olika klasser ovan och ge attributen valfria initialvärden genom olika parametrar till konstruktörerna.
- Skriv två satser: en som först stoppar in instanserna i en Vector och en som sedan loopar igenom vektorn och skriv ut alla instansers toString och namn.
- Utbildningsdepartementet vill att det inte ska gå att instansiera objekt av typerna Person och Akademiker. Det kan åstadkommas genom att placera nyckelordet **abstract** före **class**. Uppdatera koden i enlighet med detta. Vilket blir felmeddelande om man försöker instansiera en **abstract class**? Går det lika bra med en **trait**?
- Utbildningsdepartementet vill slippa implementera toString. Gör därför om typerna Student och Forskare till case-klasser. *Tips:* För att undkomma ett kompileringsfel (vilket?) behöver du använda **override val** på lämpligt ställe. Skapa instanser av de nya case-klasserna Student och Forskare och skriv ut deras toString.
- Använd abstrakta attribut i stället för parametrar för typerna som är abstrakta, så att du inte behöver skriva **override val** i klassparametrarna till de konkreta case-klasserna. Du ska också införa en case-klass IckeAkademiker som ska användas i olika statistiska medborgarundersökningar. Dessutom förser man alla personer med ett personnummer representerat som en Long. Hur ser utbildningsdepartementets kod ut nu, efter alla ändringar? Skriv ett testprogram som skapar några instanser och skriver ut deras attribut.

### 10.2.2 Extrauppgifter; träna mer

**Uppgift 6.** *Bastypen Shape och subtyperna Rectangle och Circle.* Du ska i denna uppgift skapa ett litet bibliotek för geometriska former med oföränderliga objekt implementerade med hjälp av case-klasser. De geometriska formerna har en gemensam bastyp kallad Shape. Utgå från koden nedan.

```
case class Point(x: Double, y: Double):
  def move(dx: Double, dy: Double): Point = Point(x + dx, y + dy)

trait Shape:
  def pos: Point
  def move(dx: Double, dy: Double): Shape

case class Rectangle(pos: Point, width: Double, height: Double) extends Shape:
  def move(dx: Double, dy: Double): Rectangle = copy(pos = pos.move(dx, dy))

case class Circle(pos: Point, radius: Double) extends Shape:
  def move(dx: Double, dy: Double): Circle = copy(pos = pos.move(dx, dy))
```

- Instansiera några cirklar och rektanglar och gör några relativa förflyttningar av dina instanser genom att anropa move.
- Lägg till en konkret metod moveTo i Point som gör en absolut förflyttning till koordinaterna x och y. Lägg till en abstrakt metod moveTo Shape som implementeras i subklasserna. Testa med REPL på några instanser av Rectangle och Circle.
- Lägg till metoden distanceTo(that: Point): Double i case-klassen Point som räknar ut avståndet till en annan punkt med hjälp av math.hypot. Klistra in i REPL och testa på några instanser av Point.

d) Lägg till en konkret metod `distanceTo(that: Shape): Double` i traiten `Shape` som räknar ut avståndet till positionen för en annan `Shape`. Testa i REPL på några instanser av `Rectangle` och `Circle`.

e) Gör så att `distanceTo` kan anropas med operatornotation.

### 10.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 7. Inmixning.** Man kan utvidga en klass med multipla traits med en kommaseparerad lista. På så sätt kan man fördela medlemmar i olika traits och återanvända gemensamma koddelar genom så kallad **inmixning**, så som nedan exempel visar.

En alternativ fågeltaxonomi, speciellt populär i Skåne, delar in alla fåglar i två specifika kategorier: Kråga respektive Ånka. Krågor kan flyga men inte simma, medan Ånkor kan simma och oftast även flyga. Fågel i generell, kollektiv bemärkelse kallas på gammal skånska för Fyle.<sup>2</sup>

```
trait Fyle:
  val läte: String
  def väsnas: Unit = print(läte * 2)
  val ärSimkunnig: Boolean
  val ärFlygkunnig: Boolean

trait KanSimma      { val ärSimkunnig = true }
trait KanInteSimma { val ärSimkunnig = false }
trait KanFlyga      { val ärFlygkunnig = true }
trait KanKanskeFlyga { val ärFlygkunnig = math.random() < 0.8 }

class Kråga extends Fyle, KanFlyga, KanInteSimma:
  val läte = "krax"

class Ånka extends Fyle, KanSimma, KanKanskeFlyga:
  val läte = "kvack"
  override def väsnas = print(läte * 4)
```

a) En flitig ornitolog hittar 42 fåglar i en perfekt skog där alla fågel sorter är lika sannolika, representerat av vektorn `fyle` nedan. Skriv i REPL ett uttryck som undersöker hur många av dessa som är flygkunniga Ånkor, genom att använda metoderna `filter`, `asInstanceOf`, `ärFlygkunnig` och `size`.

```
1 scala> val fyle =
2   Vector.fill(42)(if math.random() > 0.5 then new Kråga else new Ånka)
3 scala> fyle.foreach(_.väsnas)
4 scala> val antalFlygånkor: Int = ???
```

b) Om alla de fåglar som ornitologen hittade skulle väsnas exakt en gång var, hur många krax och hur många kvack skulle då höras? Använd metoderna `filter` och `size`, samt predikatet `ärSimkunnig` för att beräkna antalet krax respektive kvack.

```
1 scala> val antalKrax: Int = ???
2 scala> val antalKvack: Int = ???
```

**Uppgift 8. Finala klasser.** Om man vill förhindra att man kan göra **extends** på en klass kan man göra den final genom att placera nyckelordet **final** före nyckelordet **class**.

<sup>2</sup>[www.klangfix.se/ordlista.htm](http://www.klangfix.se/ordlista.htm)

- a) Eftersom klassificeringen av fåglar i uppgiften ovan i antingen Ånkor eller Krågor är fullständig och det inte finns några subtyper till varken Ånkor eller Krågor är det lämpligt att göra dessa finala. Ändra detta i din kod.
- b) Testa att ändå försöka göra en subklass Simkråga **extends** Kråga. Vad ger kompilatorn för felmeddelande om man försöker utvidga en final klass?

**Uppgift 9.** Accessregler vid arv och nyckelordet **protected**. Om en medlem i en supertyp är privat så kan man inte komma åt den i en subklass. Ibland vill man att subklassen ska kunna komma åt en medlem även om den ska vara otillgänglig i annan kod.

```
trait Super:
  private val minHemlis = 42
  protected val vårHemlis = 42

class Sub extends Super:
  def avslöja = minHemlis
  def kryptisk = vårHemlis * math.Pi
```

- a) Vad blir felmeddelandet när klassen Sub försöker komma åt minHemlis?
- b) Deklarera Sub på nytt, men nu utan den förbjudna metoden avslöja. Vad blir felmeddelandet om du försöker komma åt vårHemlis via en instans av klassen Sub? Prova till exempel med `(new Sub).vårHemlis`
- c) Fungerar det att anropa metoden kryptisk på instanser av klassen Sub?

**Uppgift 10.** Användning av **protected**. Den flitige ornitologen från uppgift 7 ska ringmärka alla 42 fåglar hen hittat i skogen. När hen ändå håller på bestämmer hen att även försöka ta reda på hur mycket oväsen som skapas av respektive fågelsort. För detta ändamål apterar den flitige ornitologen en Linuxdator på allt infångat fyle. Du ska hjälpa ornitologen att skriva programmet.

- a) Inför en **protected var** räknaläte i traiten Fyle och skriv kod på lämpliga ställen för att räkna hur många läten som respektive fågelinstans yttrar.
- b) Inför en metod antalLäten som returnerar antalet krax respektive kvack som en viss fågel yttrat sedan dess skapelse.
- c) Varför inte använda **private** i stället för **protected**?
- d) Varför är det bra att göra räknar-variabeln oåtkomlig från ”utsidan”?

**Uppgift 11.** Inmixning av egenskaper. Det visar sig att vår flitige ornitolog från uppgift 7 på sidan 362 sov på en av föreläsningarna i zoologi när hen var nolla på Natfak, och därför helt missat fylekategorin Pjodd. Hjälp vår stackars ornitolog så att fylehierarkin nu även omfattar Pjoddar. En Pjodd kan flyga som en Kråga men den ÄrLiten medan en Kråga ÄrStor. En Pjodd kvittrar dubbelt så många gånger som en Ånka kvackar. En Pjodd KanKanskeSimma där simkunnighetssannolikheten är 0.2. Låt ornitologen ånyo finna 42 slumpmässiga fåglar i skogen och filtrera fram lämpliga arter. Undersök sedan hur dessa väsnas, i likhet med deluppgift 7b.

**Uppgift 12.** Arvshierarki med matematiska tal. Studera den djupa arvshierarkin i paketet numbers i koden på efterföljande sidor. Paketet numbers modellerar olika sorters tal



i matematiken, med syftet att erbjuda ett s.k. DSL<sup>3</sup>, alltså ett specialspråk för en viss applikationsdomän<sup>4</sup>, här: domänen matematiska tal.

Du kan ladda ner koden för numbers här:

[github.com/lunduniversity/introprog/blob/master/compendium/examples/numbers.scala](https://github.com/lunduniversity/introprog/blob/master/compendium/examples/numbers.scala)

Notera speciellt metoden `reduce` som reducerar ett tal till sin enklaste form. Metoden `reduce` överskuggas på lämpliga ställen med relevant reduktion.

a) Rita en bild över typhierarkin, t.ex. som ett upp-och-nedvänt träd med bastypen `Number` som rot.

b) Skriv kod som använder de olika konkreta klasserna i **package** `numbers`.

```
1 scala> numbers. // Tryck Tab
2 AbstractComplex AbstractNatural AbstractReal Frac Nat Polar
3 AbstractInteger AbstractRational Complex Integ Number Real
4
5 scala> numbers.Integ(12)
6 res0: numbers.Integ = Integ(12)
7
8 scala> import numbers.Syntax._
9 import numbers.Syntax._
10
11 scala> 42.j
12 res1: numbers.Complex = Complex(Real(0),Real(42))
13
14 scala> 42.42.j
15 res2: numbers.Complex = Complex(Real(0),Real(42.42))
```

c) Ändra på metoden `+` i **trait** `Number` så att den blir abstrakt och implementera den i alla konkreta klasser.

d) Implementera fler räknesätt och bygg vidare på koden så som du finner intressant.

e) Gör så att metoden `reduce` i klassen `AbstractRational` använder algoritmen Greatest Common Divisor (GCD)<sup>5</sup> så som beskrivs här:

[www.artima.com/pins1ed/functional-objects.html#6.8](http://www.artima.com/pins1ed/functional-objects.html#6.8)

så att täljare och nämnare blir så små som möjligt.

```
1 package numbers
2
3 trait Number:
4   def reduce: Number = this
5   def isZero: Boolean
6   def isOne: Boolean
7   def +(that: Number): Number = ???
8
9 object Number:
10   def Zero = Nat(0)
11   def One = Nat(1)
12   def Im(im: BigDecimal) = Complex(Real(0), Real(im))
13   def Im(im: Real) = Complex(Real(0), im)
14   def j = Complex(Real(0), Real(1))
15   def Re(re: BigDecimal) = Complex(Real(re), Real(0))
16   def Re(re: Real) = Complex(re, Real(0))
17
18 trait AbstractComplex extends Number:
19   def re: AbstractReal
20   def im: AbstractReal
```

<sup>3</sup>[https://en.wikipedia.org/wiki/Domain-specific\\_language](https://en.wikipedia.org/wiki/Domain-specific_language)

<sup>4</sup><https://stackoverflow.com/questions/49216312/what-is-dsl-in-scala>

<sup>5</sup>[https://sv.wikipedia.org/wiki/St%C3%B6rsta\\_gemensamma\\_delare](https://sv.wikipedia.org/wiki/St%C3%B6rsta_gemensamma_delare)

```

21  def abs = Real(math.hypot(re.decimal.toDouble,im.decimal.toDouble))
22  def fi = Real(math.atan2(im.decimal.toDouble, re.decimal.toDouble))
23  override def isZero: Boolean = re.decimal == 0 && im.decimal == 0
24  override def isOne: Boolean = abs.decimal == 1.0
25  override def reduce: AbstractComplex = if im.decimal == 0 then re.reduce else this
26
27  final case class Complex(re: Real, im: Real) extends AbstractComplex
28
29  object Complex:
30    def apply(re: BigDecimal, im: BigDecimal) = new Complex(Real(re), Real(im))
31
32  final case class Polar(
33    override val abs: Real,
34    override val fi: Real
35  ) extends AbstractComplex:
36    override def re = Real(abs.decimal.toDouble * math.cos(fi.decimal.toDouble))
37    override def im = Real(abs.decimal.toDouble * math.sin(fi.decimal.toDouble))
38
39  object Polar:
40    def apply(abs: BigDecimal, fi: BigDecimal) = new Polar(Real(abs), Real(fi))
41
42  trait AbstractReal extends AbstractComplex:
43    def decimal: BigDecimal
44    override def isZero = decimal == 0
45    override def isOne = decimal == 1
46    override def re = this
47    override def im = Number.Zero
48    override def reduce: AbstractReal =
49      if decimal == 0 then Number.Zero else if decimal == 1 then Number.One else this
50
51  final case class Real(decimal: BigDecimal) extends AbstractReal
52
53  trait AbstractRational extends AbstractReal:
54    def numerator: AbstractInteger
55    def denominator: AbstractInteger
56    override def decimal = BigDecimal(numerator.integ)
57    override def isOne = numerator.integ == denominator.integ
58    override def reduce: AbstractRational =
59      if denominator.isOne then numerator.reduce else this // should use GCD
60
61  final case class Frac(numerator: Integ, denominator: Integ) extends AbstractRational:
62    require(denominator.integ != 0, "denominator must be non-zero")
63
64  object Frac:
65    def apply(n: BigInt, d: BigInt) = new Frac(Integ(n), Integ(d))
66
67  trait AbstractInteger extends AbstractRational:
68    def integ: BigInt
69    override def numerator = this
70    override def denominator = Number.One
71    override def isZero = integ == 0
72    override def isOne = integ == 1
73    override def decimal: BigDecimal = BigDecimal(integ)
74    override def reduce: AbstractInteger =
75      if isZero then Number.Zero else if isOne then Number.One else this
76
77  final case class Integ(integ: BigInt) extends AbstractInteger
78
79  trait AbstractNatural extends AbstractInteger
80
81  final case class Nat(integ: BigInt) extends AbstractNatural:
82    require(integ >= 0, "natural numbers must be non-negative")
83

```



```
84 extension (i: Int) def j = Number.Im(i)
85 extension (d: Double) def j = Number.Im(d)
```

## 10.3 Grupplaboration: snake0

### Mål

- ☐ Kunna använda arv.
- ☐ Kunna göra överskuggning av medlemmar i en supertyp.
- ☐ Kunna förklara begreppet dynamisk bindning.
- ☐ Kunna använda abstrakta klasser och skapa en klasshierarki.

### Förberedelser

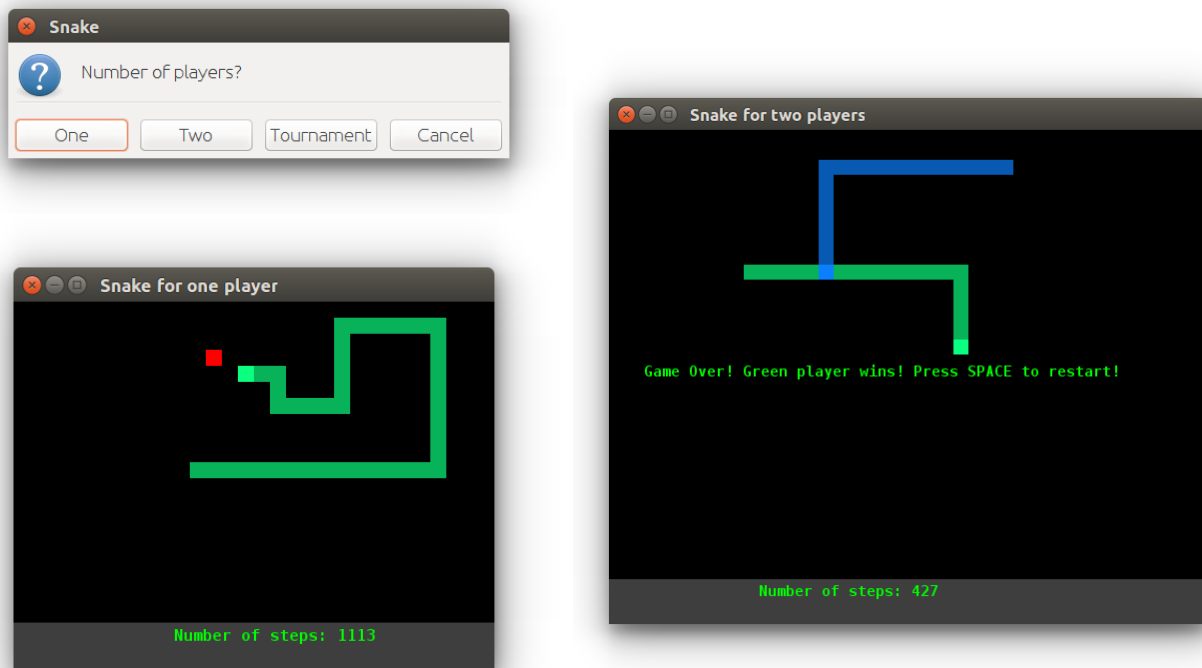
- ☐ Gör övning lookup i kapitel 9.2, speciellt uppgift 4.
- ☐ Läs dokumentationen för `introprog.BlockGame`.
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).
- ☐ Läs igenom hela laborationen och förbered dig inför första gruppmötet.
- ☐ Diskutera i din samarbetsgrupp hur ni ska dela upp koden mellan er i flera olika delar, som ni kan arbeta med var för sig. En sådan del kan vara en klass, en trait, ett objekt, ett paket, eller en funktion.
- ☐ Varje del ska ha en **huvudansvarig** individ.
- ☐ Arbetsfördelningen ska vara någorlunda jämnt fördelad mellan gruppmedlemmarna.
- ☐ Den som är huvudansvarig för en viss del redovisar den delen.
- ☐ Ni ska ta fram en gruppgemensam checklista för kodgranskning. Varje gruppmedlem ska granska minst en annan gruppmedlems kod enligt checklistan.
- ☐ Grupplaborationen görs över **två veckor** uppdelat på två delredovisningar. Vid första redovisningen ska arbetsupplägget och pågående utveckling redovisas. Vid andra tillfället ska de färdig lösningarna presenteras av respektive huvudansvarig individ.
- ☐ Vid första redovisningen ska du redogöra för handledaren hur ni delat upp koden och vem som är huvudansvarig för vad och vad ditt ansvar omfattar, samt hur ni jobbar praktiskt med att synkronisera er utveckling.
- ☐ Grupplaborationen är en **extra stor uppgift** och grupparbetet behöver ledtid för att ni ska hinna koordinera er sinsemellan. Du behöver därför planera för att arbeta med något i grupplabben i stort sett varje dag under de tillgängliga veckorna, och vara redo att bidra i diskussioner.
- ☐ Träffas i din samarbetsgrupp och diskutera ert arbetssätt utifrån följande frågor:
  - Vilka krav ska ni implementera?
  - Hur ska ni jobba med gemensamma koddelar?
  - Hur ska ni dela med er av de koddelar som ni utvecklar var för sig?

### 10.3.1 Bakgrund

Spelet *Snake*<sup>6</sup> blev mäkta populärt i Sverige redan på 1980-talet, ofta spelat på den legenda-riska datorn ABC80. Spelet finns i flera varianter, både för en spelare och som duell mellan två spelare. Varje spelare styr en mask med huvud och svans som hela tiden rör sig framåt. Det gäller att undvika att köra in i en masksvans och att samla poäng t.ex. genom att äta äpple.

Figur 10.2 visar en startdialog där man kan välja antal spelare, samt ett exempel på spel med en och två spelare. I varianten för en spelare närmar sig maskens huvud äpplet och lyckas kanske äta det om spelaren styr rätt. I varianten för två spelare vinner grön mask eftersom den blåa masken råkade köra in i den gröna maskens svans.

<sup>6</sup>Även kallat "masken". <https://sv.wikipedia.org/wiki/Snake>



**Figur 10.2:** Spelet snake för en spelare med äpple och för två spelare utan äpple.

### 10.3.2 Obligatoriska funktionella krav

Följande funktionella krav ska uppfyllas av ert program om ni är sex personer i gruppen. Om ni är färre ingår de obligatoriska krav som visas i tabell 10.1.

- ☐ **Player.** Det ska finnas spelare som motsvarar mänskliga användare och som har ett namn och fyra tangenter som den kan spela med. Varje spelare har en egen orm som den kan styra med sina tangenter.
- ☐ **Snake.** Det ska finnas ormar. En orm består av ett antal block, där det främsta blocket kallas huvud och resten av blocken kallas svans. Huvudet har en ljusare färg än kroppen. Svansens längd ökar under spelets gång. En orm rör sig i en viss riktning och varje spelare kan ändra riktningen på sin orm med sina tangenter, i en av fyra riktningar North, South, East eller West.
- ☐ **Apple.** Det ska finnas (minst ett) äpple. Ett äpple består av ett rött block och finns på en slumpvis position. Ett äpple kan ätas av en orm om ormens huvud träffar äpplet. Varje gång ett äpple äts upp av en orm så teleporteras äpplet till en ny position och kan ätas igen.
- ☐ **Banana.** Det ska finnas (minst en) banan. En banan består av tre vertikala gula block och finns på en slumpvis position. En banan äts upp av en orm om ormens huvud träffar bananen. Varje gång en banan äts upp av en orm så teleporteras bananen till en ny slumpvis position och kan ätas igen.
- ☐ **Monster.** Det ska finnas (minst ett) monster. Ett monster består av fem rosa block i kryssform. Ett monster föds på en slumpvis position och rör sig i en riktning som bestäms vid monstrets födelse. Ett orm blir uppäten och dör om ormens huvud nuddar ett monsterblock.
- ☐ **OneplayerGame.** Det ska gå att spela ensam. I varianten med en spelare finns en orm och minst ett äpple (och ev. även bananer och monster). Varje gång användarens orm lyckas äta en frukt får användaren poäng. När ormen ätit ett visst antal äpplen, eller

om ormen blivit uppäten av ett monster, är spelet slut och poängen visas. En ormsvans ska bli längre vid jämna tidsintervall eller om den äter frukt.

- ☐ **TwoPlayerGame.** Det ska gå att spela två och två. I varianten med två spelare finns två ormar. Det finns också äpplen, bananer och monster. Om en orm äter en banan blir dess svans längre. När ormen ätit ett visst antal äpplen, eller om ormen blivit uppäten av ett monster, är spelet slut och poängen visas. En ormsvans ska bli längre vid jämna tidsintervall eller om den äter frukt.
- ☐ **Settings.** Inställningar för spelet ska vara konfigurerbara genom en textfil som laddas i början av spelet. Inställningar ska vara en kontextparameter.

**Tabell 10.1:** Krav som minst ska implementeras vid respektive gruppstorlek. Om du har särskilda skäl kan du efter godkännande från kursansvarig göra labben enskilt.

Krav / Antal personer	1	2	3	4	5	6
Player	✓	✓	✓	✓	✓	✓
OnePlayerGame	✓				✓	✓
TwoPlayerGame		✓	✓	✓	✓	✓
Snake	✓	✓	✓	✓	✓	✓
Apple	✓		✓	✓	✓	✓
Banana				✓		✓
Monster			✓			✓
Settings	✓	✓	✓	✓	✓	✓

### 10.3.3 Obligatoriska design-krav

- ☐ Snake-spel ska gå att starta med huvudprogrammet nedan. Huvudprogrammet får ändras vid behov i enlighet med minimikrav vad gäller gruppstorlek i tabell 10.1, samt valbara extrakrav i avsnitt 10.3.4, och era egna ideer.

```
package snake

def createOnePlayerGame(): Unit =
  Settings.default.windowTitle = "Snake: One Player"
  OnePlayerGame().play()

def createTwoPlayerGame(): Unit =
  Settings.default.windowTitle = "Snake: Two Player"
  TwoPlayerGame().play()

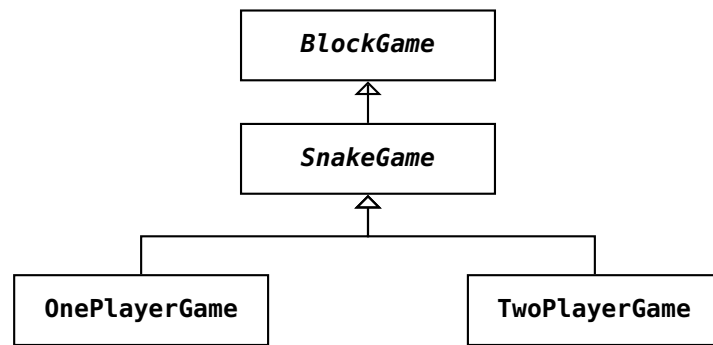
@main
def run: Unit =
  val buttons =
    Seq("One", "Two", "Competition", "Tournament", "Cancel")
  val selected =
```

```

introprog.Dialog.select("Number of players?", buttons, "Snake")
selected match
  case "One"           => createOnePlayerGame()
  case "Two"           => createTwoPlayerGame()
  case "Competition"   => println(s"TODO: $selected")
  case "Tournament"   => println(s"TODO: $selected")
  case _               => println("Goodbye!")

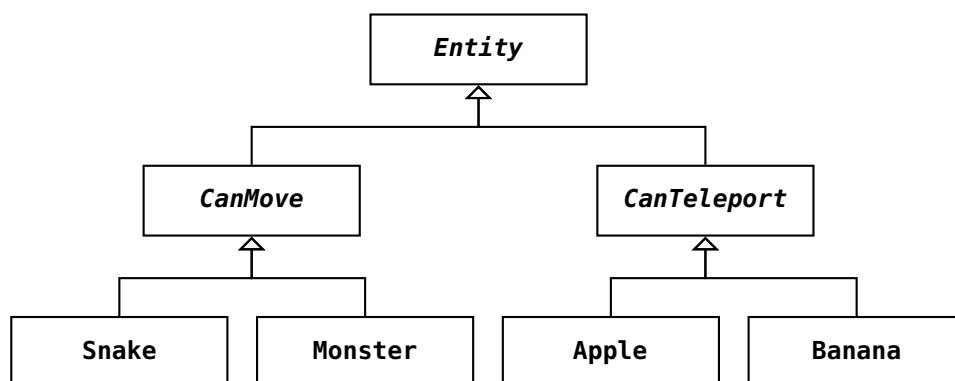
```

- ☐ Spelet ska bygga vidare på `introprog.BlockGame` enligt typhierarkin i fig. 10.3.



**Figur 10.3:** Arvshierarki med klassen `introprog.BlockGame` som bastyp.

- ☐ Ormar, monster och frukt ska utgå från bastypen `Entity` enligt typhierarkin i 10.4.



**Figur 10.4:** Arvshierarki med klassen `Entity` som bastyp.

- ☐ `Entity` representerar en varelse i ett spel och ska se ut så här:

```

package snake

trait Entity:
  def draw(): Unit

  def erase(): Unit

  def update(): Unit

  def reset(): Unit

```

```
infix def isOccupyingBlockAt(p: Pos): Boolean
```

Metoderna draw resp. erase anropas vid ritning resp. radering. Metoden reset återställer ursprungstillståndet. Metoden update anropas en gång i varje runda i spel-loopen. Predikatet isOccupyingBlockAt ger sant om positionen p finns bland de block som varelsen ockuperar på skärmen.

- ☐ CanMove representerar en entitet som kan röra sig i en viss hastighet, enligt:

```
package snake

trait CanMove extends Entity:
  def move(): Unit

  var movesPerSecond: Double = 20.0

  final def millisBetweenMoves: Int =
    (1000 / movesPerSecond).round.toInt max 1

  private var _timestampLastMove: Long = System.currentTimeMillis

  final def timestampLastMove = _timestampLastMove

  override final def update(): Unit =
    // flytta om tiden har gått millisBetweenMoves
    if System.currentTimeMillis >
      _timestampLastMove + millisBetweenMoves
    then
      _timestampLastMove = System.currentTimeMillis
      move()
```

- ☐ CanTeleport representerar en entitet som finns på en viss plats men som efter ett visst antal uppdateringar utan förvarning teleporterar sig till en ny position:

```
package snake

trait CanTeleport extends Entity:
  private var _pos = teleport()

  def pos: Pos = _pos

  protected var nbrOfStepsSinceLastTeleport = 0

  def teleportAfterSteps: Int

  def teleport(): Pos

  def update(): Unit =
    nbrOfStepsSinceLastTeleport += 1
```

```

    if nbrOfStepsSinceLastTeleport > teleportAfterSteps
    then reset()

def reset(): Unit =
    nbrOfStepsSinceLastTeleport = 0
    _pos = teleport()

```

- ☐ Det ska finnas en enumeration State i singelobjektet SnakeGame som representerar spelets övergripande tillstånd enligt följande:

```

package snake

object SnakeGame:
    enum State:
        case Starting, Playing, GameOver, Quitting
    export State.* // gör alla tillstånd synliga i SnakeGame

```

- ☐ Vid varje runda i spelloopen ska följande logik exekveras. Denna kod placeras förslagsvis i gameLoopAction, se vidare SnakeGame i avsnitt 10.3.5.

```

    if state == Playing && !isPaused then
        _iterationsSinceStart += 1
        entities.foreach(_.erase())
        entities.foreach(_.update())
        entities.foreach(_.draw())
        onIteration()
        if isGameOver then enterGameOverState()

```

- ☐ Det ska finnas ett singelobjekt Colors där alla färger som används i spelet samlas.
- ☐ Filen pairs.scala ska enligt laborationsförberedelser i övningsuppgift 4 på sidan 358 innehålla Pair[T], Dim, Pos, Dir, North, South, East, West. Se workspace här: <https://github.com/lunduniversity/introprog/tree/master/workspace/>
- ☐ Klassen Player ska se ut som följer:

```

package snake

class Player(
    var name: String,
    var keyMap: Player.KeyMap,
    val snake: Snake,
    var points: Int = 0,    // TODO: count points when e.g. eating apple
):
    def handleKey(key: String): Unit =
        ??? // om key ingår i keyMap så uppdatera snake.dir

object Player:
    enum KeyMap(left: String, right: String, up: String, down: String):
        val dir = Map(left -> West, right -> East, up -> North, down -> South)
        case Letters extends KeyMap("a", "d", "w", "s")
        case Arrows extends KeyMap("Left", "Right", "Up", "Down")

```

### 10.3.4 Valbara krav – varje person ska välja minst ett

Varje person i gruppen ska implementera *minst ett* (gärna flera) av kraven nedan. Vid implementation av flera av dessa krav blir spelet väsentligt roligare.

- ☐ **Points.** Inför ett poängsystem, där poängen beror på t.ex. längden på svansen, antalet steg, antalet svängar, antal uppätta äpplen, etc.
- ☐ **Highscore.** Spelet ska visa en lista med de spelare som fått flest poäng.
- ☐ **Apple.** Om inte redan ingår bland obl. krav enl. 10.1.
- ☐ **Monster.** Om inte redan ingår bland obl. krav enl. 10.1.
- ☐ **Banan.** Om inte redan ingår bland obl. krav enl. 10.1.
- ☐ **SelfTailCrash.** Om en spelare kör in i sin egen orms svans så är spelet förlorat. (Om detta krav ej implementeras så får man köra igenom sin egen svans utan att något händer.)
- ☐ **BoundaryCrash.** Om en spelare kör utanför spelplanen så är spelet förlorat. (Om detta krav ej implementeras så ska ormen fortsätta på andra sidan spelplanen när man når kanten.)
- ☐ **EnterPlayerName.** Spelare kan ange sitt namn, t.ex. via en dialog eller genom argument till main. Namnet används i meddelandefältet vid poängräkning och i meddelanden om vem som vunnit.
- ☐ **OnePlayerGame.** Du kan välja att implementera OnePlayerGame om det inte redan ingår i de obligatoriska kraven.
- ☐ **TwoPlayerComp extends Competition.** Två spelare ska kunna tävla i en bäst-av- $n$ -matcher-tävling i en sekvens av TwoPlayerGame.play, där den som vinner flest matcher blir totalvinnare.
- ☐ **SinglePlayerComp extends Competition.** Flera spelare ska kunna tävla i en-persons-Snake, där den som får flest poäng av  $n$  OnePlayerGame-spel blir totalvinnare.
- ☐ **Tournament extends Competition.** Många spelare ska kunna spela en turnering.<sup>7</sup> Namnen på spelarna läses in från en textfil. Valbara varianter:
  - ☐ **KnockOut extends Tournament.** Det ska gå att spela en utslagsturnering, som avslutas med final efter semi-final, etc., beroende på antal spelare.
  - ☐ **RoundRobin extends Tournament.** Det ska gå att spela en alla-möter-alla-turnering, där alla möjliga par av spelare möts i slumpvis ordning.

### 10.3.5 Tips och förslag

I detta stycke presenteras skisser till några av de klasser som behövs i enlighet med designkraven. Det är tillåtet att ändra, ta bort och lägga till, så länge de obligatoriska designkraven uppfylls. Koden finns här:

<https://github.com/lunduniversity/introprog/tree/master/workspace/>

Här följer en skiss på klassen Snake:

```
package snake

import java.awt.Color

class Snake (
  val initPos: Pos,
  val initDir: Dir,
  val headColor: Color,
  val tailColor: Color,
)(using ctx: SnakeGame, settings: Settings) extends CanMove:
```

<sup>7</sup><https://en.wikipedia.org/wiki/Tournament>



```

var dir: Dir = initDir
val initBody: List[Pos] = List(initPos + initDir, initPos)
val body: scala.collection.mutable.Buffer[Pos] = initBody.toBuffer

val initLength: Int = settings.snake.initLength
val growEvery: Int = settings.snake.growEvery
val startGrowingAfter: Int = settings.snake.startGrowingAfter

private var _nbrOfSteps = 0
def nbrOfSteps: Int = _nbrOfSteps

private var _nbrOfApples = 0
def nbrOfApples: Int = _nbrOfApples

def reset(): Unit = ??? // återställ starttillstånd, ge rätt svanslängd

def grow(): Unit = ??? // väx i rätt riktning med extra svansposition

def shrink(): Unit = ??? // krymp svansen om kroppslängden är större än 2

def isOccupyingBlockAt(p: Pos): Boolean = ??? // kolla om p finns i kroppen

def isHeadCollision(other: Snake): Boolean = ??? // kolla om huvudena krockar

def isTailCollision(other: Snake): Boolean = ??? // mitt huvud i annans svans

private var _isEatenByMonster: Boolean = false
def isEatenByMonster: Boolean = _isEatenByMonster
def eatenByMonster(): Unit = ???

def move(): Unit = ???
  // väx och krymp enl. regler
  // åtgärder om äter frukt eller blir uppäten av monster

override def toString = // bra vid println-debugging
  body.map(p => (p.x, p.y)).mkString(">:", "~", s" going $dir")

def draw(): Unit = ???

def erase(): Unit = ???

```

Här följer en skiss på den abstrakta klassen SnakeGame med de abstrakta metoderna isGameOver och play som överskuggas i de efterföljande underklasserna OnePlayerGame och TwoPlayerGame:

```

package snake

object SnakeGame:
  enum State:
    case Starting, Playing, GameOver, Quitting
  export State.*

abstract class SnakeGame(settings: Settings) extends introprog.BlockGame(
  title           = settings.windowTitle,
  dim             = settings.windowSize,
  blockSize       = settings.blockSize,
  background      = settings.background,
  framesPerSecond = settings.framesPerSecond,
  messageAreaHeight = settings.messageAreaHeight,
  messageAreaBackground = settings.messageAreaBackground
):
  // exempel på olika synlighet (diskutera val av synlighet utifrån användning)
  var entities: Vector[Entity] = Vector.empty
  protected var players: Vector[Player] = Vector.empty

```

```

private var isPaused = false

import SnakeGame.*

protected var state: State = Starting
private var _iterationsSinceStart = 0
def iterationsSinceStart = _iterationsSinceStart

def enterStartingState(): Unit = ??? //sudda, meddela "tryck space för start"

def enterPlayingState(): Unit = ??? //sudda, för varje entitet: nollställ & rita

def enterGameOverState(): Unit = ??? // meddela "game over"

def enterQuittingState(): Unit =
  println("Goodbye!")
  pixelWindow.hide()
  state = Quitting

def randomFreePos(): Pos =
  ??? // dra slump-pos tills ledig plats, används av frukt, monster

override def onKeyDown(key: String): Unit =
  println(s""key "$key" pressed"")
  state match
    case Starting => if key == " " then enterPlayingState()

    case Playing =>
      if key == "Esc" then
        println(s"Toggle pause: isPaused == $isPaused")
        isPaused = !isPaused
      else
        players.foreach(_.handleKey(key))

    case GameOver =>
      if key == " " then enterPlayingState()
      else if key == "Esc" then enterQuittingState()

    case _ =>

override def onClose(): Unit =
  println("Window Closed!")
  enterQuittingState()

/** Implement this with logic for when to end the game */
def isGameOver: Boolean

/** Override this if you want to add game-logic in gameLoopAction
 * Call super.onIteration() if you want to keep the step counter.
 */
def onIteration(): Unit =
  clearMessageArea()
  drawTextInMessageArea(s"Number of steps: $iterationsSinceStart", 10, 2)

override def gameLoopAction(): Unit =
  if state == Playing && !isPaused then
    _iterationsSinceStart += 1
    entities.foreach(_.erase())
    entities.foreach(_.update())
    entities.foreach(_.draw())
    onIteration()
  if isGameOver then enterGameOverState()

final def start(ps: Player*)(es: Entity*): Unit =

```

```
players = ps.toVector
entities = es.toVector
isPaused = false
pixelWindow.show() // möjliggör omstart även om fönstret stängts...
enterStartingState()
gameLoop(stopWhen = state == Quitting)

/** Implement this with a call to start with specific players and entities. */
def play(playerNames: String*): Unit
```

# Kapitel 11

## Kontextuella abstraktioner och varians

Begrepp som ingår i denna veckas studier:

- ☐ övre- och undre typgräns
- ☐ varians
- ☐ kontravarians
- ☐ kovarians
- ☐ typjoker
- ☐ egentyp
- ☐ givet värde (given)
- ☐ kontextparameter (using)
- ☐ generiska extensionsmetoder
- ☐ ad hoc polymorfism
- ☐ kontextgräns
- ☐ typklass
- ☐ api
- ☐ kodläsbarhet
- ☐ granskningar

## 11.1 Teori

### 11.1.1 Repetition: Typparametrar och generiska strukturer

- Med hjälp av **typparametrar** (eng. *type parameters*) kan du skapa **generiska strukturer** (eng. *generic structures*).
- Typparametrar skrivs inom hakparenteser, exempelvis: [T, U]
- Generiska strukturer fungerar för *olika* typer, **okända** vid **deklaration**.

```
case class Pair[A, B](a: A, b: B): // A och B är obundna (fria) inom []
  def swap: Pair[B, A] = Pair(b, a) // A och B bundna då swap saknar []
```

- Kompilatorn säkerställer **korrekt användning** redan vid *kompilering*.
- På användningsplatsen i koden (vid anrop eller instansiering) **binds** typparametrar till ”verkliga” typer enligt typsystemets regler.

```
1 scala> val p = Pair("hej", 42)
2 val p: Pair[String, Int] = Pair(hej,42)
3
4 scala> p.swap
5 val res0: Pair[Int, String] = Pair(42,hej)
```

- Skriver du inte ut typerna försöker kompilatorn **härleda** (eng. *infer*) dem.
- Klassen Pair kallas även **typkonstruktor** (eng. *type constructor*) då den ”verkliga” typen skapas först vid användning Pair[String, Int].

### 11.1.2 Olika sätt att begränsa typer

Det finns i Scala flera olika sätt att begränsa vilka typer du vill tillåta. En översikt av några möjligheter:

- Ge **övre** gräns för typparametrar (eng. *upper bound*)
- Ge **undre** gräns för typparametrar (eng. *lower bound*)
- Ge en **kontextgräns** för typparametrar i relation till implicit givna typer (eng. *context bound*)
- Begränsa möjlig inmixning vid deklaration av en trait med en explicit **egentyp** (eng. *self type*)

### 11.1.3 Övre och undre typgräns

Med typoperatorerna **<:** och **>:** går det att begränsa vilka typer som kan bindas till en typparameter i en generiska struktur.

Antag att T är en obunden typparameter medan U och L är bundna.

- med T **<:** U blir U en övre gräns (eng. *upper bound*) för T (en ”högsta” möjliga typ i typhierarkin)
- med T **>:** L blir L en undre gräns (eng. *lower bound*) för T (en ”lägsta” möjliga i typhierarkin)

- Minnesregel för typgränser: kolon på slutet.
- Alla typer  $T$  är implicit  $T >: \text{Nothing} <: \text{Any}$

Exempel:

```
trait Grönsak { def vikt: Int }

def f[T <: Grönsak](x: T): Int = x.vikt
```

Kompilatorn använder den övre typgränsen för att konstatera att metoden `vikt` är tillgänglig via den generiska parametern `x`.

#### 11.1.4 Exempel på övre och undre typgräns

```
class Djur
class Katt extends Djur
class Hund extends Djur
class Robothund extends Hund

def testUpperBound[T <: Hund](x: T) = println(x)
def testLowerBound[T >: Hund](x: T) = println(x)
```

```
1 scala> testUpperBound[Katt](Katt())
2 -- Error:
3 1 |testUpperBound[Katt](Katt())
4   |      ^
5   |      Type argument Katt does not conform to upper bound Hund
6
7 scala> testLowerBound[Robothund](Robothund())
8 -- Error:
9 1 |testLowerBound[Robothund](Robothund())
10  |      ^
11  |      Type argument Robothund does not conform to lower bound Hund
```

#### 11.1.5 Vad är varians?

**Är en kattbur också en djurbur??**



Om vi tillåter **varians** så blir generiska strukturer mer **flexibla**.

### 11.1.6 Varför behövs varians?

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Nedan fungerar inte! Buren ovan är **invariant** (oflexibel i sin typparameter).

```
1 scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
2 -- Error:
3 1 |val djurbur: Bur[Djur] = Bur[Katt](Katt())
4   |           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
5   |           Found:    Bur[Katt]
6   |           Required: Bur[Djur]
```

Varför fungerar detta??

```
1 scala> val djur: Vector[Djur] = Vector[Katt](Katt())
2 val djur: Vector[Djur] = Vector(Katt())
```

Vector är deklarerad som **kovariant** och därmed mer flexibel!

### 11.1.7 Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor  $F$  gäller att:  
om  $T <: U$  så  $F[T] <: F[U]$  (subtypsflexibel "på samma håll")
- I Scala kan du få en kovariant typkonstruktor med  $+$  före typparametern:

```

trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A) // kovariant tack vare + före A

```

- Nu funkar det :)

```

1 scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
2 val djurbur: Bur[Djur] = Bur(Katt())

```

- Bur[Katt] är nu en **subtyp** till Bur[Djur].
- **Oföränderliga** samlingar görs ofta kovarianta, t.ex Vector, Option.
- Generiska enumerationer blir mer flexibla om du gör dem kovarianta:

```

enum Toalett[+T]:
  case Upptagen(x: T)
  case Ledig

```

### 11.1.8 Kontravarians

Är en kattveterinär också en djurveterinär?



Ibland vill vi ha variansen på andra hållet: En veterinär som bara kan behandla katter ska inte få behandla vilket djur som helst.

### 11.1.9 Kontravarians (eng. *contravariance*)

- För en **kontravariant** typkonstruktor F gäller att:  
**om**  $T <: U$  **så**  $F[U] <: F[T]$  (subtypsflexibel "på fel håll")



- Du skapar en kontravariant typkonstruktor med - före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

class Veterinär[-A]:    // kontravariant med -
  def behandla(x: A) = println(s"$this har behandlat $x")
```

- Nu funkar det ”baklänges” (men inte på andra hållet):

```
scala> val kattveterinär: Veterinär[Katt] = Veterinär[Djur]()
val kattveterinär: Veterinär[Katt] = Veterinär@77b6d94c

scala> val kattveterinär: Veterinär[Djur] = Veterinär[Katt]()
-- Error:
1 | val kattveterinär: Veterinär[Djur] = Veterinär[Katt]()
  |                                     ~~~~~
  |                                     Found:    Veterinär[Katt]
  |                                     Required: Veterinär[Djur]
```

### 11.1.10 Variansproblem – tack kompilatorn!

- En typkonstruktor kan **inte** vara kovariant om typparametern används som parame-  
tertyp för metoder (så kallad *kontravariant position*):

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur
```

```
scala> case class Bur[+A](a: A):
  def bytTill(x: A): Bur[A] = Bur(x)
-- Error:
2 | def bytTill(x: A): Bur[A] = Bur(x)
  |     ^^^^
  |     covariant type A occurs in contravariant position in type A of parameter x
```

- Här måste typen tillåtas variera ”uppåt” med hjälp av en **undre gräns**:

```
case class Bur[+A](a: A):
  def bytTill[B >: A](x: B): Bur[B] = Bur(x)
```

```
scala> Bur[Katt](Katt()).bytTill(Hund())
val res1: Bur[Djur] = Bur(Hund())
```

### 11.1.11 När använda vilken slags varians?

- Oföränderliga generiska klasser som har metoder som är **producenter** av nya oförän-  
derliga generiska typer passar ofta bäst som **kovarianta**, t.ex. `Vector[+T]`, `Option[+T]`

- En typkonstruktor av T, som har metoder som är **konsumenter** av T, kan vara **kontra-variant**, t.ex. Veterinär[-T].  
Den kan också vara **kovariant** om konsumerande metoder **vidgar** inparametertypen till B där B >: T
- En typkonstruktor med flera parametrar kan vara **både** kontravariant **och** kovariant, t.ex. Function1[-A, +B]  
– detta är den egentliga typen av det ”syntaktiska sockret” A => B  
(ett parametervärde: A *konsumeras* och ett returvärde: B *produceras*)
- **Förändringsbara** strukturer måste vara **invarianta**, annars väntar **kaos**!  
Antag att det finns en Bur som kan ändras på plats via metoden bytTill *och samtidigt* vore kovariant (varning för känsliga kodare):

```
ejscala> val kattBur: Bur[Katt] = Bur(Katt())

ejscala> val djurBur: Bur[Djur] = kattBur // en kovariant referens till samma Bur

ejscala> djurBur.bytTill(Hund())           // ändra på plats

ejscala> val katt: Katt = kattBur.släppUt // KAOS! typosäkert hundkatt-monster :(
```

### 11.1.12 Typjoker (eng. *wildcard type*)

**Invarianta** klasser har oflexibel typparameter, vilket begränsar användningen.

```
1 scala> class Box[A](val value: A)
2
3 scala> val b: Box[Any] = Box[Int](42)
4 -- Error:
5 1 |val b: Box[Any] = Box[Int](42)
6   |                      ^^^^^^^^^^^
7   |                      Found:    Box[Int]
8   |                      Required: Box[Any]
```

Tveksam räddning: En **typjoker** (eng. *wildcard type*) skrivs med ett frågetecken och ger en helt okänd typ som **ej kontrolleras av kompilatorn**.

```
1 scala> var whatever: Box[?] = Box[Int](42)
2 var whatever: Box[?] = Box@70d7a49b
3
4 scala> whatever = Box("hej")
5 whatever: Box[?] = Box@43120a77
```

Använd bara typjoker om det *verkligen* behövs.

### 11.1.13 Mer om varians för den nyfikne

- <https://docs.scala-lang.org/scala3/book/types-variance.html>
- [https://en.wikipedia.org/wiki/Covariance\\_and\\_contravariance\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science))
- <https://www.artima.com/pins1ed/type-parameterization.html>

### 11.1.14 Typbegränsning med egentyp (eng. *self type*)

- Det går att gå ett explicit, valfritt namn, t.ex. `self`, på "mig själv", alltså denna instans, genom att skriva `self =>` i kroppen på en trait eller klass.
- En typbegränsning på den egna instansen kallas **egentyp** (eng. *self type*).
- Användbart vid inmixning: du kan begränsa med vad denna trait får mixas in.

```
trait Grönsak { def vikt: Int }
trait KanSkalas:
  self: Grönsak =>
  def viktEfterSkalning = 0.99 * vikt
```

```
scala> val g = new Grönsak with KanSkalas { val vikt = 100 }
val g: Grönsak & KanSkalas = anon1@70a91d72

scala> g.viktEfterSkalning
val res0: Double = 99.0
```

- Även om `KanSkalas` **inte** gör **extends** så är **ändå** `vikt` tillgänglig i dess kropp, eftersom vi kräver att den egna typen är en `Grönsak`.
- Kan användas för att göra s.k. **beroendeinjektion** (eng. *dependency injection*).  
[https://en.wikipedia.org/wiki/Dependency\\_injection](https://en.wikipedia.org/wiki/Dependency_injection)

### 11.1.15 Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**
- **Kombinationen** av OO och FP är **extra** kraftfull och flexibel!
- Men det finns **svårigheter**:
  - OO: ibland svårt att resonera om ändrade tillstånd och dynamisk bindning
  - FP: kan bli många parametrar som måste upprepas vid nästlade anrop
- Till vår räddning, fler coola FP-grejer:
  - **Kontextuella abstraktioner**: möjliggör (på annat ställe) **givna** värden som finns *implicit* (alltså underförstått, utan att vi behöver explicit skriva ut dem).
  - **Ad hoc polymorfism**: möjliggör olika implementationer beroende på typ, *statiskt* härledda och utan att det behöver finnas en arvsrelation.

### 11.1.16 Repetition: default-argument

- Vi har tidigare sett att kompilatorn, med **default-argument**, kan **fylla i** värden, som vi inte måste skriva, vid funktionsanrop:

```
scala> def f(x: Int, y: Int = 42) = x + y
def f(x: Int, y: Int): Int

scala> f(1,2)           // explicit givet y-värde
val res0: Int = 3

scala> f(1)             // vi kan också skippa y-värdet
val res1: Int = 43      // då används default-argumentet
```

### 11.1.17 Repetition: uppdelade parameterlistor

- Vi har tidigare sett uppdelade parameterlistor, som möjliggör stegvis applicering (s.k. Curry-funktioner):

```
scala> def f(x: Int)(y: Int = 42) = x + y
def f(x: Int)(y: Int): Int

scala> val g = f(1) // en ny funktion utan default-arg
val g: Int => Int = Lambda1352/0x08406d0840@dbc7e0a

scala> f(1)()      // y-värdet fylls i av kompilatorn
val res4: Int = 43
```

- Kan vi ta detta ett steg till och **frikoppla** deklarationen av funktionen **från** det givna värdet, och ge detta på annan plats baserat på typen?
- I Scala 3 görs detta med nyckelorden **given** resp. **using** (i Scala 2 användes gamla nyckelordet **implicit**)

### 11.1.18 Givna värden som alternativ till default-argument

Exmpel på användning av **using** och **given**:

```
1 scala> def f(x: Int)(using y: Int) = x + y
2 def f(x: Int)(using y: Int): Int
3
4 scala> f(1)(using 2) //explicit värde används
5 val res0: Int = 3
6
7 scala> f(1)           // kompilatorn hittar inget givet Int-värde
8 -- Error:
9 1 |f(1)
10 |   ^
11 |   no implicit argument of type Int was found for parameter y of method f
12
13 scala> given Int = 42 // låt 42 vara givet
14 lazy val given_Int: Int
15
16 scala> f(1)           // kompilatorn kan nu framkalla givet värde av typen Int
17 val res0: Int = 43
```

Det går också att ge det givna värdet ett namn vid behov, men ofta onödigt:

**given** blabla: Int = 42

### 11.1.19 Går det lika bra att ha en variabel som default-värde?

- Ett försök att skapa ett kontextberoende default-värde med en variabel:

```
scala> var ctx = 42
var ctx: Int = 42

scala> def f(x: Int = ctx) = x + 1
def f(x: Int): Int

scala> f()
val res0: Int = 43

scala> ctx = 0

scala> f()
val res1: Int = 1
```

- Här utgörs ”kontexten” av referensen till ett föränderligt värde vid **körtid** som måste synas i **aktuell** namnrymd. Funktionen är ej heller äkta : (
- Denna lösning kan inte erbjuda flera kontextberoende default-värden, så att funktionen fungerar i många **olika** sammanhang.
- För givna värden med **given** så finns i stället en **implicit namnrymd** (eng. *implicit scope*) med speciella synlighetsregler vid **using**.

### 11.1.20 Kontextuella abstraktioner med given, using

**Kontextuell abstraktion** (eng. *contextual abstraction*) i Scala:

- Nyckelordet **given** anger ett **givet värde** av en viss typ.
- Värdet **framkallas** vid anrop med **kontextparameter** av given typ.

```
object MinKontext:
  given String = "tagen för givet"
  def framkalla(using s: String) = s //kontextparametrar märks med using
```

```
scala> MinKontext.framkalla(using "explicit")
val res0: String = explicit

scala> given String = "ta denna"
lazy val given_String: String

scala> MinKontext.framkalla
val res1: String = ta denna

scala> import MinKontext.given String // speciell import-syntax för givna värden

scala> MinKontext.framkalla
val res2: String = tagen för givet
```

### 11.1.21 Framkalla värde med summon

I standardbiblioteket för Scala 3 finns en generisk variant av metoden `framkalla` i föregående exempel, som är definierad så här:

```
def summon[T](using x: T) = x
```

Funktionen `summon` kan användas för att testa vilket värde kompilatorn framkallar i en viss kontext:

```
object MinKontext:
  given String = "tagen för givet"
```

```
scala> summon[String]
-- Error:
1 |summon[String]
  |             ^
  | no implicit argument of type String was found for parameter x of
  | method summon in object Predef. The following import might fix the problem:
  | import MinKontext.given_String

scala> import MinKontext.given // importerar alla givna värden i MinKontext

scala> summon[String]
val res0: String = tagen för givet
```

### 11.1.22 Prioritetsordning vid framkallning av givna värden

- Det kan finnas flera **olika** givna värden av **samma** typ.
- Men då får det ju inte vara tvetydigt vilket implicit värde som ska väljas.
- Därför väljer kompilatorn ett givet värde enligt denna prioritering:
  1. **Explicita** argument till kontextparametrar märkta med **using**
  2. **given** och **import given** ... i aktuell namnrymd (eng. *current scope*)
  3. **given**-värden i **kompanjonsobjekt** för den använda typen.
  4. ... (fler knepiga regler som vi inte går in på här)
- Om flera givna värden kan framkallas för typer som ingår i en gemensam arvshierarki så väljer kompilatorn det givna värdet som är av den *mest specifika* typen.

Framkallning av givna värden vid anrop med kontextparametrar har flera namn på engelska: *to summon*, eller *term inference*, eller *implicit resolution*.

### 11.1.23 Ad hoc polymorfism med typklasser

- Med Ad hoc polymorfism kan det finnas *olika* implementationer för *olika* typer utan användning av arv (**extends**).
- Uppfanns i språket ML och vidareutvecklades i språket Haskell.
- I Haskell kallas mekanismen för Ad hoc polymorfism för **typklass**.
- Typklasser skapas i Scala genom att **kombinera** parametrisk polymorfism (typparametrar) med kontextuella abstraktioner (given-using).
- En **typklass i Scala** är en tillståndslös **trait** med minst en typparameter och minst en abstrakt metod.

```

trait Parser[T]:    // en typklass för att tolka och omvandla strängar till godtycklig typ
  def fromString(value: String): Option[T]

object Parser:      // olika implementationer erbjuds som givna värden
  given Parser[Int] with // nyckelordet with behövs då given implementerar abstrakt metod
    def fromString(value: String): Option[Int] = value.toIntOption

```

- Den specifika implementationen framkallas via en kontextuell parameter:

```

scala> def minAnvändningAvParser[T](s: String)(using p: Parser[T]) = p.fromString(s)

scala> minAnvändningAvParser[Int]("12")
val res0: Option[Int] = Some(12)

```

### 11.1.24 Hur få typklassen Parser att funka för fler typer?

```

1 scala> minAnvändningAvParser[java.awt.Color]("Color(120,10,0)")
2 -- Error:
3 1 |minAnvändningAvParser[java.awt.Color]("Color(120,10,0)")
4   |                               ^
5   | no implicit argument of type Parser[java.awt.Color] was found
6   | for parameter p of method minAnvändningAvParser

```

```

given Parser[java.awt.Color] with
  def fromString(value: String): Option[java.awt.Color] =
    if !value.startsWith("Color(") then None else
      val trimmed = value.trim.stripPrefix("Color(").stripSuffix(")")
      trimmed.split(",").map(_.toIntOption) match
        case Array(Some(r),Some(g),Some(b)) => Some(java.awt.Color(r, g, b))
        case _ => None

```

```

scala> minAnvändningAvParser[java.awt.Color]("Color(120,10,0)")
val res1: Option[java.awt.Color] = Some(java.awt.Color[r=120,g=10,b=0])

```

Detta ger **flexibilitet**: Jag kan ge givna värden för mina **egna** typer!

### 11.1.25 Namnet på kontextparametrar kan utelämnas

- Namnet på det givna värdet behövs ofta inte – det är ju *typen* som är det viktiga.
- Därför är det tillåtet att utelämnas parameternamnet vid **using**.
- Det givna värdet kan istället framkallas med summon vid behov:

```

def minAnvändningAvParser[T](s: String)(using Parser[T]) =
  summon[Parser[T]].fromString(s)

```

### 11.1.26 Kontextgräns

Denna form av **using**-parametrar...

```
def minAnvändningAvParser[T](s: String)(using Parser[T]) = ???
```

...är så vanliga att Scala har ett kortare skrivsätt som kallas **kontextgräns**:

```
def minAnvändningAvParser[T: Parser](s: String) = ???
```

Om  $F[A]$  är en typkonstruktor och du skriver  $[T: F]$  så blir  $F$  en så kallad **kontextgräns** (eng. *context boundary*). Kompilatorn expanderar detta automatiskt till kontextparametern (**using**  $F[T]$ )

### 11.1.27 Ännu smidigare typklass med extensionsmetod

Kombinera kontextgräns med extensionsmetod och få smidig punktnotation.

```
extension [T: Parser](s: String)
  def minAnvändningAvParser(default: T): T =
    summon[Parser[T]].fromString(s).getOrElse(default)
```

```
1 scala> "12".minAnvändningAvParser(default = 42)
2 val res2: Int = 12
3
4 scala> "gurka".minAnvändningAvParser(default = 42)
5 val res3: Int = 42
6
7 scala> "Color(100,100,0)".minAnvändningAvParser(java.awt.Color.black)
8 val res4: java.awt.Color = java.awt.Color[r=100,g=100,b=0]
```

- Här behöver typparametern  $T$  ej anges explicit eftersom kompilatorn kan använda default-värdet för att härleda typen.
- Extensionsmetoder kan skrivas i efterhand på valfri plats och även göras tillgängliga med import.

### 11.1.28 Sortera samlingar med given ordning

```
scala> case class Gurka(namn: String, vikt: Int)

scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))
val xs: Vector[Gurka] = Vector(Gurka(a,100), Gurka(b,50), Gurka(c,100))

scala> xs.sorted
-- Error:
1 |xs.sorted
  | ^
  | No implicit Ordering defined for B
  | where: B is a type variable with constraint >: Gurka
```

Detta kan fixas genom att tillhandahålla en given ordning för Gurka:



```
given Ordering[Gurka] with
  def compare(x: Gurka, y: Gurka): Int =
    if (x == y) then 0
    else if x.vikt < y.vikt then -1
    else 1
```

```
1 scala> xs.sorted // nu funkar det :)
2 val res0: Vector[Gurka] = Vector(Gurka(b,50), Gurka(a,100), Gurka(c,100))
```

### 11.1.29 Sortera samlingar med ännu smidigare given ordning

- Det är vanligt att man vill definiera egna ordningsrelationer.
- Därför finns en smidig hjälpmetod i kompanjonsobjektet för typklassen `Ordering` som heter `fromLessThan`:

```
given Ordering[Gurka] =
  Ordering.fromLessThan((g1, g2) => g1.vikt < g2.vikt)
```

- Den tar som inparameter en funktion som tar två instanser som ska ordnas och ger **true** om den första ska anses **mindre** (alltså kommer **före**) enligt valfri definition.
- `fromLessThan` returnerar en `Ordering` som du kan låta vara givet.
- Prova gärna detta på veckans fördjupningsövningar.
- Läs mer om typklasser i Scala här:  
<https://docs.scala-lang.org/scala3/reference/contextual/type-classes.html>

### 11.1.30 Vad är ett bra api?

- Ett api (eng. *Application Programming Interface*) är ett gränssnitt för applikationsprogrammering.
- Med "gränssnitt" menas att det (i koden) finns en gräns mellan vad som syns utåt och vad som finns innanför "under huven".
- Det finns många kvalitetsaspekter som är önskvärda för ett api:
  - Enkelt att använda på utsidan även om insidan är komplex.
  - Vadå "enkelt att använda"?
    - \* Lätt att lära
    - \* Lätt att komma ihåg
    - \* Lätt att begripa
    - \* Najs upplevelse (subjektivt)
  - Löser ett (generellt) problem på ett bra sätt. Vadå "bra"?
    - \* Snabbt (hög prestanda)
    - \* Snålt (effektiv användning av resurser, t.ex. minne)
    - \* ...
- Intressant föredrag av Joshua Bloch (Google), om god api-design:

- <https://research.google.com/pubs/archive/32713.pdf>
  - <https://youtu.be/aAb7hSCtvGw>
- 

### 11.1.31 Api-desgin med Scala

- Kombinera paradigm OO+FP för att välja bäst lämpade lösningen.
  - Avancerade abstraktionsmekanismer som kan vara utmanande för api-konstruktören men samtidigt bli enkelt för api-användaren
  - Gör det möjligt att skapa ett api som fungerar i flera körmiljöer:
    - På desktop och i back-end med JVM och Graal VM
    - I front-end i webbläsaren med Scala JS
    - Direkt till plattformsspecifik maskinkod med Scala Native
  - Avancerade saker som vi inte gått in på i kursen men som kan hjälpa api-konstruktörer att göra lättanvänt api:
    - Metaprogrammering
    - Typklassderivering
    - Kontextfunktioner
    - Opaka typer
    - Explicita null-typer
    - Lambda på typnivå och match-typer
    - ...
-

## 11.2 Övning context

### Mål

- ☐ Kunna förklara vad en kontextparameter är.
- ☐ Kunna förklara nyttan med kontextparametrar jämfört med en globala variabler och defaultargument vid lösning av konfigurationsproblemet.
- ☐ Kunna använda enkla kontextuella abstraktioner med **given** och **using**.

### Förberedelser

- ☐ Studera begreppen i kapitel 11

#### 11.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** *Kontextparameter.* Deklarera följande funktioner som tar ett heltal som kontextparameter. Skapa även en **given**-deklaration som erbjuder det givna heltalsvärdet noll:

```
scala> def f(using i: Int) = i + 1  
  
scala> def g(x: Int)(using y: Int) = x + y
```

- Anropa funktionerna `f` och `g` med ett explicit givet argument som skiljer sig från det givna heltalsvärdet med hjälp av **using** i anropet. Vad händer om du utelämnar **using**?
- Anropa funktionerna `f` och `g` utan att ange **using**-argument. Förklara vad som händer.
- Går det att blanda vanliga parametrar och kontextparametrar i samma parameterlista? Om inte vad händer?

**Uppgift 2.** *Flera olika givna värden i lokal kontext.* Olika värden beroende på kontext.

```
case class Delta(value: Int)  
object Delta:  
  given default: Delta = Delta(1)  
  
def inc(x: Int)(using dx: Delta) = x + dx.value  
  
object Context1:  
  val a = inc(1)  
  
object Context2:  
  given Delta = Delta(42)  
  val a = inc(1)
```

- Vilket värde har `Context1.a`?
- Vilket värde har `Context2.a`?
- Förklara vad som händer.

**Uppgift 3.** *Lösning på konfigurationsproblemet med hjälp av givna värden.* Antag att vi vill kunna konfigurera beteendet hos en funktion för att göra den mer flexibel. Nedan visas tre principiellt olika sätt att göra detta på för en funktion `greet` som skriver ut en hälsning:

1) en globalt åtkomlig variabel, 2) defaultargument, samt 3) kontextuell abstraktion med **given** och **using**.

```
object GlobalVar:
  case class GreetConfig(greeting: String, receiver: String)
  object GreetConfig:
    val default = GreetConfig(greeting = "Hello", receiver = "World")
    var config = default

  def greetMsg =
    s"${GreetConfig.config.greeting} ${GreetConfig.config.receiver}!"

object DefaultArgs:
  case class GreetConfig(greeting: String, receiver: String)
  object GreetConfig:
    val default = GreetConfig(greeting = "Hello", receiver = "World")

  def greetMsg(config: GreetConfig = GreetConfig.default) =
    s"${config.greeting} ${config.receiver}!"

object GivenVal:
  case class GreetConfig(greeting: String, receiver: String)
  object GreetConfig:
    given default: GreetConfig = GreetConfig("Hello", "World")

  def greetMsg(using g: GreetConfig) = s"${g.greeting} ${g.receiver}"
```

- Skriv kod som testat de olika varianterna ovan. Visa speciellt hur du kan använda default-konfigurationen och därefter ge en konfiguration som skiljer sig från default.
- Vad är för- och nackdelar med de olika varianterna ovan? Diskutera speciellt vilken/vilka lösningar som medger flera lokala konfigurationer utan att de påverkar varandra.
- Förklara vad som händer vid anrop av `summon[GivenVal.GreetConfig]`.
- Vad händer om du försöker framkalla ett givet värde för en typ som inte har något sådant?
- Måste det givna värdet vara unikt?

### 11.2.2 Extrauppgifter; träna mer

**Uppgift 4.** *Kontextparameter och givet värde.* Prova nedan i REPL.

```
1 scala> def add(x: Int)(using y: Int) = x + y
2 scala> add(1)(using 2)
3 scala> add(1)
4 scala> given ngtnam = 42
5 scala> add(1)
```

- Vad blir felmeddelandet på rad 3 ovan?
- Varför fungerar det på rad 5 utan fel?
- Definiera och testa en motsvarande funktion `sub` som kan subtrahera ett givet värde.

### 11.2.3 Fördjupningsuppgifter; utmaningar

**Uppgift 5.** *Varians och typgränser.* Koden nedan är en modell av husdjur med följande innebörd: Husdjur kan vara friska eller sjuka och föds i normalfallet friska. Det kan finnas många katter och hundar, vilka alla är olika slags husdjur.

```
trait Pet(var isHealthy: Boolean = true)
class Cat extends Pet()
class Dog extends Pet()
```

a) Förändra koden nedan så att efterföljande REPL-sats *inte* ger kompileringsfel?

```
case class Box[A](x: A)
```

```
scala> val b: Box[Any] = Box[Cat](Cat())
```

b) Prova nedan i REPL och förklara vad som händer.

```
scala> val v: Vector[Pet] = Vector[Cat](Cat())

scala> val s: Set[Pet] = Set[Cat](Cat())

scala> :settings -explain

scala> val s: Set[Pet] = Set[Cat](Cat())
```

*Ledtråd:* I Scalas standardbibliotek så är ärver Set[T] funktionstypen T => Boolean som är deklarerad kontravariant i sin inparameter.

c) Det ska finnas veterinärer som kan behandla husdjur och göra dem friska. Varför fungerar inte nedan kod? Är det ett kompileringsfel eller körtidsfel?

```
class Vet[-A]:
  def treat(x: A): Unit = x.isHealthy = true
```

d) Inför en typgräns i veterinärens typparametern som åtgärdar felet.

e) Skriv valfri kod som visar 1) att kompilatorn tillåter kattveterinärer att behandla katter men 2) förhindrar att kattveterinärer får behandla godtyckliga husdjur och att 3) en veterinär som har kompetens att behandla godtyckliga husdjur kan behandla både katter och hundar. Förklara varför kompilatorn tillåter/förhindrar detta.

**Uppgift 6.** *Typklasser och kontextparametrar.* I Scala finns möjligheter till avancerad funktionsprogrammering med s.k. **typklasser** (ä.k. *ad hoc polymorfism*). En typklass definierar generella beteenden som fungerar för godtyckliga befintliga typer utan att implementationen av dessa behöver ändras. Vi nosar i denna uppgift på hur kontextuella abstraktioner kan användas för att skapa typklasser i Scala, illustrerat med hjälp av givna ordningarna vid sortering.

Genom att kombinera koncepten givna värden, generiska klasser och kontextparametrar får man möjligheten till ad hoc polymorfism, exemplifierat med typklassen CanCompare nedan, som vi kan få att fungera för befintliga typer *utan* att de behöver ändras. Speciellt så har vi ju inte möjligheten att lägga till metoder på befintliga typer i standardbiblioteket, eftersom det inte är vår egen kod.

a) Vad händer nedan? Vilka rader ger felmeddelande? Varför?

```

1 scala> trait CanCompare[T]:
2     def compare(a: T, b: T): Int
3
4 scala> def sort[T](a: T, b: T)(using cc: CanCompare[T]): (T, T) =
5     if cc.compare(a, b) > 0 then (b, a) else (a, b)
6
7 scala> sort(42, 41)
8
9 scala> given intComparator: CanCompare[Int] with
10     override def compare(a: Int, b: Int): Int = a - b
11
12 scala> sort(42, 41)
13
14 scala> sort(42.0, 41.0)

```

b) Definiera ett givet värde som gör så att sort fungerar för värden av typen Double.

c) Definiera ett givet värde som gör så att sort fungerar för värden av typen String. *Tips:* Du har nytta av de befintliga jämförelseoperatorerna på strängar, men tänk på att compare fortfarande måste returnera ett heltal även vid jämförelse av strängar.

**Uppgift 7. Användning av given ordning.** Vi ska nu skapa en funktion `isSorted` som är generellt användbar genom att göra givna ordningsfunktioner tillgängliga för olika typer. Funktionen `def isSorted(xs: Vector[Int]): Boolean = ???` fungerar bara för samlingar av typen `Vector[Int]`.

Om vi i stället använder `def isSorted(xs: Seq[Int]): Boolean = ???` fungerar den för olika samlingar med heltal, även `Vector` och `List`.

a) Testa nedan funktion i REPL med heltalssekvenser av olika typ.

```
def isSorted(xs: Seq[Int]): Boolean = xs == xs.sorted
```

b) Det blir problem med nedan försök att göra `isSorted` generisk. Hur lyder felmeddelandet? Vad saknas enligt felmeddelandet?

```
scala> def isSorted[T](xs: Seq[T]): Boolean = xs == xs.sorted
```

c) Vi vill gärna att `isSorted` ska fungera för godtyckliga typer `T` som har en ordningsdefinition. Detta kan göras med nedan funktion där den speciella typparametern `[T:Ordering]` betyder att `isSorted` är definierad för alla samlingar där typen `T` har en given ordning `Ordering[T]`. Speciellt gäller detta för alla grundtyperna `Int`, `Double`, `String`, etc., som alla har specifika implementationer av typklassen `Ordering`.

```
def isSorted[T:Ordering](xs: Seq[T]): Boolean = xs == xs.sorted
```

Testa metoden ovan i REPL enligt nedan.

```

1 scala> isSorted(Vector(1,2,3))
2 scala> isSorted(List(1,2,3,1))
3 scala> isSorted(Vector("A","B","C"))
4 scala> isSorted(List("A","B","C","A"))
5 scala> case class Person(firstName: String, familyName: String)
6 scala> val persons = Vector(Person("Kim", "Finkodare"), Person("Voldemort","Fulhackare"))
7 scala> isSorted(persons)

```

Vad ger sista raden för felmeddelande? Varför?

d) *Implicita ordningar*. En typparameter på formen `[T:Ordering]` kallas kontextgräns (eng. *context bound*) och föranleder kompilatorn att automatiskt expandera funktionshuvudet för `isSorted` med en kontextparameter. I stället för att använda `[T:Ordering]` kan vi själva lägga till en kontextparameter som motsvarar kontextgränsen. Då får vi också tillgång till ett namn, här nedan `ord`, på den implicita ordningen och kan använda det namnet i funktionskroppen och anropa metoder som är medlemmar av typklassen `Ordering`. (Namnet på kontextparametern kan också utelämnas, men då får vi istället gå omvägen via inbyggda funktionen `summon[T]` för att be kompilatorn leta upp den givna instansen för den typparameter som ges vid anropet.)

```
def isSorted[T](xs: Seq[T])(using ord: Ordering[T]): Boolean =
  xs.zip(xs.tail).forall(x => ord.lteq(x._1, x._2))
```

Objekt av typen `Ordering` har jämförelsemetoder som t.ex. `lteq` (förk. *less than or equal*) och `gt` (förk. *greater than*).

Det finns givna ordningar för alla grundtyper i standardbiblioteket, alltså t.ex. `Ordering[Int]`, `Ordering[String]`, etc. Testa så att kompilatorn hittar ordningen för samlingar med värden av några grundtyper. Kontrollera även enligt nedan att det fortfarande blir problem för egendefinierade klasser, t.ex. `Person` (detta ska vi råda bot på i uppgift 8).

```
1 scala> isSorted(Vector(1,2,3))
2 scala> isSorted(Array(1,2,3,1))
3 scala> isSorted(Vector("A","B","C"))
4 scala> isSorted(List("A","B","C","A"))
5 scala> class Person(firstName: String, familyName: String)
6 scala> val persons = Vector(Person("Kim", "Finkodare"), Person("Robin","Fulhack"))
7 scala> isSorted(persons)
```

e) *Importera implicita ordningsoperatorer från en Ordering*. Om man gör import på ett `Ordering`-objekt får man tillgång till implicita konverteringar som gör att jämförelseoperatorerna fungerar. Testa nedan variant av `isSorted` på olika sekvenstyper och verifiera att `<=`, `>`, etc., nu fungerar enligt nedan.

```
def isSorted[T](xs: Seq[T])(given ord: Ordering[T]): Boolean = {
  import ord._
  xs.zip(xs.tail).forall(x => x._1 <= x._2)
}
```

### Uppgift 8. Skapa egen implicit ordning med Ordering.

a) Ett sätt att skapa en egen, specialanpassad ordning för dina egna klasser är att mappa dina objekt till typer som redan har en implicit ordning. Med hjälp av metoden `by` i objektet `scala.math.Ordering` kan man skapa ordningar genom att bifoga en funktion `T => S` där `T` är typen för de objekt du vill ordna och `S` är någon annan typ, t.ex. `String` eller `Int`, där det redan finns en given ordning.

```
1 scala> case class Team(name: String, rank: Int)
2 scala> val xs =
3     Vector(Team("fnatic", 1499), Team("nlp", 1473), Team("lumi", 1601))
4 scala> xs.sorted // Hur lyder felmeddelandet? Varför blir det fel?
5 scala> val teamNameOrdering: Ordering[Team] = Ordering.by(t => t.name)
6 scala> xs.sorted(using teamNameOrdering) //explicit ordning
7 scala> given Ordering[Team] = Ordering.by(t => t.rank)
8 scala> xs.sorted // Varför funkar det nu?
```



b) Vill man sortera i omvänd ordning kan man använda `Ordering.fromLessThan` som tar en funktion  $(T, T) \Rightarrow \text{Boolean}$  vilken ska ge **true** om första parametern ska komma före, annars **false**. Om vi vill sortera efter rank i omvänd ordning kan vi göra så här:

```
1 scala> val highestRankFirst: Ordering[Team] =
2     Ordering.fromLessThan((t1, t2) => t1.rank > t2.rank)
3 scala> xs.sorted(using highestRankFirst)
```

c) Om du har en case-klass med flera fält och vill ha en fördefinierad implicit sorteringsordning samt även erbjuda en alternativ sorteringsordning, så kan du placera en default ordningsdefinition i ett kompanjonsobjekt; detta är nämligen ett av de ställen där kompilatorn söker sist efter eventuella implicita värden innan den ger upp att leta.

```
case class Team(name: String, rank: Int)
object Team:
  given highestRankFirst: Ordering[Team] =
    Ordering.fromLessThan((t1, t2) => t1.rank > t2.rank)
  val nameOrdering: Ordering[Team] = Ordering.by(t => t.name)
```

```
1 scala> val xs =
2     Vector(Team("fnatic", 1499), Team("nlp", 1473), Team("lumi", 1601))
3 scala> xs.sorted
4 scala> xs.sorted(Team.nameOrdering)
```

d) Det går också med kompanjonsobjektet ovan att få jämförelseoperatorer att fungera med din case-klass, genom att importera medlemmarna i lämpligt ordningsobjekt. Verifiera att så är fallet enligt nedan:

```
1 scala> Team("fnatic",1499) < Team("gurka", 2) // Vilket fel? Varför?
2 scala> import Team.highestRankFirst.given
3 scala> Team("fnatic",1499) < Team("gurka", 2) // Inget fel? Varför?
```

**Uppgift 9. Specialanpassad ordning genom att ärva från *Ordered*** Om det finns en väldefinierad, specifik ordning som man vill ska gälla för sina case-klass-instanser kan man göra den ordnad genom att låta case-klassen mixa in traiten *Ordered* och implementera den abstrakta metoden *compare*. (Detta illustrerar användning av subtypspolymorfism (d.v.s arv) i stället för ad hoc polymorfism med typklasser.)

*Bakgrund:* En trait som används på detta sätt kallas **gränssnitt** (eng. *interface*), och om man *implementerar* ett gränssnitt så uppfyller man ett "kontrakt", som i detta fall innebär att man implementerar det som krävs av ordnade objekt, nämligen att de har en konkret *compare*-metod. Du lär dig mer om gränssnitt i kommande kurser.

a) Implementera case-klassen *Team* så att den är en subtyp till *Ordered* enligt nedan skiss. Högre rankade lag ska komma före lägre rankade lag. Metoden *compare* ska ge ett heltal som är negativt om *this* kommer före *that*, noll om de ordnas lika, annars positivt.

```
case class Team(name: String, rank: Int) extends Ordered[Team]:
  override def compare(that: Team): Int = ???
```

*Tips:* Du kan anropa metoden *compare* på alla grundtyper, t.ex. *Int*, eftersom de implementerar gränssnittet *Ordered*. Genom att negera uttrycket blir ordningen den omvända.

```
scala> -(2.compare(1))
```

b) Testa att din case-klass nu uppfyller det som krävs för att vara ordnad.



```
scala> Team("fnatic",1499) < Team("gurka", 2)
```

c) Diskutera med handledare eller kursare skillnader och likheter mellan gränssnitt och typklasser, med ledning av denna och föregående uppgifter.

## 11.3 Laboration: snake1

### Mål

- ☐ Se mål i förra veckans uppgift. Denna laboration sträcker sig över två veckor. Förra veckans redovisning avsåg arbetsupplägget och pågående utveckling, samt ditt individuella ansvar. Denna vecka ska de färdig lösningarna presenteras av respektive huvudansvarig individ.

### Förberedelser

- ☐ Gör övning context i avsnitt [11.2](#)

#### 11.3.1 Redovisning av grupplabb

1. Förklara kodens övergripande struktur.
2. Beskriv vilka delar du har bidragit till i koden.
3. Ge en kort förklaring av koncept som du tränat på.
4. Redogör för vad du lärt dig om utmaningarna med systemutveckling i grupp.
5. Redogör den återkoppling du fått från granskningar och hur du arbetat med att förbättra läsbarheten under dina stegvisa utvidgningar av din kod.
6. Redogör för hur och vad du återkopplat när du granskat någon annans kod.

## Kapitel 12

# Valfri fördjupning, Projekt

Begrepp som ingår i denna veckas studier:

- ☐ välj valfritt fördjupningsområde
- ☐ påbörja projekt

## 12.1 Projektuppgift

### 12.1.1 Om din avslutande projektuppgift

#### Läs noga kompendium Del 1, kapitel -1 avsnitt "Projektuppgift"!

Några viktiga punkter:

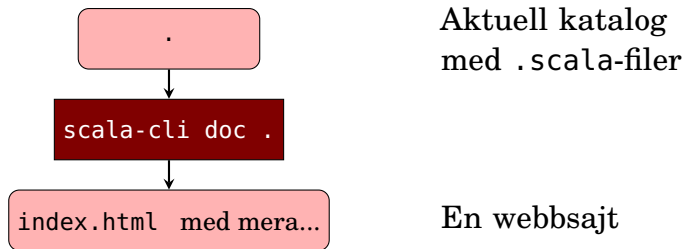
- Mål: skapa ett stort program med många samverkande klasser/moduler.
  - Du väljer själv projektuppgift. Börja redan idag att planera ditt arbete!
  - Projektet går över **två veckor**. Schemalagda labbar i december är obligatoriska för de som har kvar obligatoriska moment.
  - I kompendium Del 2, kapitel 13, finns flera förslag att välja bland, men du kan också definiera ett eget projekt som passar just dig.
  - Inför redovisningen ska du skapa automatiskt genererad dokumentation utifrån relevanta dokumentationskommentarer för minst hälften av dina publika metoder, enligt instruktioner i Appendix E.
  - Redovisning sker i datorsal på schemalagd tid:
    - Förklara hur din kod fungerar.
    - Beskriv framväxten av ditt program.
    - Gå igenom den genererade dokumentationen av din kod.
- 

### 12.1.2 Projektuppgifter

- bank
    - känd domän: skapa bank med transaktionshistorik
    - oföränderlig data tillsammans med tillståndsförändring
  - music
    - skapa ett enkelt kompositionsverktyg som spelar musik
    - sätta sig in i en domänmodell
  - photo
    - en enkel variant av photoshop
    - inblick i enkel matrismatematik
  - egendefinierat projekt
    - Lagom svårt: ej för enkel uppgift, men ta dig inte vatten över huvudet!
    - diskutera med handledare och dokumentera
- 

### 12.1.3 Skapa dokumentation

Scala CLI kan ta dokumentationskommentarerna i källkoden och skapa en webbsajt med dokumentation.



### 12.1.4 Dokumentationskommentarer

För att kod ska bli begriplig för människor är det bra att dokumentera vad den gör. Det finns **tre olika sorters kommentarer**:

```
// Enradskommentarer börjar med dubbla snedstreck
//      men de gäller bara till radslut

/* Flerradskommentarer börjar med
   snedstreck-asterisk
   och slutar med asterisk-snedstreck. */

/** Dokumentationskommentarer placeras före
 *  t.ex. en funktion och berättar vad den gör
 *  och vad eventuella parametrar används till.
 *  Börjar med snedstreck-asterisk-asterisk.
 *  Varje ny kommentarsrad börjar med asterisk.
 *  Avslutas med asterisk-stjärna.
 */
```

Kommentarer påverkar inte hur en maskin exekverar koden, men hjälper människor att använda koden. Läs mer i Appendix E.

---

## 12.2 Övning extra

### Mål

- ☐ Denna veckas övning innehåller valfri fördjupning.
- ☐ Sökning och sortering:
  - Kunna använda inbyggda sökmetoder.
  - Förstå när binärsökning är lämplig och möjlig.
  - Kunna implementera binärsökning.
  - Kunna implementera urvalssortering, både till ny samling och på plats.
- ☐ Trådar och jämlöpande exekvering:
  - Känna till vad en tråd är och kunna förklara begreppet jämlöpande exekvering.
  - Känna till vad metoderna `run` och `start` gör i klassen `Thread`.
  - Kunna skapa och starta en tråd med överskuggad `run`-metod.
  - Kunna skapa ett enkelt program som från två trådar tävlar om att uppdatera en variabel och förklara varför beteendet kan bli oförutsägbart.
  - Kunna använda en `Future` för att köra igång flera parallella beräkningar.
  - Kunna registrera en callback på en `Future` med metoden `onComplete`.

### 12.2.1 Uppgifter om sökning och sortering

**Uppgift 1.** *Tidmätning.* I kommande uppgifter kommer du att ha nytta av funktionen `timed` enligt nedan.

```
def timed[T](code: => T): (T, Long) =
  val now = System.nanoTime
  val result = code
  val elapsed = System.nanoTime - now
  println("\ntime: " + (elapsed / 1e6) + " ms")
  (result, elapsed)
```

a) Klistra in `timed` i REPL och testa så att den fungerar, genom att mäta hur lång tid nedan uttryck tar att exekvera.

```
1 scala> val (v, t1) = timed{ (1 to 1000000).toVector.reverse }
2 scala> val (s, t2) = timed{ v.toSet }
3 scala> timed{ v.find(_ == 1) }
4 scala> timed{ s.find(_ == 1) }
5 scala> timed{ s.contains(1) }
```



b) Försök förklara skillnaderna i exekveringstid mellan de olika sätten att söka reda på talet 1 i samlingen. Ungefär hur många gånger behöver man använda `contains` på heltalsmängden `s` för att det ska löna sig att skapa `s` i stället för att linjärsöka i `v` med `find` i ovan exempel?

**Uppgift 2.** *Sökning med inbyggda sökmetoder.*

a) *Linjärsökning framifrån med `indexOfSlice`.* Studera dokumentationen för Scalas samlingsmetod `indexOfSlice`<sup>1</sup> och skriv 8 olika uttryck i REPL som, både med en sträng och

<sup>1</sup>[docs.scala-lang.org/overviews/collections/seqs.html](https://docs.scala-lang.org/overviews/collections/seqs.html)

med en vektor med heltal, provar 4 olika fall: (1) finns i början, (2) finns någonstans i mitten, (3) finns i slutet, samt (4) finns ej.

b) *Linjärsökning bakifrån med `lastIndexOfSlice`*. Studera dokumentationen för Scalas samlingsmetod `lastIndexOfSlice`<sup>2</sup> och skriv 8 olika uttryck i REPL som, både med en sträng och med en vektor med heltal, provar 4 olika fall: (1) finns i början, (2) finns någonstans i mitten, (3) finns i slutet, samt (4) finns ej.

c) *Sökning med inbyggd binärsökning*. Om en samling är sorterad kan man utnyttja detta för att göra snabbare sökning. Vid **binärsökning** (eng. *binary search*)<sup>3</sup> börjar man på mitten och kollar vilken halva att söka vidare i; sedan delar man upp denna halva på mitten och kollar vilken fjärdedel att söka vidare i, etc.

I objektet `scala.collection.Searching`<sup>4</sup> finns en metod `search` som, om den importeras, erbjuder binärsökning för alla sorterade sekvenssamlingar. Om samlingen är sorterad ger den ett objekt av case-klassen `Found` som innehåller indexet för platsen där elementet först hittats; alternativt om det som eftersöks ej finns, ges ett objekt av case-klassen `InsertionPoint` som innehåller indexet där elementet borde ha varit placerad om det funnits i samlingen. Observera att om samlingen inte är sorterad är resultatet "odefinierat", d.v.s. något returneras men det är *inte* att lita på; man måste alltså först sortera samlingen eller vara helt säker på att den är sorterad.

Undersök hur `search` fungerar genom att förklara vad som händer nedan. Vilken är snabbast av `lin` och `bin` nedan? Använd `timed` från uppgift 1.

```
1 scala> val udda = (1 to 1000000 by 2).toVector
2 scala> import scala.collection.Searching._
3 scala> udda.search(udda.last)
4 scala> udda.search(udda.last + 1)
5 scala> udda.reverse.search(udda(0))
6 scala> def lin(x: Int, xs: Seq[Int]) = xs.indexOf(x)
7 scala> def bin(x: Int, xs: Seq[Int]) = xs.search(x) match
8     case Found(i) => i
9     case InsertionPoint(i) => -i
10 scala> timed{ lin(udda.last, udda) }
11 scala> timed{ bin(udda.last, udda) }
```

d) Om en samling innehåller  $n$  element, hur många jämförelser behövs då vid binärsökning i värsta fall? *Tips*: Läs om algoritmen på Wikipedia<sup>3</sup>.

**Uppgift 3.** Sök bland LTH:s kurser med linjärsökning.

a) Via denna URL kan du ladda ner en tab-separerad lista med alla kurser som ges på LTH under innevarande läsår: <http://cs.lth.se/pgk/kurser>  
Vilken data finns i filen? Du kan undersöka detta t.ex. med:

```
scala> import scala.io.Source.fromURL
scala> val url = "https://fileadmin.cs.lth.se/pgk/lthkurser201819.txt"
scala> val data = fromURL(url,"UTF-8").getLines.mkString("\n")
```

b) Klistra in objektet `courses` på sidan 405 i REPL.<sup>5</sup> Vad gör koden? Hur många kurser innehåller `courses.lth`?

<sup>2</sup>[docs.scala-lang.org/overviews/collections/seqs.html](https://docs.scala-lang.org/overviews/collections/seqs.html)

<sup>3</sup>[en.wikipedia.org/wiki/Binary\\_search\\_algorithm](https://en.wikipedia.org/wiki/Binary_search_algorithm)

<sup>4</sup>[http://www.scala-lang.org/api/current/scala/collection/Searching\\$.html](http://www.scala-lang.org/api/current/scala/collection/Searching$.html)

<sup>5</sup>Du kan ladda ner koden från:

[github.com/unduniversity/introprog/tree/master/compendium/examples/lth-courses/courses.scala](https://github.com/unduniversity/introprog/tree/master/compendium/examples/lth-courses/courses.scala)

```

object courses:
  def download(): Vector[Course] =
    val url = "https://fileadmin.cs.lth.se/pgk/lthkurser201819.txt"
    val lines = scala.io.Source.fromURL(url, "UTF-8").getLines.toVector
    lines.drop(1).map(s => Course.fromLine(s, '\t'))

  lazy val lth: Vector[Course] = download()

  case class Course(
    code: String,
    nameSv: String,
    nameEn: String,
    credits: Double,
    level: String
  )

object Course:
  def fromLine(line: String, separator: Char): Course =
    import scala.util.Try
    val xs = line.split(separator).toSeq
    def str(i: Int): String = Try(xs(i)).getOrElse("")
    def num(i: Int): Double = Try(xs(i).toDouble).getOrElse(0.0)
    Course(str(0), str(1), str(2), num(3), str(4))

```

**Figur 12.1:** Kod för att ladda ner data om alla kurser på LTH.

c) *Linjärsökning med find.* Teknologen Oddput Clementina vill gå första bästa datavetenskapskurs som är på G2-nivå. Hjälp Oddput med att söka upp första förekommande kurs genom linjärsökning med samlingsmetoden `find`. Kurskoder vid datavetenskap börjar på EDA eller ETS<sup>6</sup>. *Tips:* Du har nytta av att definiera predikatet `def isCS(s: String): Boolean` som i sin tur lämpligen nyttjar strängmetoden `startsWith`.

d) *Implementera linjärsökning.* Som träning ska du nu implementera en egen linjärsökningsfunktion med signaturen:

```
def linearSearch[T](xs: Seq[T])(p: T => Boolean): Int = ???
```

Funktionen ska ta en sekvenssamling `xs` och ett predikat `p` som är en funktion som tar ett element och returnerar ett booleskt värde. Typen `Seq` är supertyp till alla sekvenssamlingar, så om vi använder den som parametertyp för parametern `xs` så fungerar funktionen för `Vector`, `Array`, `List`, etc. Genom typparametern `T` blir funktionen generisk och fungerar för godtycklig typ. Funktionen `p` ska ge `true` om parametern är ett eftersökt element. Funktionen `linearSearch` ska returnera index för första hittade elementet i `xs` där `p` gäller. Om det inte finns något element som uppfyller predikatet ska `-1` returneras. Skriv först pseudokod för funktionen med penna och papper. Du ska använda `while`.

e) Implementera en funktion `def rndCode: String` som genererar slumpmässiga kurskoder som består av 4 bokstäver mellan A och Z följt av 2 siffror mellan 0 och 9. *Tips:* Använd REPL i kombination med en editor för att stegvis skapa och testa hjälpfunktioner som löser lämpliga delproblem.

f) Använd `rndCode` från föregående deluppgift för att fylla en vektor kallad `xs` med en halv miljon slumpmässiga kurskoder. För varje slumpkod i `xs` sök med din funktion `linearSearch`

<sup>6</sup>Detta är en förenklad bild av LTH:s kurskodnamnsystem. Några kurser från EIT-institutionen kommer att slinka med, men det bortser vi ifrån i denna uppgift.



efter index i vektorn `courses.lth` från deluppgift b. Mät totala tiden för de 500000 linjärsökningarna med hjälp av funktionen `timed` från uppgift 1. Hur många av de slumpmässiga kurskoderna hittades bland de verkliga kurskoderna på LTH?

g) Hur kan du implementera `linearSearch` med den inbyggda samlingsmetoden `indexWhere`?

**Uppgift 4.** Sök bland LTH:s kurser med binärsökning. Sökalgoritmen `BINSEARCH` kan formuleras med nedan pseudokod:

```

Indata : En växande sorterad sekvens  $xs$  med  $n$  heltal och
           ett eftersökt heltal  $key$ 
Utdata : Ett heltal  $i \geq 0$  som anger platsen där  $x$  finns, eller ett negativt tal  $i$  där  $-i$ 
           motsvarar platsen där  $x$  ska sättas in i sorterad ordning om  $x$  ej finns i
           samlingen.
1 sätt intervallet  $(low, high)$  till  $(0, n - 1)$ 
2  $found \leftarrow \text{false}$ 
3  $mid \leftarrow -1$ 
4 while  $low \leq high$  and not  $found$  do
5    $mid \leftarrow$  platsen mitt emellan  $low$  och  $high$ 
6   if  $xs(mid) == key$  then
7      $found \leftarrow \text{true}$ 
8   else
9     if  $xs(mid) < key$  then
10       $low \leftarrow mid + 1$ 
11    else
12       $high \leftarrow mid - 1$ 
13    end
14  end
15 end
16 if  $found$  then
17    $mid$ 
18 else
19    $-(low + 1)$ 
20 end

```

a) Prova algoritmen ovan med penna och papper på en sorterad sekvens med mindre än 10 heltal. Prova om algoritmen fungerar med ett jämnt antal tal, ett udda antal tal, en sekvens med ett heltal och en tom sekvens. Prova både om talet du letar efter finns och om det inte finns.

b) Implementera binärsökning i en funktion med signaturen

```
def binarySearch(xs: Seq[String], key: String): Int = ???
```

och testa i REPL för olika fall. Vad händer om sekvensen inte är sorterad?

c) Använd `binarySearch` för att leta efter LTH-kurser enligt nedan. Använd `rndCode`, `timed` och `courses` från tidigare uppgifter.

```
def binarySearch(xs: Seq[String], key: String): Int = ???
```

```
val lthCodesSorted = courses.lth.map(_.code).sorted
```

```
val xs = Vector.fill(500000)(rndCode)
```

```
val (_, elapsedBin) =
```

```

    timed{xs.map(x => binarySearch(lthCodesSorted, x))}
val (_, elapsedLin) =
    timed{xs.map(x => linearSearch(lthCodesSorted)(_ == x))}
    println(elapsedLin / elapsedBin)

```

d) Hur mycket snabbare blev binärsökningen jämfört med linjärsökningen?<sup>7</sup>

**Uppgift 5. Insättningssortering.** Implementera sortering av en heltalssekvens till en sekvens med **insättningssortering** (eng. *insertion sort*) i en funktion med följande signatur:

```
def insertionSort(xs: Seq[Int]): Seq[Int] = ???
```

*Lösningssidé:* Skapa en ny, tom sekvens som ska bli vårt sorterade resultat. För varje element i den osorterade sekvensen: Sätt in det på rätt plats i den nya sorterade sekvensen.

a) *Pseudokod:* Kör nedan pseudokod med papper och penna t.ex. på sekvensen 5 1 4 3 2  
1. Rita minnessituationen efter varje runda i loopen. Här använder vi internt i funktionen föränderliga ArrayBuffer som är snabb på insättning och avslutar med toVector så att vi lämnar ifrån oss en oföränderlig sekvens.

```

1 result ← en ny, tom ArrayBuffer
2 foreach element e in xs do
3   | pos ← leta upp rätt position i result
4   | stoppa in e på plats pos i result
5 end
6 result.toVector

```

b) Implementera insertionSort. Använd en **while**-loop för att implementera rad 3 i pseudokoden. Sök upp dokumentationen för metoden insert på ArrayBuffer. Testa insert på ArrayBuffer i REPL och verifiera att den kan användas för att stoppa in på slutet på den "oanvända" positionen som är precis efter sista positionen. Vad händer om man gör insert på positionen size + 2?

Klistra in din implementation av insertionSort i REPL och testa så att allt fungerar:

```

1 scala> insertionSort(Vector())
2 res0: Seq[Int] = Vector()
3
4 scala> insertionSort(Vector(42))
5 res1: Seq[Int] = Vector(42)
6
7 scala> insertionSort(Vector(1,2,3))
8 res2: Seq[Int] = Vector(1, 2, 3)
9
10 scala> insertionSort(Vector(5,1,4,3,2,1))
11 res3: Seq[Int] = Vector(1, 1, 2, 3, 4, 5)

```

**Uppgift 6. Sortering på plats.** Implementera sortering på plats (eng. *in-place*) i en Array[String] med urvalssortering (eng. *selection sort*)

*Lösningssidé:* För alla index *i*: sök *minIndex* för "minsta" strängen från plats *i* till sista plats och byt plats mellan strängarna på plats *i* och plats *minIndex*. Se även animering här: [sv.wikipedia.org/wiki/Urvalssortering](http://sv.wikipedia.org/wiki/Urvalssortering)


<sup>7</sup>Vid en körning på en i7-4970K med 4.0GHz tog elapsedLin cirka 3000 ms och elapsedBin cirka 60 ms. Binärsökning var alltså i detta fall ungefär 50 gånger snabbare än linjärsökning.

Implementera enligt nedan skiss. *Tips:* Du har nytta av en modifierad variant av lösningen till uppgift 13 i kapitel 2.


```
def selectionSortInPlace(xs: Array[String]): Unit =  
  def indexOfMin(startFrom: Int): Int = ???  
  def swapIndex(i1: Int, i2: Int): Unit = ???  
  for i <- 0 to xs.size - 1 do swapIndex(i, indexOfMin(i))
```

**Uppgift 7.** *Undersök om en sekvens är sorterad.* Ett enkelt och lättläst sätt att undersöka om en sekvens är sorterad visas nedan.

```
1 scala> def isSorted(xs: Vector[Int]): Boolean = xs == xs.sorted
```

-  a) Om  $xs$  har  $10^6$  element, hur många jämförelser kommer i värsta fall att ske med `isSorted` enligt ovan? Metoden `sorted` använder algoritmen Timsort<sup>8</sup>. Sök upp antalet jämförelser i värstafallet på Wikipedia.

Denna lösning är dock relativt långsam för stora samlingar. Man behöver ju inte först sortera för att avgöra om det är sorterat (om man inte ändå hade tänkt sortera av andra skäl), det räcker att kolla att elementen är i växande ordning.

- b) Implementera en effektivare variant av `isSorted` som använder en **while**-sats och kollar att elementen är i växande ordning. Din algoritm ska sluta söka så fort osorterade element hittats.
-  c) Vad blir antalet jämförelser i värstafallet med metoden i deluppgift b om du har  $n$  element?
- d) Man kan kolla om en sekvens är sorterad med det listiga tricket att först zippa sekvensen med sin egen svans och sedan kolla om alla element-par uppfyller sorteringskriteriet, alltså `xs.zip(xs.tail).forall(???)` där `???` byts ut mot lämpligt predikat. Vilken typ har 2-tupeln `xs.zip(xs.tail)` om `xs` är av typen `Vector[Int]`? Implementera `isSorted` med detta listiga trick.

**Uppgift 8.** *Insättningssortering på plats.* Implementera och testa sortering på plats i en array med heltal med <sup>9</sup>.

Implementera och testa funktionen nedan i Scala med följande signatur:

```
def insertionSort(xs: Array[Int]): Unit
```

Placera metoden i ett objekt med lämpligt namn, samt skapa ett huvudprogram med testkod. Kompilera och kör från terminalen. Börja med att skriva sorteringsalgoritmen i pseudokod.

<sup>8</sup>[stackoverflow.com/questions/14146990/what-algorithm-is-used-by-the-scala-library-method-vector-sorted](https://stackoverflow.com/questions/14146990/what-algorithm-is-used-by-the-scala-library-method-vector-sorted)

<sup>9</sup>[en.wikipedia.org/wiki/Insertion\\_sort](https://en.wikipedia.org/wiki/Insertion_sort)

**Uppgift 9.** *Sortering till ny sekvens med urvalssortering.* Implementera och testa sortering till ny sekvens med urvalssortering<sup>10</sup> i Scala, enligt nedan skiss. Du har nytta av lösningen till uppgift 13 i kapitel 2.

```
def selectionSort(xs: Seq[String]): Seq[String] =
  def indexOfMin(xs: Seq[String]): Int = ???
  val unsorted = xs.toBuffer
  val result = scala.collection.mutable.ArrayBuffer.empty[String]
  /*
  så länge unsorted inte är tom
    minPos = indexOfMin(unsorted)
    elem   = unsorted.remove(minPos)
    result.append(elem)
  */
  result.toVector
end selectionSort
```

### 12.2.2 Uppgifter om trådar och jämlöpande exekvering

**Uppgift 10.** *Trådar.* Klassen `java.lang.Thread` används för att skapa **trådar** med jämlöpande exekvering (eng. *concurrent execution*). På så sätt kan man få olika koddelar att köra samtidigt.

Klassen `Thread` definierar en tom `run`-metod. Vill man att tråden ska göra något vettigt får man överskugga `run` med det man vill ska göras.

En tråd körs igång med metoden `start` och då anropas automatiskt `run`-metoden och tråden exekverar koden i `run` jämlöpande med övriga trådar. Om man anropar `run` direkt blir det *inte* jämlöpande exekvering.

a) Skapa en tråd som gör något som tar lite tid och kör med `run` resp. `start`.

```
1 def zzz = { print("zzzzzz"); Thread.sleep(5000); println(" VAKEN!") }
2 zzz
3 val t2 = new Thread{ override def run = zzz }
4 t2.run; println("Gomorra!")
5 t2.start; println("Gomorra!")
6 t2.start
```

b) Vad händer om man anropar `start` mer än en gång på samma tråd?

c) Skapa två trådar med överskuggade `run`-metoder och kör igång dem samtidigt enligt nedan. Vilken ordning skrivs hälsningarna ut efter rad 3 resp. rad 4 nedan? Förklara vad som händer.

```
1 val g = new Thread{ override def run = for i <- 1 to 100 do print("Gurka ") }
2 val t = new Thread{ override def run = for i <- 1 to 100 do print("Tomat ") }
3 g.run; t.run
4 g.start; t.start
```

d) Använd `Thread.sleep` enligt nedan. Är beteendet helt förutsägbart (deterministiskt)? Förklara vad som händer. Du kan (om du kör Linux) avbryta REPL med `Ctrl+C`<sup>11</sup>.

<sup>10</sup>[en.wikipedia.org/wiki/Selection\\_sort](https://en.wikipedia.org/wiki/Selection_sort)

<sup>11</sup>[stackoverflow.com/questions/6248884/can-i-stop-the-execution-of-an-infinite-loop-in-scala-repl](https://stackoverflow.com/questions/6248884/can-i-stop-the-execution-of-an-infinite-loop-in-scala-repl)

```

1 def ibland(block: => Unit) = new Thread {
2   override def run = while(true) { block; Thread.sleep(600) }
3 }.start
4 ibland(print("zzz ")); ibland(print("snark ")); ibland(println("hej!"))

```

**Uppgift 11.** *Jämlöpande variabeluppdatering.* Skriv klasserna Bank och Kund i en editor och klistra sedan in koden i REPL.

```

class Bank:
  private var _saldo = 0;
  def saldo: Int = _saldo
  def sättIn(): Unit = _saldo += 1
  def taUt(): Unit = _saldo -= 1
end Bank

class Kund(bank: Bank):
  def slösaSpara(): Unit =
    bank.taUt()
    Thread.sleep(1)
    bank.sättIn()
  end slösaSpara
end Kund

```

a) Använd funktionen ibland från föregående uppgift och kör nedan rader i REPL. Resultatet av jämlöpande variabeluppdatering blir här heltokigt och leder till mycket upprörda bankkunder och -ägare. Förklara vad som händer.

```

1 val bank = new Bank
2 println(bank.saldo)
3 bank.sättIn()
4 println(bank.saldo)
5 bank.taUt()
6 println(bank.saldo)
7
8 val bamse = new Kund(bank)
9 val skutt = new Kund(bank)
10
11 bamse.slösaSpara()
12 skutt.slösaSpara()
13 println(bank.saldo)
14
15 def ofta(block: => Unit) = new Thread { // varje millisekund
16   override def run = while true do { block; Thread.sleep(1)}
17 }.start
18
19 ofta(bamse.slösaSpara()); ofta(skutt.slösaSpara())
20
21 def ibland(block: => Unit) = new Thread { // varje 600 ms
22   override def run = while(true) do { block; Thread.sleep(600) }
23 }.start
24
25 ibland(println(bank.saldo))

```

**Uppgift 12.** *Trådsäkra AtomicInteger.* Det finns stöd i JVM för att åstadkomma uppdateringar som inte kan avbrytas av andra trådar under pågående minnesskrivning. En

operation som inte kan avbrytas kallas **atomär** (eng. *atomic*). Studera dokumentationen för `AtomicInteger`<sup>12</sup> och prova nedan kod. Förklara vad som händer.

Använd funktionerna ofta och ibland från tidigare uppgifter.

```
class SäkerBank:
  import java.util.concurrent.atomic.AtomicInteger
  private var _saldo = new AtomicInteger
  def saldo: Int = _saldo.get
  def sättIn(): Unit = _saldo.incrementAndGet()
  def taUt(): Unit = _saldo.decrementAndGet()
end SäkerBank

class SäkerKund(bank: SäkerBank):
  def slösaSpara =
    bank.taUt()
    Thread.sleep(1)
    bank.sättIn()
  end slösaSpara
end SäkerKund
```

```
1 val sb = new SäkerBank
2 val farmor = new SäkerKund(sb)
3 val vargen = new SäkerKund(sb)
4
5 ofta(farmor.slösaSpara); ofta(vargen.slösaSpara)
6
7 ibland(println(sb.saldo))
```

**Uppgift 13.** *Jämlöpande exekvering med `scala.concurrent.Future`.* Att skapa och hålla reda på trådar kan bli ganska omständligt och knepigt att få rätt på. Med hjälp av `scala.concurrent.Future` kan man på ett enklare sätt skapa jämlöpande exekvering.

*Bakgrund:* Med en `Future` skapas jämlöpande exekvering som "under huven" använder ett ramverk som heter Akka<sup>13</sup>, skrivet i Scala och Java. Akka erbjuder automatisk multitrådning med s.k. trådpooler och möjliggör avancerad parallellprogrammering på en hög abstraktionsnivå, där man själv slipper skapa instanser av klassen `Thread`. I stället kan man helt enkelt placera sin kod inramad med `Future{ "körs parallellt" }` efter att man importerat det som behövs.

a) För att skapa jämlöpande exekvering med `Future` behöver man först göra import enligt nedan; då skapas ett exekveringssammanhang med trådpooler redo för användning. Starta om REPL och studera felmeddelandet efter rad 1 nedan. Importera därefter enligt nedan. Vad har `f` för typ?

```
1 scala> concurrent.Future { Thread.sleep(1000); println("En sekund senare!") }
2 scala> import scala.concurrent._
3 scala> import ExecutionContext.Implicits.global
4 scala> val f = Future { Thread.sleep(1000); println("En sekund senare!") }
```

b) Skapa en procedur `printLater` enligt nedan som skriver ut argumentet efter slumpmässig tid. Förklara vad som händer nedan.

```
1 scala> def printLater(a: Any): Unit =
```

<sup>12</sup>[docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html)

<sup>13</sup><http://akka.io/>

```

2 Future { Thread.sleep((math.random() * 10000).toInt); print(a + " ") }
3 scala> (1 to 42).foreach(i => printLater(i)); println("alla är igång!")

```

c) Skapa enligt nedan en Future som räknar ut hur många siffror det är i ett väldigt stort tal. Med onComplete kan man ange vad som ska göras när den tunga beräkningen är färdig; detta kallas att "registrera en callback". Vilken returtyp har big? Hur många siffror har det stora talet? Vad har r för typ? Justera argumentet till big om du inte orkar vänta på resultatet...

```

1 scala> BigInt(10).pow(100)
2 scala> BigInt(10).pow(100).toString.size
3 scala> def big(n: Int) = Future { BigInt(n).pow(n).toString.size }
4 scala> big(1234567).onComplete{r => println(r + " siffror")}

```

d) Den stora vinsten med Future är att man kan köra vidare under tiden, varför anropet av Future kallas **icke-blockerande** (eng. *non-blocking*). Det händer ibland att man ändå vill blockera exekveringen i väntan på ett resultat. Man kan då använda objektet `scala.concurrentAwait` och dess metod `result` enligt nedan. Använd big från föregående uppgift och gör en blockerande väntan på resultatet enligt nedan. Vad händer? Vad händer om du väntar för kort tid?

```

1 scala> import scala.concurrent.duration._
2 scala> Await.result(big(1234567), 20.seconds)

```

**Uppgift 14.** Använda Future för att göra flera saker samtidigt. I denna uppgift ska du ladda ner webbsidor parallellt med hjälp av Future, så att en nedladdning kan avslutas under tiden en annan dröjer.

a) Koden för en minimal webbsida ser ut som nedan. Du kan beskåda sidan här: <http://fileadmin.cs.lth.se/pgk/mini.html> eller skriva in nedan kod i en fil som heter något som slutar på .html och öppna filen i din webbläsare.

```

<!DOCTYPE html>
<html>
<body>
HELLO WORLD!
</body>
</html>

```

b) För att simulera slöa webbservrar kan man ladda ner en sida via sajten <http://deelay.me/>

Ladda ner ovan sida med 2 sekunders fördröjning:

<http://deelay.me/2000/http://fileadmin.cs.lth.se/pgk/mini.html>

c) Man kan ladda ner webbsidor med `scala.io.Source`. Vad händer nedan? Försök, med ledning av hur delay beräknas, uppskatta hur lång tid du måste vänta i medeltal, i bästa fall, respektive värsta fall, innan du kan se första webbsidan i vektorn laddningar nedan?

```

1 scala> def ladda(url: String) = scala.io.Source.fromURL(url).getLines.toVector
2 scala> def slöladda(url: String) =
3     val delay = (math.random() * 1000 + 2000).toInt
4     val delaySite = s"http://deelay.me/$delay/"
5     ladda(delaySite+url)
6     end slöladda

```



```

7 scala> ladda("http://fileadmin.cs.lth.se/pgk/mini.html")
8 scala> def seg = slöadda("http://fileadmin.cs.lth.se/pgk/mini.html")
9 scala> val laddningar = Vector.fill(10)(seg)
10 scala> laddningar(0)

```

d) Innan vi kan köra igång en Future så måste vi, som visats i uppgift 13 importera den underliggande exekveringsmiljön som är redo att parallelisera ditt program i trådar utan att du själv måste skapa dem. Vad händer nedan?

```

1 scala> import scala.concurrent._
2 scala> import ExecutionContext.Implicits.global
3 scala> val f = Future(seg)
4 scala> f // kolla om den är klar annars prova igen senare
5 scala> f

```

e) Ladda indata utan att blockera (eng. *non-blocking input*). Förklara vad som händer nedan.

```

1 scala> val nonBlocking = Future(Vector.fill(10)(seg))
2 scala> nonBlocking // kolla igen senare om ej klar
3 scala> nonBlocking

```

f) Ladda indata separat i olika parallella trådar. Förklara vad som händer nedan. Kör uttrycket på rad 3 nedan upprepade gånger i snabb följd efter varandra med pil-upp+Enter i REPL.

```

1 scala> val para = Vector.fill(10)(Future(seg))
2 scala> para
3 scala> para.map(_._isCompleted)
4 scala> para.map(_._isCompleted) // studera hur de blir färdiga en efter en
5 scala> para(0)

```

g) Registrera en callback med metoden onComplete. Förklara vad som händer nedan.

```

1 scala> val action = Vector.fill(10)(Future(seg))
2 scala> action(0).onComplete(xs => println(s"ready:$xs"))
3 scala> // vänta tills laddning på plats 0 är klar

```

h) Registrera en callback för felhantering i händelse av undantag med metoden onFailure. Förklara vad som händer nedan.

```

1 scala> def lycka = { Thread.sleep(3000); println(":)") }
2 scala> def olycka = { Thread.sleep(3000); 42 / 0; lycka }
3 scala> Future(lycka).onFailure{ case e => println(s":( $e") }
4 scala> Future(olycka).onFailure{ case e => println(s":( $e") }

```

**Uppgift 15.** Räkna ut stora primtal parallellt genom att använda nedan funktioner. Implementera isPrime enligt pseudokod från den engelska wikipediasidan om primtalstest<sup>14</sup> med den s.k. ”naiva algoritmen”. Räkna ut 10 st slumpvisa primtal med 16 siffror vardera. Gör beräkningarna parallellt med hjälp av Future.

```

def isPrime(n: BigInt): Boolean = ???

def nextPrime(start: BigInt): BigInt =
  var i = start

```

<sup>14</sup>[en.wikipedia.org/wiki/Primality\\_test](https://en.wikipedia.org/wiki/Primality_test)

```

while !isPrime(i) do i += 1
i
end nextPrime

def randomBigInt(nDigits: Int): BigInt =
  def rndChar = ('0' + (math.random() * 10).toInt).toChar
  val str = Array.fill(nDigits)(rndChar).mkString
  BigInt(str)
randomBigInt

```



### Uppgift 16. Svara på teorifrågor.

- Vad är en tråd?
- Hur skapar man en tråd med klassen Thread?
- Hur startar man en tråd?
- Vilka problem kan man råka ut för om man uppdaterar samma resurs i flera olika trådar?
- Vad innebär det att kod är *trådsäker*?
- Nämn några fördelar med att använda Future jämfört med att använda trådar direkt.

**Uppgift 17. Klasser med atomär uppdatering.** Läs om och testa klasserna AtomicBoolean, AtomicDouble och AtomicReference för atomär uppdatering i paketet `java.util.concurrent.atomic`.

Använd några av dessa tillsammans med `scala.concurrent.Future`.

### Uppgift 18. Skapa din egen multitrådade webserver.

- Skriv in<sup>15</sup> nedan kod i en editor och spara i en fil med namn `webserver.scala` och kompilera och kör med `scala-cli run webserver.scala` och beskriv vad som händer när du med din webbläsare surfar till adressen:

<http://localhost:8089/abbasillen>

```

1 package webserver
2
3 import java.net.{ServerSocket, Socket}
4 import java.io.OutputStream
5 import java.util.Scanner
6 import scala.util.Try
7
8 object html:
9   def page(body: String): String = //minimal web page
10     s"""<!DOCTYPE html>
11       |<html>
12       |<head><meta charset="UTF-8"><title>Min Sörver</title></head>
13       |<body>
14       |$body
15       |</body>
16       |</html>
17       """.stripMargin
18
19   def header(length: Int): String = //standardized header of reply
20     s"HTTP/1.0 200 OK\nContent-length: $length\n" +
21       "Content-type: text/html\n\n"

```

<sup>15</sup>Eller ladda ner här: [github.com/lunduniversity/introprog/blob/master/compendium/examples/simple-web-server/webserver.scala](https://github.com/lunduniversity/introprog/blob/master/compendium/examples/simple-web-server/webserver.scala)

```

22
23
24 object start:
25   def handleRequest(cmd: String, uri: String, socket: Socket): Unit =
26     val os = socket.getOutputStream
27     val response = html.page("Baklänges: " + uri.reverse)
28     os.write(html.header(response.size).getBytes("UTF-8"))
29     os.write(response.getBytes("UTF-8"))
30     os.close
31     socket.close
32   end handleRequest
33
34   def serverLoop(server: ServerSocket): Unit =
35     println(s"http://localhost:${server.getLocalPort}/hej")
36     while true do
37       Try {
38         var socket = server.accept // blocks thread until connect
39         val scan = new Scanner(socket.getInputStream, "UTF-8")
40         val (cmd, uri) = (scan.next, scan.next)
41         println(s"Request: $cmd $uri")
42         handleRequest(cmd, uri, socket)
43       }.recover{ case e: Throwable => s"Connection failed: $e" }
44
45   def main(args: Array[String]) =
46     val port = Try{ args(0).toInt }.getOrElse(8089)
47     serverLoop(new ServerSocket(port))

```

b) Du ska nu skapa en webserver som gör något lite mer intressant. Den ska svara med det 13:e Fibonacci-talet<sup>16</sup> om du surfar till <http://localhost:8089/fib/13>. Spara din webserver från föregående deluppgift under det nya namnet `fibserver.scala` och använd koden nedan och lägg till och ändra så att din server kan svara med Fibonacci-tal. Vi börjar med att räkna ut Fibonacci-tal i funktionen `compute.fib` nedan på ett onödigt processorkrävande sätt med exponentiell tidskomplexitet så att webbservern verkligen får jobba, för att i senare deluppgifter implementera `compute.fib` med linjär tidskomplexitet och därmed undvika onödig planetuppvärmning.

```

// lägg till nedan i webserver.scala från
// https://github.com/lunduniversity/introprog/blob/master/compendium/examples/simple-web-server/webserver.scala

object compute:
  def fib(n: BigInt): BigInt =
    if n < 0 then 0 else
      if n == 1 || n == 2 then 1
      else fib(n - 1) + fib(n - 2)
  end fib
end compute

def fibResponse(num: String) =
  num.toIntOption match
    case Some(n) => html.page(s"fib($n) == " + compute.fib(n))
    case None    => html.page(s"FEL: skriv ett heltal, inte $num")

def errorResponse(uri:String) = html.page(s"Error: $uri </br> use /fib/heltal")

// ändra handleRequest i start i webserver.scala till
def handleRequest(cmd: String, uri: String, socket: Socket): Unit =
  val os = socket.getOutputStream
  val afterSlash = uri.toString.drop(1) // skip initial slash
  println(s"afterSlash:$afterSlash")

```

<sup>16</sup><https://sv.wikipedia.org/wiki/Fibonacci>

```

val response: String =
  if afterSlash.startsWith("fib/") then fibResponse(afterSlash.stripPrefix("fib/"))
  else ErrorResponse(uri)
os.write(html.header(response.size).getBytes("UTF-8"))
os.write(response.getBytes("UTF-8"))
os.close
socket.close
end handleRequest

```

Kör i terminalen med `scala-cli run webserver.scala` och beskriv vad som händer i din webbläsare när du surfar till servern.

- c) Surfa efter flera stora Fibonacci-tal samtidigt i olika flikar i din browser. Hur märks det att servern bara kör i en enda tråd?
- d) Gör din server multitrådad med hjälp av den nya server-loopen nedan.

```

import scala.concurrent._
import ExecutionContext.Implicits.global

def serverLoop(server: ServerSocket): Unit = {
  println(s"http://localhost:${server.getLocalPort}/hej")
  while (true) {
    Try {
      var socket = server.accept // blocks thread until connect
      val scan = new Scanner(socket.getInputStream, "UTF-8")
      val (cmd, uri) = (scan.next, scan.next)
      println(s"Request: $cmd $uri")
      Future { handleRequest(cmd, uri, socket) }.onFailure {
        case e => println(s"Request failed: $e")
      }
    }.recover { case e: Throwable => s"Connection failed: $e" }
  }
}

```

- e) Surfa efter flera stora Fibonacci-tal samtidigt i olika flikar i din browser. Hur märks det att servern är multitrådad?
- f) Det är onödigt att räkna ut samma Fibonacci-tal flera gånger. Med hjälp av en cache i form av en föränderlig Map kan du spara undan redan uträknade värden. Det funkar dock inte med en vanlig `scala.collection.mutable.Map` i vår multitrådade webserver, eftersom den inte är **trådsäker** (eng. *thread-safe*). Med trådosäkra föränderliga datastrukturer blir det samma besvärliga beteende som i uppgift 11.

Du ska i stället använda `java.util.concurrent.ConcurrentHashMap`. Sök upp dokumentationen för `ConcurrentHashMap` och försök förstå koden nedan. Hur fungerar metoderna `containsKey`, `put` och `get`?

```

object compute {
  import java.util.concurrent.ConcurrentHashMap
  val memcache = new ConcurrentHashMap[BigInt, BigInt]

  def fib(n: BigInt): BigInt =
    if (memcache.containsKey(n)) {
      println("CACHE HIT!!! no need to compute: " + n)
      memcache.get(n)
    } else {
      println("cache miss :( must compute fib: " + n)
      val f = fastFib(n)
    }
}

```

```

    memcache.put(n, f)
  f
}

private def fastFib(n: BigInt): BigInt = {
  if (n < 0) 0 else
  if (n == 1 || n == 2) 1
  else fib(n - 1) + fib(n - 2)
}
}

```

g) Använd ovan fib-objekt i en ny version av din webserver. Spara den i en ny kodfil med namnet `fibserver-memcached.scala`. Undersök hur snabbt det går med stora Fibonacci-tal med den nya varianten. Hur stora tal kan du räkna ut? Kan servern fortsätta efter överflödad stack? Förklara varför.

h) Nu när vi kan få väldigt stora Fibonacci-tal kan det vara användbart att stoppa in radbrytningar på webbsidan. Html-taggen `<br>` ger en radbrytning.

```

def insertBreak(s: String, n: Int = 80): String = {
  if (s.size < n) s
  else s.take(n) + "<br>" + insertBreak(s.drop(n), n)
}

```

Använd den rekursiva funktionen ovan för att pilla in radbrytningstaggar på var  $n$ :te position i långa strängar. Testa hur det ser ut på webbsidan med ovan funktion när din server svarar med väldigt stora tal.

i) Vi ska nu använda det större heap-minnet i stället för stack-minnet och därmed inte begränsas av stackens max-storlek. Skriv om `fastFib` så att den använder en `while`-sats i stället för ett rekursivt anrop. Denna uppgift är ganska klurig, men om du kör fast kan du snegla i lösningarna i Appendix för inspiration.

Hur stora tal klarar din server nu? Vad händer med servern när minnet tar slut? Hur kan du skydda servern så att den inte kan hänga sig?

## 12.3 Projektuppgift: bank

### 12.3.1 Fokus

- ☐ Kunna implementera ett helt program efter given specifikation
- ☐ Kunna sätta samman olika delar från olika moduler
- ☐ Förstå hur Java-klasser kan användas i Scala
- ☐ Förstå och bedöma när immutable/mutable såväl som var/val bör användas i större sammanhang
- ☐ Kunna använda sig av kompanjonsobjekt
- ☐ Kunna läsa och skriva till fil
- ☐ Kunna söka i olika datastrukturer på olika sätt

### 12.3.2 Bakgrund

I detta projekt ska du skriva ett program som håller reda på bankkonton och kunder i en bank. Programmet ska utöver att hålla reda på bankens nuvarande tillstånd även föra historik över alla tillståndsändringar. Historiken ska vara så pass detaljerad att det nuvarande tillståndet kan återskapas genom att återuppspela alla ändringar som finns lagrade i historiken.

Programmet ska vara helt textbaserat, man ska alltså interagera med programmet via terminalen där en meny skrivs ut och input görs via tangentbordet.

Du ska skriva större delen av programmet själv, utan någon färdig kod. Programmet ska dock följa de specifikationer som ges i uppgiften, såväl som de objektorienterade principer du lärt dig i kursen.

### 12.3.3 Krav

Kraven för bankapplikationen återfinns här nedan. För att bli godkänd på denna uppgift måste samtliga krav uppfyllas:

- Programmet ska ha följande menyval:
  - 1. Hitta konton för en viss kontoinnehavare med angivet ID.
  - 2. Söka efter kunder på (del av) namn.
  - 3. Sätta in pengar på ett konto.
  - 4. Ta ut pengar på ett konto.
  - 5. Överföra pengar mellan två olika konton.
  - 6. Skapa ett nytt konto.
  - 7. Ta bort ett befintligt konto.
  - 8. Skriva ut bankens alla konton, sorterade i bokstavsordning efter innehavare.
  - 9. Skriva ut historiken över alla ändringar av bankens tillstånd.
  - 10. Återställa banken till tillståndet den hade vid ett givet datum. För enkelhetens skull får du permanent kassera all historik som skapades efter det datum banken återställs till.
  - 11. Avsluta.
- När något av följande sker ska programmet notera det i historiken:

- Pengar sätts in på ett konto.
  - Pengar tas ut från ett konto.
  - Pengar överförs mellan två konton.
  - Ett konto skapas.
  - Ett konto tas bort.
- Historiken ska sparas både i minnet och i en fil.
- Då programmet startas ska det läsa in historikfilen för att återskapa tillståndet som banken hade tidigare.
- Formatet för historikfilen ska vara detsamma som formatet för den bifogade exempelfilen.
- Allt som berör användargränssnittet (såsom utskrifter till terminalen och inläsning från terminalen) ska ske i `BankApplication` eller hjälpklasser till `BankApplication`, inte i någon annan av klasserna som specificeras i uppgiften.
- Alla metoder och attribut ska ha lämplig synlighet, så att interna, förändringsbara delar inte i onödan exponeras.
- Valen av `val/var` och `immutable/mutable` måste vara lämpliga.
- Programmet ska fungera som i de bifogade exemplen på körning av programmet.
- Rimlig felhantering ska finnas. Det är alltså önskvärt att programmet inte kraschar då användaren matar in felaktig input, utan istället säger till användaren att input är ogiltig. Du kan dock anta att historikfilen alltid är i rätt format.
- Programdesignen ska följa de specifikationer som är angivna nedan.
- Det räcker med att banken ska kunna hantera heltal, men detta ska göras med klassen `BigInteger` för att tillåta stora belopp. Om din bank hanterar decimaltal ska detta ske med `BigDecimal` för att undvika avrundningsfel.
- Klassen `BankAccount` ska generera ett unikt kontonummer för varje konto. Dessa ska återställas om bankens tillstånd återställs till ett tidigare datum, d.v.s. att om en återställning av banken tar bort ett konto så ska dess kontonummer återigen bli tillgängligt.
- Det enda sättet att förändra tillståndet för en `Bank` ska vara (förutom att anropa `returnToState`) att anropa `doEvent` med en `BankEvent` som beskriver tillståndsförändringen. Vid en första anblick kan detta kan verka lite väl bökit, men när ändringshistoriken ska implementeras kommer det vara till stor hjälp att det finns en `BankEvent` som representerar varje ändring.
- Du ska inför redovisningen generera automatisk dokumentation baserat på dokumentationskommentarer enligt instruktioner i Appendix E. Du ska skriva relevanta dokumentationskommentarer för minst hälften av dina publika metoder. Det är ofta användbart att skriva dokumentationskommentarerna *före* implementationen av metodkroppen.

### 12.3.4 Design

Nedan följer beskrivning av medlemmar som de olika klasserna bankapplikationen måste innehålla. Dessa påbörjade klasser finns i kursens workspace, tillsammans med de färdigskrivna klasserna HistoryEntry och Date samt BankEvent med tillhörande subtyper: [https://github.com/lunduniversity/introprog/tree/master/workspace/w13\\_bank\\_proj](https://github.com/lunduniversity/introprog/tree/master/workspace/w13_bank_proj)

```
package bank

/**
 * A customer of a bank with provided name and id.
 */
case class Customer(name: String, id: Long):
  override def toString(): String = ???
```

```
package bank

/**
 * A bank account for hold by a specific customer.
 * The account is given a unique account number and initially
 * has a balance of 0 kr.
 */
class BankAccount(val holder: Customer):
  val accountNumber: Int = ???

  /**
   * Deposits the provided amount in this account.
   */
  def deposit(amount: BigInt): Unit = ???

  /**
   * Returns the balance of this account.
   */
  def balance: BigInt = ???

  /**
   * Withdraws the provided amount from this account,
   * if there is enough money in the account. Returns true
   * if the transaction was successful, otherwise false.
   */
  def withdraw(amount: BigInt): Boolean = ???

  override def toString(): String = ???
```

```
package bank

import time.Date

/**
 * A bank with no accounts and no history.
 */
class Bank():
  /**
   * Returns a list of every bank account in the bank.
   * The returned list is sorted in alphabetical order based
   * on customer name.
   */
  def allAccounts(): Vector[BankAccount] = ???
```



```

/**
 * Returns the account holding the provided account number.
 */
def findByNumber(accountNbr: Int): Option[BankAccount] = ???

/**
 * Returns a list of every account belonging to
 * the customer with the provided id.
 */
def findAccountsForHolder(id: Long): Vector[BankAccount] = ???

/**
 * Returns a list of all customers whose names match
 * the provided name pattern.
 */
def findByName(namePattern: String): Vector[Customer] = ???

/**
 * Executes an event in the bank.
 * Returns a string describing whether the
 * event was successful or failed.
 */
def doEvent(event: BankEvent): String = ???

/**
 * Returns a log of all changes to the bank's state.
 */
def history(): Vector[HistoryEntry] = ???

/**
 * Resets the bank to the state it had at the provided date.
 * Returns a string describing whether the event was
 * successful or failed.
 */
def returnToState(returnDate: Date): String = ???

```

### 12.3.5 Tips

- Använd ett **match**-uttryck för att hantera de olika subtyperna av BankEvent när du implementerar doEvent.

```

event match {
  case Deposit(account, amount) => ???
  case Withdraw(account, amount) => ???
  case Transfer(accFrom, accTo, amount) => ???
  case NewAccount(id, name) => ???
  case DeleteAccount(account) => ???
}

```

- För att skriva till fil på ett enkelt sätt kan man t.ex. använda sig av statiska metoder i klassen Files som finns tillgänglig i java.nio.file. För att undvika portabilitetsproblem kan man då använda sig av ett bestämt Charset, t.ex. UTF\_8, som finns tillgänglig i java.nio.charset.StandardCharsets.UTF\_8.

- För att läsa ifrån en fil kan man t.ex. använda sig av Source som finns tillgänglig i `scala.io.Source`.
- Var noggrann med att dina tester innehåller fler fall än de som givits i exempel (se [12.3.9](#)), vilka kan behövas för mer omfattande testning och avlusning och efterfrågas på redovisningen.

### 12.3.6 Obligatoriska uppgifter

**Uppgift 1.** Implementera klassen `Customer`. Testa så att den fungerar REPL.

**Uppgift 2.** Implementera klassen `BankAccount`. Testa så att den fungerar i REPL.

**Uppgift 3.** Skapa ett huvudprogram i singelobjektet `BankApplication`. Gör så att huvudprogrammet skriver ut menyn korrekt och kan ta input från tangentbordet som motsvarar de menyval som ska finnas. Låt val av menyerna ge ett meddelande som berättar för användaren att de ännu ej är implementerade.

**Uppgift 4.** Implementera klassen `Bank`.

- Implementera menyval 6. När användaren väljer att skapa ett nytt konto ska `BankApplication` skapa ett `NewAccount`-objekt som den sedan använder som argument i ett anrop till `doEvent` i `Bank`. Det är i `doEvent` (eller en privat funktion som anropas från `doEvent`) som kontot faktiskt ska skapas.
- Implementera menyval 8. Kontrollera att både menyval 6 och 8 fungerar rätt.
- Implementera menyval 9. Varje gång `doEvent` exekveras utan fel ska dess `BankEvent`-argument läggas till i historiken tillsammans med det nuvarande datumet.
- Implementera alla andra menyval, förutom menyval 10. Testa de nya menyvalen noga efterhand som du implementerar dem, i synnerhet så att ändringshistoriken fungerar korrekt. Gör de utökningar du anser behövs.

**Uppgift 5.** Implementera säkerhetskopiering av historiken.

- När något läggs till i historiken ska det också skrivas till en historikfil omedelbart. Banken ska ej behöva avslutas för att utskriften ska hamna på fil, om så vore fallet kan information gå förlorad om banken kraschar. Använd `toLogString`-metoden i `HistoryEntry` för att få utskrifter i rätt format.
- När programmet startar ska det läsa in alla händelser från historikfilen och återuppspela dem en efter en. På så sätt kan bankens tillstånd återställas, fastän vi bara har sparat ändringshistoriken och inte själva tillståndet. Använd `fromLogString`-metoden i `HistoryEntry` när du läser in strängar från filen.

**Uppgift 6.** Implementera menyval 10 genom att först nollställa bankens tillstånd och sedan återuppspela allt i historiken som hände före det givna datumet. Resten av historiken bör tas bort permanent, både i minnet och i historikfilen.

### 12.3.7 Frivilliga extrauppgifter

Gör först klart projektets obligatoriska delar. Därefter kan du, om du vill, utöka ditt program enligt följande.

**Uppgift 7.** Implementera ett nytt menyalternativ som skriver ut all kontohistorik för en given person. I historiken ska finnas typ av händelse med tillhörande parametrar, dåvarande saldo vid händelsen, såväl som datumet för händelsen. (Du kan ha nytta av denna funktion när du testar ditt program.)

**Uppgift 8.** Skriv en eller flera av klasserna Customer och BankAccount i Java istället och använd dig av dessa i din Scala-kod. (Detta är en nyttig uppgift som förberedelse inför efterkommande fördjupningskurs, som har Java som huvudspråk.)

### 12.3.8 Exempel på historikfil

I workspace-katalogen för denna projektuppgift medföljer en historikfil. Inläsning och utskrift ska ske med dess format. Varje rad representerar en händelse, och formatet för en rad är: **År Månad Dag Timme Minut BankEventTyp Argument**. De olika sorternas BankEvent representeras med följande bokstäver: D för Deposit, W för Withdraw, T för Transfer, N för NewAccount och E för DeleteAccount.

### 12.3.9 Exempel på körning av programmet

Nedan visas möjliga exempel på körning av programmet. Data som matas in av användaren är markerad i fetstil. Ditt program måste inte se identiskt ut, men den övergripande strukturen såväl som resultat av körningen ska vara densamma. När det första exemplet börjar förutsätts det att banken inte har några konton.

Listan över val, som är markerad i kursiv stil i det första exemplet, är inte utskriven i senare exempel för att spara plats på pappret. Ditt program ska alltid skriva ut listan över val före användaren ska mata in ett val.

```
-----
1. Hitta konton för en given kund
2. Sök efter kunder på (del av) namn
3. Sätt in pengar
4. Ta ut pengar
5. Överför pengar mellan konton
6. Skapa nytt konto
7. Radera existerande konto
8. Skriv ut alla konton i banken
9. Skriv ut ändringshistoriken
10. Återställ banken till ett tidigare datum
11. Avsluta
Val: 6
Namn: Adam Asson
Id: 6707071234
Nytt konto skapat med kontonummer:
1000
10:03 14/5-2016
```

```
-----
Val: 1
Id: 6707071234
Konto 1000 (Adam Asson, id 6707071234)
0 kr
10:04 14/5-2016
-----
Val: 6
Namn: Berit Besson
Id: 8505255678
Nytt konto skapat med kontonummer:
1001
10:12 14/5-2016
-----
Val: 8
Konto 1000 (Adam Asson, id 6707071234)
0 kr
Konto 1001 (Berit Besson, id 8505255678)
0 kr
10:13 14/5-2016
```

-----  
Val: **2**

Namn: **adam**

Adam Asson, id 6707071234

10:15 14/5-2016  
-----

Val: **6**

Namn: **Berit Besson**

Id: **8505255678**

Nytt konto skapat med kontonummer:  
1002

13:56 14/5-2016  
-----

Val: **2**

Namn: **erit**

Berit Besson, id 8505255678

14:01 14/5-2016  
-----

Val: **3**

Kontonummer: **1000**

Summa: **5000**

Transaktionen lyckades.

14:36 14/5-2016  
-----

Val: **5**

Kontonummer att överföra ifrån: **1000**

Kontonummer att överföra till: **1001**

Summa: **1000**

Transaktionen lyckades.

14:37 14/5-2016  
-----

Val: **8**

Konto 1000 (Adam Asson, id 6707071234)  
4000 kr

Konto 1001 (Berit Besson, id 8505255678)  
1000 kr

Konto 1002 (Berit Besson, id 8505255678)  
0 kr

14:52 14/5-2016  
-----

Val: **7**

Ange konto att radera: **1002**

Transaktionen lyckades.

14:54 14/5-2016  
-----

-----  
Val: **8**

Konto 1000 (Adam Asson, id 6707071234)  
4000 kr

Konto 1001 (Berit Besson, id 8505255678)  
1000 kr

14:55 14/5-2016  
-----

Val: **9**

10:03 14/5-2016: Skapade ett konto  
tillhörandes Adam Asson, id 6707071234

10:12 14/5-2016: Skapade ett konto  
tillhörandes Berit Besson, id 8505255678

13:56 14/5-2016: Skapade ett konto  
tillhörandes Berit Besson, id 8505255678

14:36 14/5-2016: Satte in 5000 kr i konto  
1000

14:37 14/5-2016: Överförde 1000 kr från  
konto 1000 till konto 1001

14:54 14/5-2016: Raderade konto 1002

14:58 14/5-2016  
-----

Val: **10**

Vilket datum vill du återställa banken  
till?

År: **2016**

Månad: **5**

Datum (dag): **14**

Timme: **10**

Minut: **5**

Banken återställd.

15:00 14/5-2016  
-----

Val: **9**

10:03 14/5-2016: Skapade ett konto  
tillhörandes Adam Asson, id 6707071234

15:00 14/5-2016  
-----

Val: **8**

Konto 1000 (Adam Asson, id 6707071234)  
0 kr

15:01 14/5-2016

-----  
Val: **3**

Kontonummer: **1001**

Summa: **5000**

Transaktionen misslyckades. Inget sådant  
konto hittades.

15:06 14/5-2016

-----  
Val: **4**

Kontonummer: **1000**

Summa: **1000**

Transaktionen misslyckades. Otillräckligt  
saldo.

15:23 14/5-2016

## 12.4 Projektuppgift: music

### Förberedelser

- ☐ Testa så att datorn du ska använda på redovisningen kan spela upp ljud med `javax.sound.midi` genom att köra igång `Main` i den givna koden.
- ☐ Det är bra om du kan ta med hörlurar till datorsalen så att du inte stör andra.
- ☐ Hämta given kod via [kursen github-plats](#) eller via hemsidan under [Download](#).

### 12.4.1 Bakgrund

När man skriver program skapar man ofta modeller av en viss verklig *domän*, som kan vara t.ex. försäkringskassans regelverk eller en fysiksimulering i ett datorspel. För att kunna skapa sådana modeller behöver man ofta skaffa sig *domänkunskap* genom att noga sätta sig in i vad olika koncept i domänen innebär och hur de är relaterade. Med denna kunskap kan du skapa kod som modellerar domänen, utifrån noga valda förenklingar av den komplexa verkligheten. Förmåga att kunna skapa domänmodeller utgör en viktig grund för konsten att utveckla bra programvarusystem, och du kommer lära dig mer om detta i kommande kurser.

I denna laboration ska du skapa ett program baserat på en förenklad modell av domänen *musik*. Du får färdig kod som modellerar hur toner är uppbyggda, samt hur olika sträng-instrument fungerar. Med denna domänmodell ska du skapa ditt eget musikprogram som använder *ackord* som är uppbyggda av flera toner som spelas tillsammans.

### 12.4.2 Domänmodell

#### Tonhöjd

En **ton** (eng. *note*) som spelas på ett instrument, t.ex. ett piano eller en gitarr, har en **tonhöjd** (eng. *pitch*) som är relaterad till den specifika grundfrekvens som tonens ljud har. I vår modell av musikdomänen tillordnar vi olika distinkta tonhöjder ett unikt heltal. En tonhöjd kan då beskrivas av en **case class** `Pitch(nbr: Int)` där vi använder `nbr` i intervallet (0 to 127). Heltalet 60 motsvarar en viss ton, som även har namnet "C5", och som ligger ungefär i mitten av tangentbordet på ett piano.

Inom (västerländsk) musik utgår man från 12 olika *tonklasser* (eng. *pitch classes*). Dessa tolv tonklasser är ordnade i en sekvens av så kallade *halva tonsteg* och har följande **tonklassnamn**:

```
val pitchClassNames: Vector[String] =
  Vector("C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B")
```

Efter tonklassen med namnet B återkommer tonklassen med namnet C. Symbolen # representerar en höjning ett halvt tonsteg. Tonklassen `C#` uttalas *siss* på svenska, och *see sharp* på engelska.<sup>17</sup> Notera att det är ett halvt tonsteg mellan E och F, samt mellan B och C (det finns därför varken `E#` eller `B#` i listan med tonklassnamn.<sup>18</sup>)

På ett piano motsvaras de vita tangenterna av tonklassnamnen C D E F G A B och de svarta tangenterna motsvaras av tonklassnamnen C# D# F# G# A#.

<sup>17</sup>Man använder även b-förtecknet *b*, som uttalas *flat* på engelska, för sänkning av en ton ett halvt tonsteg, men för enkelhetens skull bortser vi i vår modell från detta sätt att namnge toner.

<sup>18</sup>Varför det är på detta viset kan du läsa mer om på t.ex. Wikipedia, men du kan också nöja dig med att det helt enkelt är så på grund av historiska skäl.

En s.k. **tonklass** är ett positivt heltal i intervallet 0 until 12 som motsvaras av index för tonklassnamnet i `pitchClassNames`. En tonhöjd `Pitch(nbr)` tillhör tonklassen `nbr % 12`.

Med hjälp av heltalsdivision med 12 får man fram tonhöjdens så kallade **oktav**, alltså `nbr / 12`. Ett piano har normalt toner som spänner över 7 eller 8 oktav. En tonhöjd `Pitch(nbr)` kan även namnges med en kombination av tonklassnamnet och tonens oktav, t.ex. "C5".

Med denna domänbeskrivning kan vi skapa en mer detaljerad modell av konceptet tonhöjd med hjälp av en case-klass och tillhörande kompanjonsobjekt:

```
package music

case class Pitch(nbr: Int): //Tonhöjd
  assert((0 to 127) contains nbr, s"Error: nbr $nbr outside (0 to 127)")
  def pitchClass: Int      = nbr % 12
  def pitchClassName: String = Pitch.pitchClassNames(pitchClass)
  def name: String         = s"$pitchClassName$octave"
  def octave: Int          = nbr / 12
  def +(offset: Int): Pitch = Pitch(nbr + offset)
  override def toString    = s"Pitch($name)"

object Pitch:
  val defaultOctave = 5 // mittenoktaven på ett pianos tangentbord

  val pitchClassNames: Vector[String] =
    Vector("C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B")

  val pitchClassIndex: Map[String, Int] = pitchClassNames.zipWithIndex.toMap

  def fromString(s: String): Option[Pitch] = scala.util.Try {
    val (pitchClassName, octaveName) = s.partition(c => !c.isDigit)
    val octave = if octaveName.nonEmpty then octaveName.toInt else 5
    Pitch(pitchClassIndex(pitchClassName) + octave * 12)
  }.toOption

  def apply(s: String): Pitch =
    fromString(s).getOrElse(throw new IllegalArgumentException)
```

Kompanjonsobjektet har två fabriksmetoder som kan skapa `Pitch`-objekt från en strängrepresentation av en tonhöjd.

- Metoden `fromString` omvandlar en sträng till en `Option[Pitch]`.
- Metoden `apply` kastar ett undantag om det inte går att omvandla en sträng till ett `Pitch`-objekt.



**Uppgift 1.** Vilka två uttryck i `Try`-blocket kan ge undantagen `NumberFormatException` respektive `NoSuchElementException`? Undersök liknande uttryck i REPL som ger dessa undantag. Hur kan fabriksmetoden `fromString` skrivas om så att den använder `toIntOption` i stället för `toInt` på strängen och `get` i stället för `apply` på nyckelvärde-tabellen och utan att använda `scala.util.Try`? Vad finns det för nackdelar med att gå omvägen via `scala.util.Try` i stället för metoder som direkt ger `Option`?




**Uppgift 2.** Undersök klassen `Pitch` i REPL.

```
1 > sbt
2 sbt> console
3 Welcome to Scala 2.13.3 (OpenJDK 64-Bit Server VM, Java 11.0.8).
```



```

4 Type in expressions for evaluation. Or try :help.
5
6 scala> import music._
7 import music._
8
9 scala> Pitch("C#5").nbr
10 res0: Int = 61
11
12 scala> Pitch("C") + 1
13 res1: music.Pitch = Pitch("C#5")

```

-  a) Ge ett exempel på argument till `Pitch.apply` som gör att undantag kastas.
-  b) Ge ett exempel på argument till `Pitch.fromString` som ger `None`.
-  c) Ge ett exempel på argument till `Pitch.+` som gör att undantag kastas.

**Uppgift 3.** Ändra i implementationen av `fromString` så att du i stället för `.toOption` gör en mönstermatchning med **match** på Try-resultaten `Success` och `Failure` i varsin case-klausul på formen **case** `Success(x) =>` och **case** `Failure(e: ???) => ???` och returnera lämpligt värde. Ta endast hand om de två förväntade undantagstyperna som du identifierade i uppgift 1. Gör så att alla övriga eventuella undantag kastas genom denna klausul: **case** `Failure(e) => throw e`

- a) Testa så att din lösning fungerar i både normalfall och vid felaktigt tonhöjdsnamn.
-  b) Undersök vad som händer om du kommenterar bort olika case-klausuler. När ger kompilatorn varning? Varför?
-  c) Finns det någon fördel resp. nackdel med att bara fånga vissa undantag?

## Ackord

Ett ackord består av flera toner som spelas tillsammans. Man kan spela ett ackord på ett stränginstrument genom att slå an en mängd toner samtidigt eller en sekvens av toner i snabb följd. Man väljer att kalla en av tonerna i ackordet (oftast den lägsta/första tonen) för **grundton** (eng. *root*).

Ett **intervall** är en tons relativa tonhöjdsavstånd från grundtonen. Ackord har olika namn beroende på vilka intervall som ingår i ackordet. Det finns väldigt många olika ackordnamn, men här begränsar vi oss för enkelhetens skull till fyra olika typer av ackord:<sup>19</sup>

- dur-ackord, betecknas t.ex. "C",
- moll-ackord, betecknas t.ex. "Cm"
- sju-ackord, betecknas t.ex. "C7"
- maj-sju-ackord som betecknas t.ex. "Cmaj7".

I case-klassen `Chord` nedan finns en metod `name` som definerar vilka intervall som ingår i de olika ackordtyperna ovan, utom maj-sju-ackord. Den krångliga modulo-12-omräkningen innan matchningen gör så att intervall i olika oktaver behandlas lika, även för negativa intervall.

<sup>19</sup>Om du vill veta mer om ackordnamn läs här: [https://en.wikipedia.org/wiki/Chord\\_\(music\)](https://en.wikipedia.org/wiki/Chord_(music))



```

package music

case class Chord(ps: Vector[Pitch]):
  assert(ps.nonEmpty, "Chord pitch sequence is empty")

  val pitchClasses: Vector[Int] = ps.map(_.pitchClass).toVector

  def apply(i: Int): Pitch = ps(i)

  def intervals(root: Pitch = ps(0)): Vector[Int] = ps.map(_ - root.nbr)

  def relativePitchClasses(root: Pitch = ps(0)): Vector[Int] =
    intervals(root).map(i => (i%12 + 12) % 12).distinct.sorted

  def name(root: Pitch = ps(0)): String = relativePitchClasses(root) match
    case Vector(0, 4, 7)      => root.pitchClassName
    case Vector(0, 3, 7)      => root.pitchClassName + "m"
    case Vector(0, 4, 7, 10) => root.pitchClassName + "7"
    case _                    => root.pitchClassName + intervals(root).mkString("[", ",", ",")

  override def toString = ps.map(_.name).mkString("Chord(", ",", ",")

object Chord:
  def apply(xs: String*): Chord = Chord(xs.map(Pitch.apply).toVector)

  def random(pitchNumbers: Seq[Int] = (60 to 72), n: Int = 3): Chord =
    val shuffled = scala.util.Random.shuffle(pitchNumbers).toVector
    Chord(shuffled.take(n).map(Pitch.apply))

```

### Uppgift 4.

- Maj-sju-ackord har samma intervall som sju-ackord, förutom att det fjärde intervallet ska vara 11 halva tonsteg från grundtonen i stället för 10. Lägg till en case-klausul i `Chord.name` så att maj-sju-ackord ges namn som slutar med ändelsen "maj7".
- Testa din kod och kontrollera så att ackordet `Chord("D4", "F#4", "A4", "C#5")` får namnet "Dmaj7"

```

1 scala> Chord("D4", "F#4", "A4", "C#5").name()
2 res2: String = "Dmaj7"

```

- Vilka fyra toner har ett Cmaj7-ackord med grundtonen "C5"?

### Stränginstrument

Ett stränginstrument, t.ex. ett piano eller en gitarr, kännetecknas av att det kan spela ackord genom att flera strängar kan sättas i svängning så att många toner spelas tillsammans. I vår modell fångar vi denna egenskap med en trait `StringInstrument` som har en metod `toChordOpt` som ger något ackord om minst en sträng spelas.

Gitarr och ukulele är exempel på stränginstrument som har en greppbräda (eng. *fret board*). Man spelar på ett stränginstrument med greppbräda (eng. *fretted instrument*) genom att trycka strängar mot greppbrädan med en hand, samtidigt som man knäpper på strängarna med den andra handen. Olika instanser av dessa instrument kan skilja sig åt vad gäller antalet strängar och hur dessa strängar är stämda. En normal gitarr har 6 strängar, medan en normal ukulele bara har 4 strängar. Dessa egenskaper modelleras i koden nedan.

Varje sträng har en stämskruv med vilken kan man ändra strängens spänning, strängens s.k. **stämning** (eng. *tuning*). Om man knäpper på alla lösa strängarna på en gitarr med standardstämning spelas tonerna E3, A3, D4, G4, B4, E5, räknat från den tjockaste till den tunnaste strängen.

```
package music

trait StringInstrument { def toChordOpt: Option[Chord] }

case class Piano(isKeyDown: Set[Int]) extends StringInstrument:
  def toChordOpt: Option[Chord] =
    if isKeyDown.nonEmpty then
      Some(Chord(isKeyDown.toVector.sorted.map(Pitch.apply)))
    else None

trait FrettedInstrument extends StringInstrument:
  def nbrOfStrings: Int
  def tuning: Vector[Pitch]
  def grip: Vector[Int]
  def toChordOpt: Option[Chord] =
    val notes =
      for i <- grip.indices if grip(i) >= 0
      yield tuning(i) + grip(i)
    if notes.nonEmpty then Some(Chord(notes.toVector)) else None

case class Guitar(pos: (Int,Int,Int,Int,Int,Int)) extends FrettedInstrument:
  val grip = Vector(pos._1, pos._2, pos._3, pos._4, pos._5, pos._6)
  val nbrOfStrings = 6
  val tuning =
    "E3 A3 D4 G4 B4 E5".split(' ').map(Pitch.apply).toVector

case class Ukulele(pos: (Int,Int,Int,Int)) extends FrettedInstrument:
  val grip = Vector(pos._1, pos._2, pos._3, pos._4)
  val nbrOfStrings = 4
  val tuning =
    "A5 D5 F#5 B5".split(' ').map(Pitch.apply).toVector
```

Om man trycker på greppbrädans olika positioner får man olika toner, beroende på vilken position man trycker på. Positionerna på greppbrädan räknas från ett och uppåt. Position 0 motsvarar **lös sträng**, alltså att man slår an en sträng utan att trycka på greppbrädan över denna sträng. En negativ position, tex. -1, anger att en sträng inte spelas alls; många gitarrackord spelas genom att bara en delmängd av strängarna slås an. Ett exempel på ett gitarrackord visas i figur 12.2.



**Uppgift 5.** Studera modellen av stränginstrument ovan och använd REPL för att svara på dessa frågor:

- Vad är namnet på detta pianoackord om vi väljer att lägsta tonen i ackordet är grundton: `Piano(Set(60, 64, 67, 70))`
- Vad heter tonerna som ingår i ackordet `Guitar(3,3,2,0,1,0)`.
- Vad heter detta ackord om vi väljer ett A som grundton: `Ukulele(0,2,1,2)`

### Elektroniska instrument

Ett elektroniskt instrument syntetiserar ljud med hjälp av analog och/eller digital elektronik, och kallas därför **synthesizer**, ofta förkortat *synt* (eng. *synth*).



**Figur 12.2:** Ett C-dur-ackord på en gitarr motsvarande Guitar(3,3,2,0,1,0).

De flesta moderna PC-operativsystem inkluderar mjukvaruimplementerade syntar som följer den så kallade MIDI-standarden. Java-paketet `javax.sound.midi` innehåller klasser som kan få en sådan MIDI-synt att spela musik.

MIDI-standarden baseras på en modell av ett pianotangentbord där olika toner kan vara "på" eller "av" beroende på om en tangent är nedtryckt eller ej. Dessa toners höjd är modellerade på samma sätt som i vår klass `Pitch`, där alltså tonhöjden 60 motsvarar tonen "C5", etc. En tangent kan tryckas ner olika hårt, vilket representeras av ett heltalsvärde i `Range(0,128)` kallat `velocity`. Ett högt värde ger en stark ton, medan ett litet värde motsvarar en svag (tyst) ton.

En synt som följer MIDI-standarden kan spela upp ljud via 16 olika så kallade **kanaler** (eng. *channel*), numrerade (0 until 16), där varje kanal kan ställas in så att den spelar ett ljud som t.ex. liknar ett visst verkligt instrument, så som piano eller gitarr.

I kursens workspace i paketet `music` finns en `Synth`-modul som förenklar användningen av Java-paketet `javax.sound.midi`. I modulen `Synth` finns metoden `playBlocking` som kan spela flera toner under en viss tid med hjälp av synten på ditt ljudkort. Exekveringen av ditt program blockeras tills tonerna spelats klart, därav "*blocking*" i namnet.

Metoden `playBlocking` har följande parametrar, default-argument och returtyp: <sup>20</sup>

```
def playBlocking(
  noteNumbers: Seq[Int] = Seq(60), // en sekvens av tonhöjder
  velocity: Int          = 60,      // hur hårt anslag i Range(0, 128)
  duration: Long         = 300,     // hur länge i millisekunder
  spread: Long           = 50,      // millisekunder mellan tonerna
  after: Long            = 0,       // millisekunder innan första tonen
  channel: Int           = 0        // MIDI-kanal som spelar tonerna
): Unit
```

<sup>20</sup>Om du är nyfiken kan du studera implementationen av `Synth`-modulen här:

[https://github.com/lunduniversity/introprog/tree/master/workspace/w13\\_music\\_proj](https://github.com/lunduniversity/introprog/tree/master/workspace/w13_music_proj) Koden blir lättare att förstå om du samtidigt läser api-dokumentationen av paketet `javax.sound.midi` och även lära dig mer om MIDI-standarden med hjälp av t.ex. wikipedia.

**Uppgift 6.** Anropa `playBlocking()` i REPL och undersök om din dator kan spela tonen "C5". Använd gärna lurar så att du inte stör dina labbkamrater. Prova vad som händer när du ger olika argument till `playBlocking`.

**Uppgift 7.** Gör klart modulen `ChordPlayer` enligt nedan så att metoden `play` kan spela ett ackord. Case-klassen `Strike` representerar ett ackordanslag.

```
package music

object ChordPlayer:

  case class Strike(
    velocity: Int      = 50,    // hur hårt anslag i Range(0, 128)
    duration: Long     = 500,   // hur länge i millisekunder
    spread: Long       = 50,    // millisekunder mellan tonerna
    after: Long        = 0      // millisekunder innan första tonen
  )

  def play(chord: Chord, strike: Strike = Strike(), channel: Int = 0): Unit =
    strike match
      case Strike(v, d, s, a) => ???
```

**Uppgift 8.** Implementera ett singelobjekt med namnet `Test` med en `main`-metod som med hjälp av din `play`-metod från föregående uppgift spelar några olika ackord.

**Uppgift 9.** Gör en terminalapp som kan spela ackord. I kursens workspace i `w13_music_proj` finns en påbörjad terminalapp som du kan bygga vidare på. Den har redan en `Main.main`-metod som startar en loop där användaren kan ge kommando (eng. *Command Line Interface, CLI*). Kommandot `? ger hjälp` och kommandot `:q avslutar`.

```
1  *** Welcome to music!
2  music> ?
3  ?          print help
4  :q         quit this app
5  !         play chord TODO
6  music> !
7  play chord TODO
8  music> :q
9  Goodbye music!
```

Det finns, som syns ovan, också ett påbörjat kommando `!` som är tänkt att spela ett ackord, men som än så länge bara skriver ut ett `TODO`-meddelande. Gör så att användaren med `!` kan spela ackord från olika instrument enligt nedan:

```
1  music> ! p 60 64 67
2  Play Piano(Set(60, 64, 67)) Chord(C5,E5,G5)
3  music> ! g 0 2 2 0 0 0
4  Play Guitar((0,2,2,0,0,0)) Chord(E3,B3,E4,G4,B4,E5)
```

**Uppgift 10.** Skapa ett kommando som låter användare definierar egna namn på kommandon som sedan enkelt kan köras med hjälp av det definierade namnet. Vid definition med tidigare existerande namn så ska den gamla definitionen ersättas

```
1  music> def Em = ! g 0 2 2 0 0 0
2  defined Em = ! g 0 2 2 0 0 0
3  music> Em
4  Play Guitar((0,2,2,0,0,0)) Chord(E3,B3,E4,G4,B4,E5)
```

Det ska fungera att göra nästlade def-kommando, alltså att kroppen för en def innehåller namnet på en annan def. Testa att definiera en hel låt som i sin tur består av definierade ackord.

**Uppgift 11.** Du ska inför redovisningen generera automatisk dokumentation baserat på dokumentationskommentarer enligt instruktioner i Appendix E. Du ska skriva relevanta dokumentationskommentarer för minst hälften av dina publika metoder. Det är ofta användbart att skriva dokumentationskommentarerna *före* implementationen av metodkroppen.

### 12.4.3 Valfria uppgifter

**Uppgift 12.** Gör så att definitioner sparas mellan körningar.

**Uppgift 13.** Implementera fler valfria kommandon. Du kan t.ex.

- skapa kommando som ritar en grepptabell för gitarrackord eller fingersättning på piano med `introprog.PixelWindow`.
- skapa kommando för att göra drumbeats, t.ex. `drum mygrove x x o x` för 2 hihat + basatrumma + 1 hihat och start mygrove och stop mygrove för uppspelning av beat i bakgrunden med `playConcurrently`. Se i `Synth.scala` för vilka instrument som ger trumljud med ledning av deras namn `music.Synth.instruments.map(_.getName)`, t.ex. `Standard Kit` eller `Synth Drum`. Använd kanal 10 för att spela upp trumljuden.

**Uppgift 14.** Använd öppen-källkodsprojektet `jline` i stället för `scala.io.StdIn.readLine` för att automatiskt få pil-upp-historik, Ctrl+A Ctrl+K, TAB-completion, etc. Se exempel på användning av `jline` här: <https://github.com/bjornregnell/termut>

## 12.5 Projektuppgift: photo

### 12.5.1 Bakgrund

Detta projekt innebär att du ska implementera en egen bilbehandlingsapplikation, en mycket förenklad variant av *Phototoshop* eller *Gimp*.

En digital bild består av ett rutnät, en s.k. matris (eng. *matrix*), av pixlar, var och en med en viss färg. Om man har många små pixlar bredvid varandra i ett rutnät, så flyter de samman för ögat och betraktaren upplever en bild.

Bilder kan manipuleras genom applicering av olika s.k. *filter*, som förändrar bildens pixlar på ett intressanta sätt. Du ska, utifrån given matematisk teori, implementera olika filter med hjälp av speciella matrisoperationer.

Det finns olika system för hur man färgsätter pixlar. T.ex. så används CMYK-systemet (cyan, magenta, gul, svart) vid blandning av färg som ska tryckas på papper eller annat material. På en dator, däremot, används vanligtvis RGB-systemet, som har de tre grundfärgerna röd, grön och blå. Mättnaden av varje grundfärg anges av ett heltal som vi i fortsättningen förutsätter ligger i intervallet  $[0, 255]$ . 0 anger "ingen färg" och 255 anger "maximal färg". Man kan därmed representera  $256 \times 256 \times 256 = 16\,777\,216$  olika färgnyanser. Man kan också representera gråskalor; det gör man med färger som har samma värde på alla tre grundfärgerna: (0, 0, 0) är helt svart, (255, 255, 255) är helt vitt.

I detta projekt kommer vi skapa matriser av heltal för att beräkna intressanta egenskaper hos en bild, till exempel intensiteten för varje pixel. För att spara plats vid bearbetning av stora bilder så använder vi, heltalsmatriser med typen `Short`, som använder 16 bitar i minnet, i stället för `Int`, som använder 32 bitar i minnet.

### 12.5.2 Förberedelser

I detta projekt har du nytta av följande delar av `introprog-scalalib` och `java.awt`:

- `introprog.Image` för bildhantering.
- `introprog.PixelWindow` och `introprog.Dialog` för användarinteraktion.
- `introprog.IO` för filhantering.
- `java.awt.Color` för hantering av pixelfärger.

Läs noga dokumentationen för klasserna i `introprog` här och gör egna experiment i REPL så du förstår hur de kan användas: <https://cs.lth.se/pgk/api/>

Läs om klassen `java.awt.Color` här:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/java/awt/Color.html> Hämta och studera noga den kod som är given för detta projekt här:

<https://github.com/lunduniversity/introprog/tree/master/workspace/>

### 12.5.3 Matris med värden av typen `Short`

**Uppgift 1. Matrix.** I den givna kodfilen `Matrix.scala` finns hjälp-funktioner för att skapa och uppdatera matriser med värden av typen `Short`, för att spara minne vid stora bilder.

Gör klart saknade implementationer och testa noga i REPL så att allt fungerar som det ska innan du går vidare. *Tips:* Du har nytta av `Array.tabulate`.



```
scala> import photo.*

scala> val m = Matrix(3,3)(1,2,3,4,5,6,7,8,9) // en 3x3-matris med Short-värden
val m: photo.Matrix = Array(Array(1, 4, 7), Array(2, 5, 8), Array(3, 6, 9))

scala> m(0,1)
val res0: Short = 4

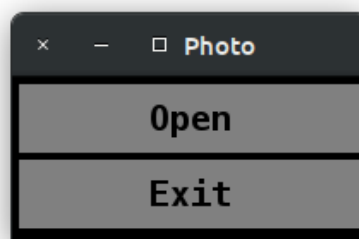
scala> m(1,0) = 42

scala> m
val res1: photo.Matrix = Array(Array(1, 4, 7), Array(42, 5, 8), Array(3, 6, 9))

scala> m.row(0)
val res2: Array[Short] = Array(1, 42, 3)
```

### 12.5.4 Användargränssnitt

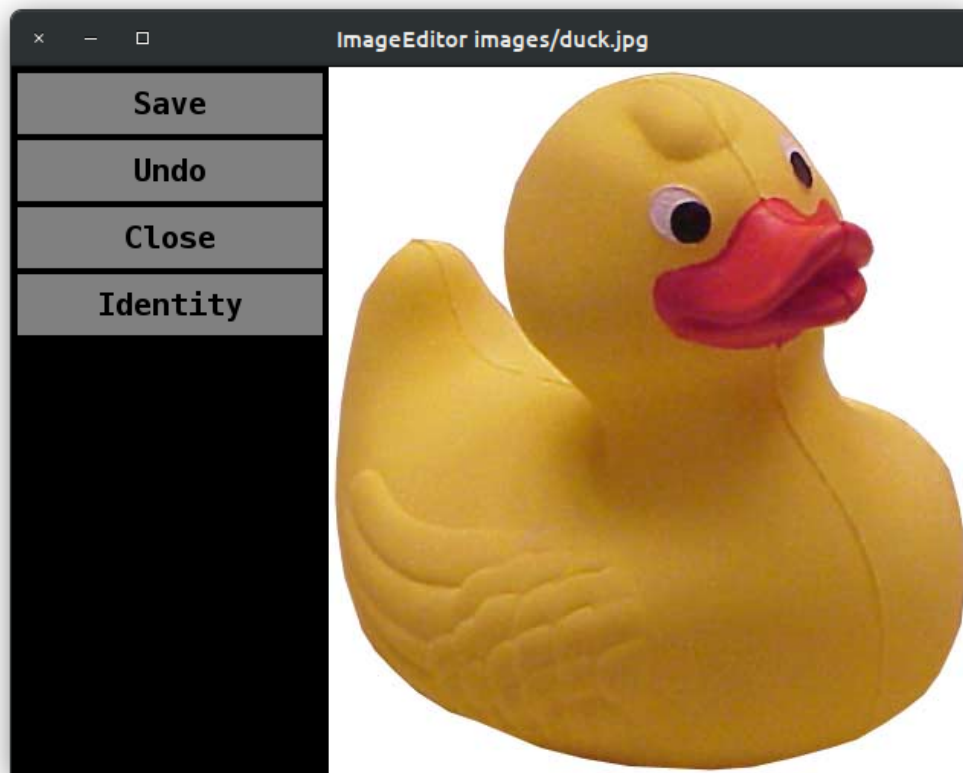
När appen startar så visas ett fönster enligt fig. 12.3, implementerat av givna koden i `Main.scala`. Med hjälp av givna `Button.scala` skapas en kolumn med knappar som är klickbara. Studera koden i `Main.scala` och `Button.scala` noga så att du förstår vad som händer. Ännu öppnas inget `ImageEditor`-fönster, men det ingår i näst uppgift.



**Figur 12.3:** Photo-applikationens startfönster.

#### Uppgift 2. `ImageEditor`. Följande krav ska implementeras:

- Du ska skapa en kodfil `ImageEditor.scala` som innehåller en klass med samma namn som implementerar ett bildredigeringsfönster med en kolumn med knappar till vänster och en bild inläst från fil till höger, så som visas i fig. 12.4.
- Vid tryck på `Exit`-knappen ska en varningsfråga "Ok to Exit without save?" ges med `introprog.Dialog.isOK` och användaren ska kunna ångra avslut. Om avslut ändå väljs så ska detta ske med `System.exit(0)` så att alla ev. aktiva fönster och tillhörande trådar avbryts direkt.
- Vid tryck på `Open`-knappen ska en fil väljas med hjälp av `introprog.Dialog.file`. Om det i aktuell katalog finns en underkatalog vid namn `images` så ska filbläddringen börja där, annars i aktuell katalog.
- Efter OK på filöppningen ska en bild öppnas i ett bildredigeringsfönster enligt fig. 12.4 med knapparna `Save`, `Undo`, `Close`, plus en knapp för varje filter, till vänster om bilden. Fönstrets höjd och bredd ska avpassas så att hela bilden och alla knappar får plats.



**Figur 12.4:** Bildredigeringsfönstret, innan fler filter (utöver identitetsfiltret) implementerats. Varje filter som implementeras ska ha en motsvarande knapp.

- Huvudfönstret och alla bildredigeringsfönster ska fungera parallellt. Detta ska du åstadkomma genom att händelseloopen i ImageEditor-klassen körs som argument till metoden `runInParallell` enligt nedan:

```
def runInParallell(block: => Unit) =
  new Thread{ override def run(): Unit = block }.start

def startEventLoop(): Unit = runInParallell:
  // initialisering och händelseloop här
```

- Det ska gå att göra Undo i flera steg och återställa alla bilder före applicering av filter i tur och ordning. *Tips:* Inför i attributet `var history: Vector[Image]` som från början innehåller den ursprungliga bilden.
- Fönstrets titel ska innehålla namnet ImageEditor och de två sista delarna av den sökväg (eng. *path*) som öppnats, enligt exempel i fig. 12.4, eftersom en fullständig sökväg t.ex. `/home/userxyz/workspace/photo/images/duck.jpg` riskerar att inte få plats i fönstrets titelbalk.
- Vid Save ska fråga om filnamn ställas med `introprog.Dialog.file` och kontroller göras om filen redan finns eller ej, och om den finns ska en fråga med `introprog.Dialog.isOK` ställas om den ska skrivas över eller ej.
- Alla implementerade filter ska ha en knapp som applicerar filtret och sparar resultatet i historiken så att filtret kan ångras med Undo. Om filtret har argument så ska en informativ dialog öppnas där användaren kan ange argument via `introprog.Dialog.input`. Filterknapparna ska vara sorterade i bokstavsordning efter filtrets namn, se fig. 12.5.



### 12.5.5 Filter

Du ska bygga vidare på givna koden i `Filter.scala` som visas nedan. Du ska implementera och testa ett antal olika filter som ändrar bilder på intressanta sätt med hjälp av olika matrisalgoritmer.

```
package photo

import introprog.Image

trait Filter:
  def name: String

  def argDescriptions: Seq[String] = Seq()

  def nbrOfArgs = argDescriptions.length

  def apply(im: Image, args: Double*): Image

object Filter:
  val byIndex: Vector[Filter] = Vector(Identity)

  val byName: Map[String, Filter] = byIndex.map(f => f.name -> f).toMap

  def intensity(im: Image): Matrix = ???

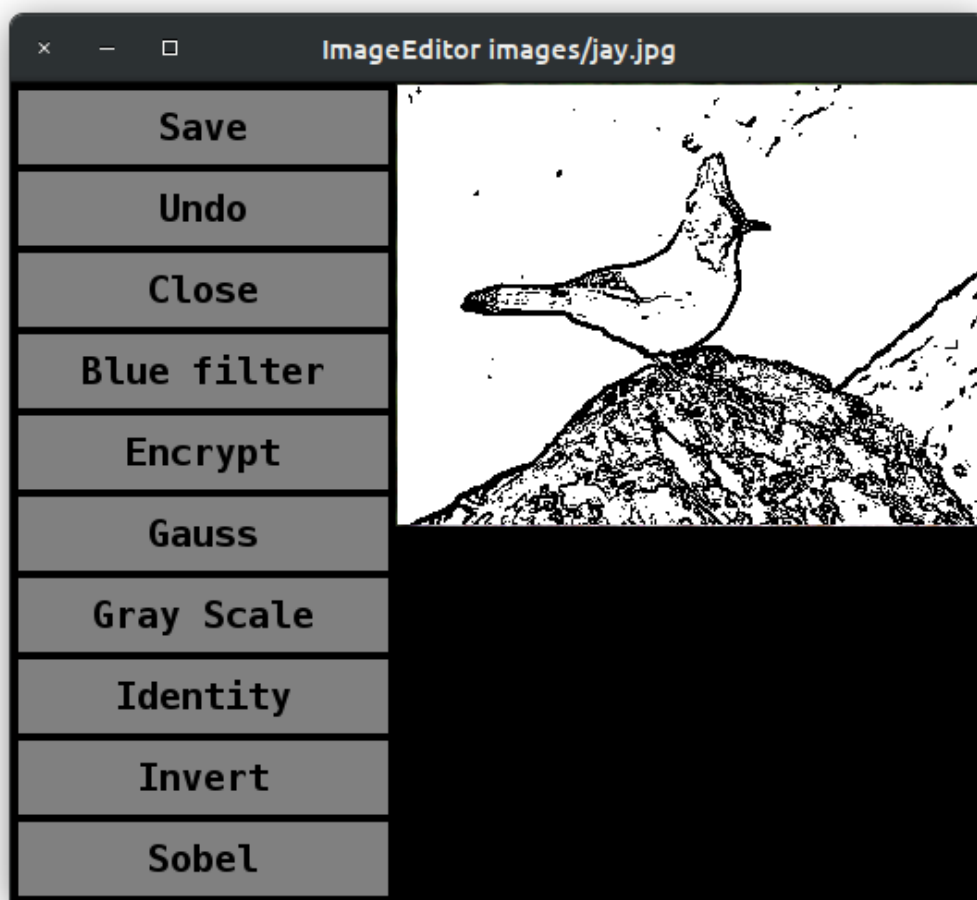
  def convolve(p: Matrix, x: Int, y: Int, kernel: Matrix, weight: Int): Short = ???

  object Identity extends Filter:
    val name = "Identity"
    def apply(im: Image, args: Double*): Image =
      val result = Image.ofDim(im.width, im.height)
      result.updated((x, y) => im(x, y))
```

`Filter.scala` innehåller en bastyp för alla filter med ett antal medlemmar som alla filter ska implementera, enligt nedan krav. Det finns ett färdigimplementerat filter, `Identity`, som kan användas för testsyften; detta filter gör inget annat än kopierar alla bildpunkter till en ny bild och har således ingen editrande effekt.

- Metoden `apply` ska returnera en ny bilden där filtret applicerats, utan att förändra inparameter-bilden.
- Ett filter ska kunna ha noll eller flera argument av typen `Double` som kan påverka vad som händer när filtret appliceras. Varje sådant argument ska i tur och ordning ha en kort, instruktiv beskrivning i sekvensen `argDescription`.
- Metoden `intensity` ska beräkna en s.k. *intensitetsmatris* och behövs vid implementeringen av gråskale-, Gauss- och Sobel-filtren. Hur en intensitetsmatris beräknas beskrivs nedan.
- Metoden `convolve` ska göra en s.k. faltning (medelvärdesbildning i matriser) och behövs vid implementering av Gauss- och Sobel-filtren. Hur en faltning görs beskrivs nedan.
- Alla implementerade filter ska finnas i sekvensen `byIndex`, som används i tabellen `byName`. Dessa behövs för att skapa alla filterknappar och applicera respektive filter.

Du ska implementera och testa alla filter i uppgifterna nedan, ett i taget. Uppgifterna är ordnade i stigande svårighetsgrad.



**Figur 12.5:** Bildredigeringsfönstret med alla filter implementerade. Ett kontrastförstärkande så kallat Sobel-filter är applicerat med tröskelvärde 150.

**Uppgift 3. Blåfilter.** Skapa ett filter i ett singelobjekt med namn Blue som vid applicering ger en blå version av bilden, där varje pixel bara innehåller den blå komponenten. *Tips:* Du har nytta av metoden `getBlue` i klassen `java.awt.Color`.

**Uppgift 4. Negativ.** Skapa ett filter `Invert` som inverterar en bild, dvs skapar en "negativ" kopia av bilden. Ljusa färger ska alltså bli mörka och mörka färger ska bli ljusa. Fundera över vad som kan menas med en inverterad eller negativ kopia. *Tips:* Även de nya RGB-värdena ska vara i heltal i intervallet 0 – 255. De nya RGB-värdena beräknas *inte* med något divisionsuttryck över de gamla värdena (då skulle de nya värdena bli decimaltal och inte heltal i intervallet 0 – 255).

**Uppgift 5. Gråskalefilter.** Skapa ett filter `GrayScale` som gör om bilden till en gråskalebild. Implementera först `intensity`-metoden i `trait Filter` genom att bilda medelvärde av alla tre RGB-komponenterna. Använd sedan intensiteten för varje pixel för att bestämma gråskalenivån. Om intensiteten i en pixel till exempel är 105 så ska den nya gråskale-pixeln var ett `Color`-objekt med RGB-värdena (105, 105, 105).

**Uppgift 6. Kryptering.** Skapa ett filter `XorCrypto` som krypterar bilden med xor-operatörn `^`. Denna operator gör binär xor mellan bitarna i ett heltal. Exempelvis ger  $8 \wedge 127$  värdet 119. Om man gör xor igen med 127, alltså  $119 \wedge 127$ , får man tillbaka värdet 8. Varje pixel krypteras genom att använda xor-operatörn med ursprungsvärdena för rött, grönt och blått tillsammans med slumpmässiga heltalsvärden som genereras ur en ny instans av `scala.util.Random`. Tre nya slumpstal ska dras för varje pixels RGB-komponent ur samma `Random`-instans. Låt användaren ge ett argument som du använder som slumpstalsfrö vid skapande av `Random`-instansen. På så sätt kan du återskapa bilden genom att applicera krypteringsfiltret igen, med samma argument, på den numera krypterade bilden.

Om filtrets `argDescriptions`-sekvens är icke-tom så ska `ImageEditor` fråga efter varje argument i tur och ordning och visa varje beskrivning i dialogrutan. Användarens indata görs om till ett decimaltal av typen `Double` före att argumenten används i metoden `apply`. Bestäm själv hur du vill hantera defaultvärden och felhantering om användaren anger en sträng som inte går att göra om till en `Double`. *Tips:* Du har nytta av `toDoubleOption` och `getOrElse`.

```
object XorCrypto extends Filter:
  val name = "Encrypt"
  override val argDescriptions = Seq("Encryption key")

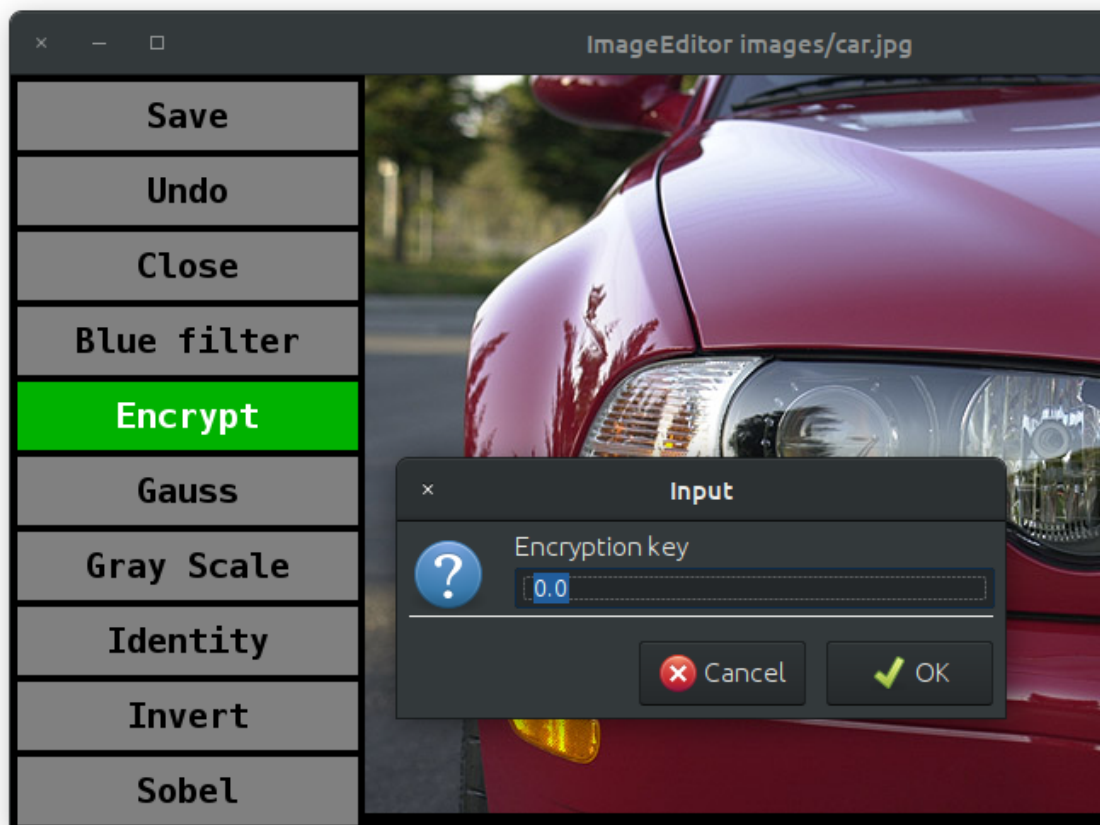
  def apply(im: Image, args: Double*): Image = ???
```

```
scala> Seq(8, 127, 8 ^ 127).map(_.toBinaryString)
val res0: Seq[String] = List(1000, 1111111, 1110111)

scala> 8 ^ 127
val res1: Int = 119

scala> 119 ^ 127
val res2: Int = 8
```

**Uppgift 7. Gaussfilter.** Ett Gaussfilter gör bilden lite mindre skarp. Gaussfiltrering är ett exempel på så kallad *faltningsfiltrering*. Faltning (eng. *convolution*) är en slags lokal medelvärdesbildning. Nya pixlar skapas genom att kombinera varje pixel med dess omgivande pixlar enligt en speciell matrisalgoritm.



**Figur 12.6:** Krypteringsfilter före applicering, under pågående inmatning av nyckel.

För att åstadkomma detta utnyttjar man en så kallad *faltningskärna*  $K$  som är en liten kvadratisk heltalsmatris. Man placerar  $K$  över varje element i intensitetsmatrisen och multiplicerar varje element i  $K$  med motsvarande element i intensitetsmatrisen. Man summerar produkterna och dividerar summan med summan av elementen i  $K$  för att få det nya värdet på intensiteten i punkten (alltså ett slags medelvärde). Divisionen görs för att den nya intensiteten ska hamna i rätt intervall (0 – 255). Exempel:

$$intensity = \begin{pmatrix} 5 & 4 & 2 & 8 & \dots \\ 4 & 3 & 4 & 9 & \dots \\ 9 & 8 & 7 & 7 & \dots \\ 8 & 6 & 6 & 5 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad K = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Här är summan av elementen i  $K$   $1 + 1 + 4 + 1 + 1 = 8$ . För att räkna ut det nya värdet på intensiteten i punkten (1, 1) med det nuvarande värdet är 3, beräknar man följande:

$$newintensity = \frac{0 \cdot 5 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 4 + 4 \cdot 3 + 1 \cdot 4 + 0 \cdot 9 + 1 \cdot 8 + 0 \cdot 7}{8} = \frac{32}{8} = 4$$

Man fortsätter med att flytta  $K$  ett steg åt höger och beräknar på motsvarande sätt ett nytt värde för elementet med index (1) (2) (där det nuvarande värdet är 4 och det nya värdet blir 5). Därefter gör man på samma sätt för alla element utom för "ramen" dvs elementen i matrisens ytterkanter.

Implementera och testa noga först metoden

`convolve(p: Matrix, x: Int, y: Int, kernel: Matrix, weight: Int): Short`  
i **trait** Filter som alltså ska ge den normerade produktsumman av kernel och punkterna i närheten av (x,y) i matrisen p normerat med weight. Tips: Du har nytta av metoderna `round` och `toShort`.

```
scala> import photo.*

scala> val p = Matrix(4,4)(5,4,2,8,4,3,4,9,9,8,7,7,8,6,6,5)
val p: photo.Matrix = Array(Array(5, 4, 9, 8), Array(4, 3, 8, 6), Array(2, 4, 7, 6), Array(8, 9, 7, 6))

scala> val K = Matrix(3,3)(0,1,0,1,4,1,0,1,0)
val K: photo.Matrix = Array(Array(0, 1, 0), Array(1, 4, 1), Array(0, 1, 0))

scala> Filter.convolve(p, 1, 1, K, K.flatten.sum)
val res0: Short = 4
```

Skapa därefter ett filter Gauss som gör en faltning med hjälp av `convolve` för varje färgkomponent separat. Gör på följande sätt:

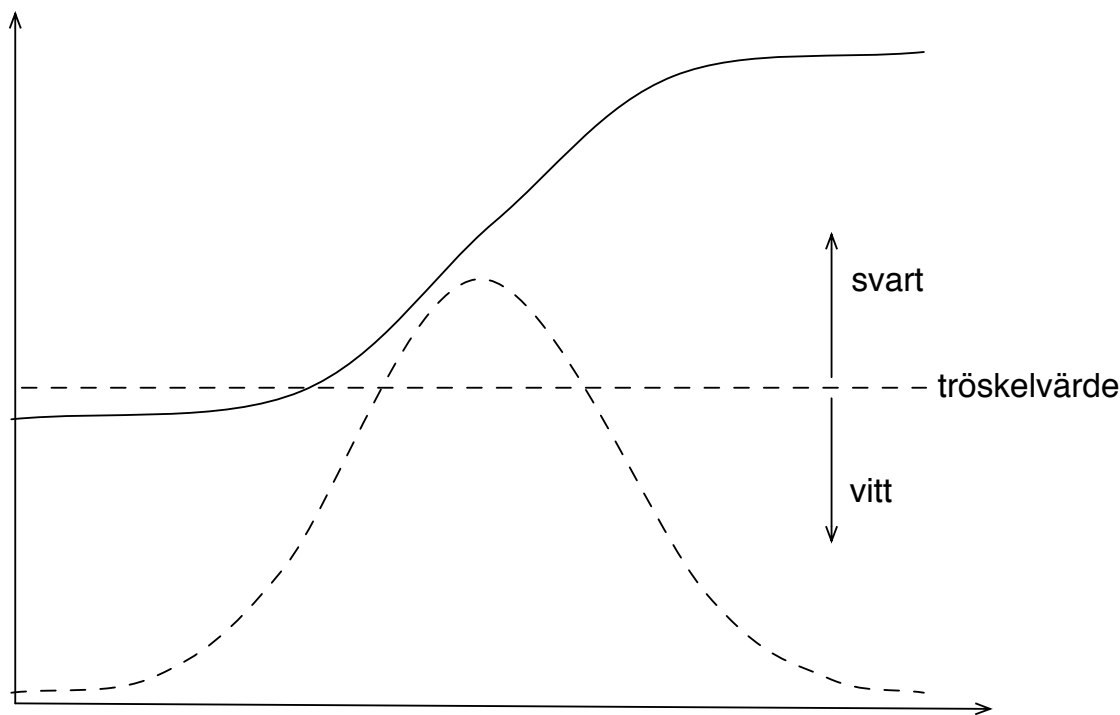
1. Bilda tre short-matriser och lagra pixlarnas red-, green- och blue-komponenter i matriserna.
2. Utför faltningen av de tre komponenterna för varje element och uppdatera result med de uträknade värdena.
3. Elementen i ramen behandlas inte, men i result måste också dessa element få värden. Enklast är att flytta över dessa element oförändrade från im till result. (Man kan också sätta dem till `Color.BLACK`, men då kommer den filtrerade bilden att se något mindre ut.)

Använd kernel  $K$  enligt ovan och låt `weight` vara summan av alla element i  $K$ .

Det kan vara intressant att prova med andra värden än 4 i mitten av faltningsmatrisen. Med värdet 0 får man en större utjämning eftersom man då inte alls tar hänsyn till den aktuella pixelns värde. Låt användaren mata in argument för mittvärdet, mellan 0 och 50, och beskriv detta i `argDescriptions`.<sup>21</sup>

**Uppgift 8. Sobelfilter.** Sobelfiltrering är, precis som Gaussfiltrering, en typ av faltningsfiltrering. Med Sobelfiltrering får man dock motsatt effekt i jämförelse med Gaussfiltrering, dvs man förstärker konturer i en bild. I princip deriverar man bilden i x- och y-led och sammanställer resultatet.

<sup>21</sup>Det kan ibland vara svårt att se någon skillnad mellan den Gauss-filtrerade bilden och originalbilden. Om man vill ha en riktigt suddig bild så måste man använda en större matris som faltningskärna. Prova gärna detta som extrauppgift.



**Figur 12.7:** En funktion (heldragen linje) och dess derivata (streckad linje).

I figur 12.7 visas en funktion  $f$  (heldragen linje) och funktionens derivata  $f'$  (streckad linje). Vi ser att där funktionen gör ett ”hopp” så får derivatan ett stort värde. Om funktionen representerar intensiteten hos pixlarna längs en linje i x-led eller y-led så motsvarar ”hoppen” en kontur i bilden. Om man sedan bestämmer sig för att pixlar där derivatans värde överstiger ett visst tröskelvärde ska vara svarta och andra pixlar vita så får man en bild med starka konturer.

Nu är ju intensiteten hos pixlarna inte en kontinuerlig funktion som man kan derivata enligt vanliga matematiska regler. Men man kan approximera derivatan, till exempel med följande formel:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Om man låter  $h$  gå mot noll så får man definitionen av derivatan. Efter ytterligare teoretiska utredningar så kan man visa att det går att uttrycka derivering i en matris med hjälp av faltning enligt följande:

1. Beräkna intensitetsmatrisen med metoden intensity.
2. För varje punkt i intensitetsmatrisen gör två faltningar med dessa kärnor:

$$SobelX = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad SobelY = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Använd metoden convolve med vikten 1. Koefficienterna i matrisen  $SobelX$  uttrycker derivering i x-led, medan  $SobelY$  uttrycker derivering i y-led. För att förklara varför

koefficienterna ibland är 1, ibland 2, ibland positiva och ibland negativa, måste man studera den bakomliggande teorin noggrant, men det gör vi inte här.

3. Om resultaten av faltningen i en punkt betecknas med  $s_x$  och  $s_y$  så får man en indikator på närvaron av en kontur med `math.abs(s_x) + math.abs(s_y)`. Absolutbelopp behöver man eftersom man har negativa koefficienter i faltningsmatriserna.
4. Sätt pixeln till svart om indikatorn är större än tröskelvärdet, till vit annars. Låt tröskelvärdet bestämmas av ett argument som användaren kan ange.

Skapa ett filter Sobel som implementerar konturförstärkning med ovan algoritm. Se exempel i fig. 12.5. Du ska låta användaren ge tröskelvärdet med argumentbeskrivningen "Threshold (0.0 - 255.0)".

**Uppgift 9.** Du ska inför redovisningen generera automatisk dokumentation baserat på dokumentationskommentarer enligt instruktioner i Appendix E. Du ska skriva relevanta dokumentationskommentarer för minst hälften av dina publika metoder. Det är ofta användbart att skriva dokumentationskommentarerna *före* implementationen av metodkroppen.

### 12.5.6 Frivilliga extrauppgifter

**Uppgift 10. Kortkommando.** Gör så att det blir möjligt att applicera filter med hjälp av tangenttryck. Utvidga `trait Filter` så att alla filter kan ha kortkommando. Skriv på knappen vad kortkommandot är så att användaren kan upptäcka det.

**Uppgift 11. Kontrastfilter.** Om man applicerar kontrastfiltrering på en färgbild så kommer bilden att konverteras till en gråskalebild. (Man kan naturligtvis förbättra kontrasten i en färgbild och få en färgbild som resultat. Då behandlar man de tre färgkanalerna var för sig.) Många bilder lider av alltför låg kontrast. Det beror på att bilden inte utnyttjar hela det tillgängliga området 0–255 för intensiteten. Man får en bild med bättre kontrast om man "töjer ut" intervallet enligt följande formel (linjär interpolation):

```
val newIntensity = 255 * (intensity - 45) / (225 - 45)
```

Som synes kommer en punkt med intensiteten 45 att få den nya intensiteten 0 och en punkt med intensiteten 225 att få den nya intensiteten 255. Mellanliggande punkter sprids ut jämnt över intervallet  $[0, 255]$ . För punkter med en intensitet mindre än 45 sätter man den nya intensiteten till 0, för punkter med en intensitet större än 225 sätter man den nya intensiteten till 255. Vi kallar intervallet där de flesta pixlarna finns för  $[lowCut, highCut]$ . De punkter som har intensitet mindre än `lowCut` sätter man till 0, de som har intensitet större än `highCut` sätter man till 255. För de övriga punkterna interpolerar man med formeln ovan (45 ersätts med `lowCut`, 225 med `highCut`).

Det återstår nu att hitta lämpliga värden på `lowCut` och `highCut`. Detta är inte något som kan göras enkelt, eftersom värdena beror på intensitetsfördelningen hos bildpunkterna. Man börjar därför med att först beräkna bildens intensitetshistogram, dvs hur många punkter i bilden som har intensiteten 0, hur många som har intensiteten 1, . . . , till och med 255.

I de flesta bildbehandlingsprogram kan man sedan titta på histogrammet och interaktivt bestämma värdena på `lowCut` och `highCut`. Så ska vi dock inte göra här. I stället bestämmer vi oss för ett procenttal `cutOff`, som användaren kan ange som argument från terminalen, och som beräknar `lowCut` så att `cutOff` procent av punkterna i bilden har en intensitet som



är mindre än `lowCut` och `highCut` så att `cutOff` procent av punkterna har en intensitet som är större än `highCut`.

**Exempel:** antag att en bild innehåller 100 000 pixlar och att `cutOff` är 1.5. Beräkna bildens intensitetshistogram genom registrering av varje intensitet i en heltals-array

```
val histogram = Array.ofDim[Int](256)
```

och beräkna `lowCut` så att

```
histogram(0) + ... + histogram(lowCut) = 0.015 * 100000
```

så nära det går att komma, det blir troligen inte exakt likhet. Beräkna `highCut` på liknande sätt.

Sammanfattning av algoritmen:

1. Beräkna intensitetsmatrisen.
2. Beräkna bildens intensitetshistogram.
3. Argument från användaren användas som `cutOff`.
4. Beräkna `lowCut` och `highCut` enligt exempel ovan.
5. Beräkna den nya intensiteten för varje pixel enligt interpolationsformeln och lagra de nya pixlarna i `result`.

Skapa ett filter `Contrast` som implementerar algoritmen. I katalogen *images* kan bilden *moon.jpg* vara lämplig att testa, eftersom den har låg kontrast. Anmärkning: om `cutOff` sätts = 0 så får man samma resultat av denna filtrering som man får av `GrayScale`. Detta kan man se genom att studera interpolationsformeln.

**Uppgift 12. Eget filter.** Skapa ett eget valfritt filter. Till exempel så kan du skapa ett filter som tar fem argument, där de två första värdena representerar ett intensitetsintervall och de tre sista värdena representerar röd-, grön- och blå-komponenterna till en pixel som ska ersättas med denna färg då intensiteten ligger utanför det givna intervallet.

**Uppgift 13. Egna interaktiva verktyg.** Skapa valfria interaktiva redigeringsverktyg med mus- och tangentinput. Börja med ett markeringsverktyg som gör så att en rektangelformad del av bilden kan markeras med hjälp av musen. Gör det möjligt att applicera filter på den markerade delen av bilden. Du kan också göra så att argument till t.ex. Gauss-filtret kan ställas in med ett skjutreglage som du ritar under knappen och som kan regleras med mus eller piltangenter.



## Kapitel 13

# Repetition

Begrepp som ingår i denna veckas studier:

- ☐ träna på extensor
- ☐ redovisa projekt
- ☐ träna inför muntligt prov

## 13.1 Tips

### 13.1.1 Repetitionsämnen på begäran från tidigare år

- |   |     |
|---|-----|
| 1. namnanrop, värdeanrop                  | w03 |
| 2. funktionsvärde, funktionstyp och thunk | w03 |
| 3. <code>--classpath</code>               | w04 |
| 4. <code>import</code>                    | w04 |
| 5. Option                                 | w06 |
| 6. Try                                    | w06 |
| 7. <code>enum</code>                      | w07 |
| 8. avlusning, läsa felmeddelande          | w08 |
| 9. <code>given using</code>               | w11 |

### 13.1.2 Några extra önskemål från tidigare år (i mån av tid)

- |   |     |
|---|-----|
| 1. Closure ("fångad variabelrymd")      | w03 |
| 2. Skillnad på objekt och singelobjekt? | w04 |
| 3. Mönstermatchning.                    | w06 |
| 4. När använda vilken sekvenstyp?       | w07 |
| 5. Typhärledning.                       | w08 |
| 6. Komposition eller arv?               | w10 |

### 13.1.3 Några tumregler/tips vid val av abstraktion

Om du skulle behöva samla både attribut och metoder: Extensionsmetod, singelobjekt, case-klass, klass, trait eller abstrakt klass, eller enum?

- Om du vill lägga till en metod på befintlig typ utan att ha nya attribut, använd **extension**.
- Använd **object** om du behöver samla metoder (och variabler) i en modul som bara finns i en upplaga. Du får lokal namnrymd och punktnotation på köpet.
- Behöver du modellera **oföränderlig data**, använd en **case class** eller **enum**.
- Om du vill ha uppräknade värden som du vill kunna iterera över och matcha på i förseglad struktur, med värden i egen namnrymd, använd **enum**.
- Med **case class** och **enum** Du får då även innehållslighet och en massa annat godis på köpet!
- Behöver du **förändringsbart tillstånd** (eng. *mutable state*) använd en vanlig **class**. Det normala är att det föränderliga tillståndet (de attribut som är föränderliga) är **private** eller **protected** och att all uppdatering och avläsning av tillståndet sker indirekt genom metoder (getters/setters/...).
- Behöver du en abstrakt bastyp använd en **trait**, speciellt om du vill ha möjlighet till inmixning. Om du vill förhindra inmixning eller underlätta användning från Java, använd **abstract class**.

### 13.1.4 Tips om val av samling

Det är ofta enklare med oföränderliga samlingar med oföränderliga element och skapa nya samlingar vid förändring. Men för vissa algoritmer blir det enklare eller effektivare om du ändrar på plats i förändringsbar samling.

- Behöver du hantera värden i sekvens?
    - Om du klarar dig utan förändring av innehållet efter konstruktion: **val**-referens till Vector
    - Om du behöver ändra innehåll men **inte** antal element: **val**-referens till Array
    - Om du behöver ändra innehåll **och** antal element: **var**-referens till Vector och t.ex. metoden patch, eller **val**-referens till ArrayBuffer och t.ex. metoden insert
  - Behöver du hantera värden x som ska vara unika?
    - Oföränderlig: Set
    - Förändringsbar: **val**-referens till `scala.collection.mutable.Set`
  - Behöver du leta upp värden x: Int utifrån en nyckel av t.ex. String?
    - Oföränderlig: `Map[String, Int]`
    - Förändringsbar: **val**-referens till `scala.collection.mutable.Map[String, Int]`
- 

### 13.1.5 Före tentan:

1. Repetera övningar och labbar i kompendiet.
  2. Läs igenom föreläsningsanteckningar.
  3. Studera **snabbref mycket noga** så att du vet vad som är givet och var det står, så att du kan hitta det du behöver snabbt.
  4. Skapa och **memorera** en personlig **checklista** med programmeringsfel du brukar göra, som även inkluderar småfel, så som glömda parenteser och semikolon, och annat som en kompilator/IDE normalt hittar.
  5. Tänk igenom hur du ska disponera dina 5 timmar på tentan.
  6. Gör minst en extenta som om det vore **skarpt läge**:
    - (a) Avsätt 5 ostörda timmar (stäng av telefon, dator etc).
    - (b) Inga hjälpmedel. Bara snabbref.
    - (c) Förbered dryck och tilltugg.
- 

### 13.1.6 På tentan:

1. Läs igenom **hela** tentan först.  
**Varför?** Förstå helheten. Delarna hänger ihop.
2. Notera och begrunda specifika begrepp och definitioner.  
**Varför?** Begreppen är avgörande för förståelsen av uppgiften.
3. Notera förenklingar, antaganden och specialfall.  
**Varför?** Uppgiften blir mkt enklare om du inte behöver hantera dessa.

4. **Fråga** tentamensansvarig om du inte förstår uppgiften – speciellt om det finns miss-tänkta felaktigheter eller förmodat oavsiktliga oklarheter.  
**Varför?** Det är inte lätt att konstruera en ”perfekt” tenta.  
Du får fråga vad du vill, men det är inte säkert du får svar :)
  5. Läs specifikationskommentarerna och metodsSignaturerna i alla givna klass-specifikationer **mycket noga**.  
**Varför?** Det är ett vanligt misstag att förbise de ledtrådar som ges där.
  6. Återskapa din memorerade personliga checklista för vanliga fel som du brukar göra och avsätt tid till att gå igenom den på tentan. Varje fix plockar poäng!
  7. Lämna in ett försök även om du vet att lösningen inte är fullständig. Det gäller att plocka så många poäng det går. En ofullständig lösning kan ändå ge poäng.
  8. Om du har svårigheter kan det bli kamp mot klockan. Försök hålla huvudet kallt och prioritera utifrån var du kan plocka flest poäng. Ge inte upp! Ta en kort äta-dricka-paus för att få mer energi!
-

## 13.2 Övning examprep

**Uppgift 1.** *Gör klart ditt projekt.*

**Uppgift 2.** *Gör en extenta.*

**Uppgift 3.** *Förbered din projektredovisning.*

**Uppgift 4.** *Skapa dokumentation för ditt projekt. Läs mer i Appendix E om hur du skapar dokumentation.*

**Uppgift 5.** *Repetera övningar och laborationer.*

## Kapitel 14

# MUNTligt PROV

På schemalagd tid senast sista läsveckan i december ska du avlägga ett obligatoriskt muntligt prov för handledare. Du måste vara godkänt på alla laborationer för att få göra det muntliga provet. Syftet med provet är att kontrollera att du har godkänd förståelse för de begrepp som ingår i kursen. Du rekommenderas att förbereda dig noga inför provet, t.ex. genom att gå igenom grundläggande begrepp för varje kursmodul och repetera grundövningar och laborationer.

Provet sker som ett stickprov ur kursens innehåll. Du kommer att få några slumpvis valda frågor där du ombeds förklara några av de begrepp som ingår i kursen. Du får även uppdrag att skriva kod som liknar kursens övningar och förklara hur koden fungerar. Du kan träna på typiska frågor här: <https://cs.lth.se/pgk/muntabot/>

Om det visar sig oklart huruvida du uppnått godkänd förståelse kan du behöva komplettera ditt muntliga prov. Kontakta kursansvarig för information om omprov.

**Del III**

**Appendix**

# Appendix A

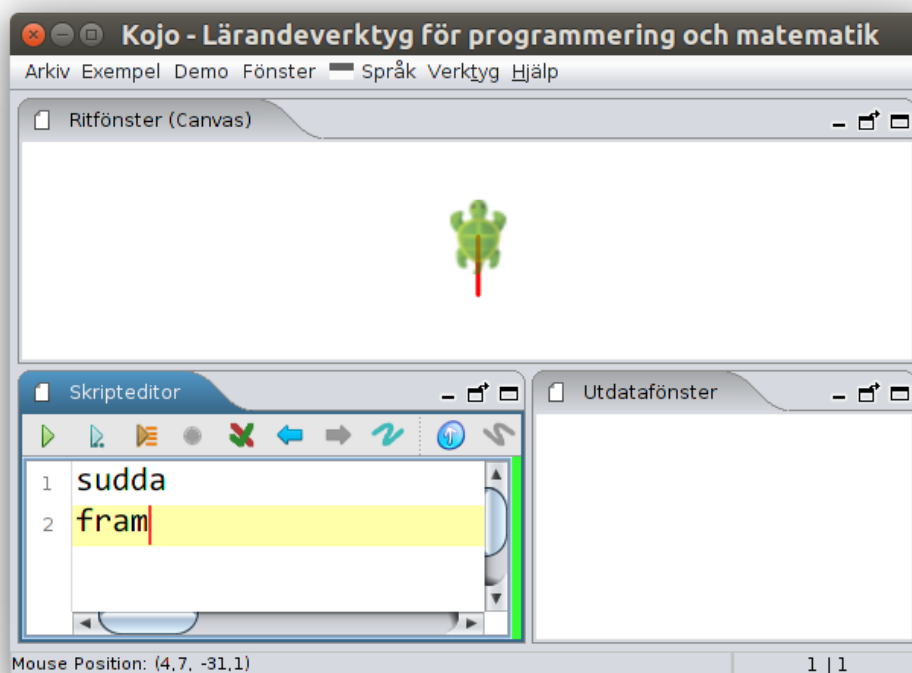
## Kojo

### A.1 Vad är Kojo?

Kojo<sup>1</sup> är en integrerad utvecklingsmiljö för Scala som är speciellt anpassad för programmeringsundervisning i grundskolan. Kojo används i LTH:s Science Center Vattenhallen för utbildning av grundskolelärare i programmering och vid skolbesök och annan besöksverksamhet, i vilken lärare och studenter vid LTH arbetar som handledare.

Kojo är öppen källkod och utvecklingsgemenskapen leds av Lalit Pant från Indien. I Kojo finns även lättillgängliga bibliotek som gör tröskeln lägre att programmera rörlig grafik och enkla spel.

Under kursens första laboration använder vi grafikbiblioteket i Kojo för att illustrera grundläggande begrepp, så som sekvens, alternativ, repetition och abstraktion.



**Figur A.1:** Den nybörjarvänliga utvecklingsmiljön Kojo för Scala på svenska.

<sup>1</sup>[en.wikipedia.org/wiki/Kojo\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Kojo_(programming_language))



## A.2 Använda grafikbiblioteket i Kojo

Kojo bygger på den beprövade pedagogiska idén med sköldpaddsgrafik (eng. *turtle graphics*)<sup>2</sup>, där du skriver program som styr en sköldpadda med en penna under magen. När sköldpaddan rör sig bildas ett streck av valfri färg på skärmen. Beroende på hur du bestämmer att sköldpaddan ska röra sig och vilken färg du bestämmer att pennan ska ha, kan du skapa olika intressanta bilder och samtidigt lära dig om programmeringens grunder.

Under kursens första laboration ska du använda grafikbiblioteket i Kojo tillsammans med editorn VS code och `scala-cli` i terminalen (se appendix B och C). Ladda ner filen `kojolib.scala` från <https://fileadmin.cs.lth.se/kojolib.scala> och spara i en ny katalog med hjälp av din webbläsare, eller via dessa kommandon (notera att det är stora bokstaven `O` och inte en nolla i optionen `-sLO`):

```
> mkdir w01-kojo
> cd w01-kojo
> curl -sLO https://fileadmin.cs.lth.se/kojolib.scala
```

Nu kan du starta Scala REPL och rita med Kojo så här:

```
> scala-cli repl .
Welcome to Scala 3.1.2 (17.0.2, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> fram; höger; fram; vänster
```

Du kan starta VS code i aktuellt bibliotek så här:

```
> code .
```

Skriv nedan program i VS code och spara det i samma katalog som den tidigare nedladdade filen, under ett nytt valfritt filnamn, t.ex. `rita.scala`:

```
@main def rita = { fram; höger; fram; vänster }
```

Kör ditt fristående program med:

```
> scala-cli run .
```

Du ska nu få upp ett fönster som heter Kojo Canvas med en sköldpadda som ritat två streck. När du stänger fönstret så avslutas programmet. Prova fler sköldpaddsfunktioner enligt tabell A.1.

I stället för att ladda ned filen `kojolib.scala` så kan du placera dess innehåll på lämpligt ställe i ditt program enligt nedan. Observera att raden som börjar med `//> using lib` ska vara en enda lång rad utan radbrytningar.

```
//> using scala "3"
//> using dep "net.kogics:kojo-lib:0.2.0,url=https://github.com/lunduniversity/introprog/releases/download/kojo-lib-0.2.0/kojo-lib-0.2.0.jar"

export net.kogics.kojo.Swedish.*, padda.*, CanvasAPI.*, TurtleAPI.*
```

Scala-koden för den svenska paddans api finns här:

[github.com/litan/kojo-lib/blob/main/src/main/scala/net/kogics/kojo/i18n/Swedish.scala](https://github.com/litan/kojo-lib/blob/main/src/main/scala/net/kogics/kojo/i18n/Swedish.scala)

<sup>2</sup>[https://en.wikipedia.org/wiki/Turtle\\_graphics](https://en.wikipedia.org/wiki/Turtle_graphics)

## A.3 Kojo Desktop

Kojo finns som fristående skrivbordsapplikation, kallad Kojo Desktop. Kojo Desktop innehåller en egen editor med syntaxfärgning för Scala, men fungerar ännu så länge bara för Scala 2. En av de synligaste skillnaderna mellan Scala 2 och Scala 3 är att klammerparenteser vid flerradiga funktioner är nödvändiga i Scala 2, medan Scala 3 har valfria klammerparenteser. Så om du använder Kojo Desktop behöver du komma ihåg att omgärda sekvenser av rader som hör ihop med { och }.

Kojo Desktop är förinstallerad på LTH:s datorer och körs igång med terminalkommandot `kojo` eller via applikationsmenyn. För instruktioner om hur du installerar Kojo Desktop på din egen dator se här: [lth.se/programmera/installera](http://lth.se/programmera/installera)

När du startar Kojo första gången, välj "Svenska" i språkmenyn och starta om Kojo. Därefter fungerar grafikfunktionerna på svenska enligt tabell A.1 på sidan 455. När du startat om Kojo inställt på svenska ser programmet ut ungefär som i figur A.1 på sidan 453.

Det finns ett antal användbara kortkommando som du hittar i menyerna i Kojo Desktop. Undersök speciellt Ctrl+Alt+Mellanslag som ger autokomplettering baserat på det du börjat skriva.

## A.4 Kojo i Webbläsaren

En begränsad variant av Kojo finns tillgänglig för programmering direkt i din webbläsare här: <http://kojo.lu.se/>

När du trycker på play-knappen så kompileras din kod på en server till Javascript via ScalaJS och därefter körs Javascript-koden i din webbläsare. Kojo på webben är också ännu så länge begränsad till Scala 2 och kräver att du omgärdar sekvenser av rader som hör ihop med { och }.

## A.5 Mer om Kojo

I detta dokument finns en enkel introduktion till Kojo:

"Introduction to Kojo" <http://www.kogics.net/kojo-ebooks#intro>

I tabell A.1, som fortsätter på efterföljande sidor, finns ett urval av kommando i Kojo på svenska och engelska.

**Tabell A.1:** Ett urval av funktioner i Kojo. Se även [lth.se/programmera](http://lth.se/programmera)

<i>Svenska / Engelska</i>	<i>Vad händer?</i>
sudda <code>clear()</code>	Ritfönstret suddas
fram <code>forward()</code>	Paddan går framåt 25 steg.
fram(100) <code>forward(100)</code>	Paddan går framåt 100 steg.
höger <code>right(90)</code>	Paddan vrider sig 90 grader åt höger.
höger(45) <code>right(45)</code>	Paddan vrider sig 45 grader åt höger.
vänster <code>left(90)</code>	Paddan vrider sig 90 grader åt vänster.

```
vänster(45)
left(45)

hoppa
hop()

hoppa(100)
hop(100)

hoppaTill(100, 200)
jumpTo(100, 200)

gåTill(100, 200)
moveTo(100, 200)

hem
home()

öster
setHeading(0)

väster
setHeading(180)

norr
setHeading(90)

söder
setHeading(-90)

mot(100,200)
towards(100, 200)

sättVinkel(90)
setHeading(90)

vinkel
heading

sakta(5000)
setAnimationDelay(5000)

println("hej")

textstorlek(100)
setPenFontSize(100)

båge(100, 90)
arc(100, 90)

cirkel(100)
circle(radie)

synlig
visible()

osynlig
invisible()

läge.x
position.x

läge.y
position.y

pennaNer
penDown()

pennaUpp
penUp()

pennanÄrNere

färg(rosa)
setPenColor(pink)

fyll(lila)
setFillColor(purple)
```

Paddan vrider sig 45 grader åt vänster.

Paddan hoppar 25 steg utan att rita.

Paddan hoppar 100 steg utan att rita.

Paddan hoppar till läget (100, 200) utan att rita.

Paddan vrider sig och går till läget (100, 200).

Paddan går tillbaka till utgångsläget (0, 0).

Paddan vrider sig så att nosen pekar åt höger.

Paddan vrider sig så att nosen pekar åt vänster.

Paddan vrider sig så att nosen pekar uppåt.

Paddan vrider sig så att nosen pekar neråt.

Paddan vrider sig så att nosen pekar mot läget (100, 200)

Paddan vrider nosen till vinkeln 90 grader.

Ger vinkelvärdet dit paddans nos pekar.

Gör så att paddan ritar jättesakta.

Skriver texten hej.

Paddan skriver med jättestor text nästa gång du gör skriv.

Paddan ritar en båge med radie 100 och vinkel 90.

Paddan ritar en cirkel med radie 100.

Paddan blir synlig.

Paddan blir osynlig.

Ger paddans x-läge

Ger paddans y-läge

Sätter ner paddans penna så att den ritar när den går.

Lyfter upp paddans penna så att den INTE ritar när den går.

Kollar om pennan är nere eller inte.

Sätter pennans färg till rosa.

Sätter ifyllnadsfärgen till lila.

<code>fill(genomskinlig)</code>	Gör så att paddan inte fyller i något när den ritar.
<code>setFillColor(noColor)</code>	
<code>bredd(20)</code>	Gör så att pennan får bredden 20.
<code>setPenThickness(20)</code>	
<code>sparaStil</code>	Sparar pennans färg, bredd och fyllfärg.
<code>saveStyle()</code>	
<code>laddaStil</code>	Laddar tidigare sparad färg, bredd och fyllfärg.
<code>restoreStyle()</code>	
<code>sparaLägeRiktning</code>	Sparar pennans läge och riktning
<code>savePosHe()</code>	
<code>laddaLägeRiktning</code>	Laddar tidigare sparad riktning och läge
<code>restorePosHe()</code>	
<code>siktePå</code>	Sätter på siktet.
<code>beamsOn()</code>	
<code>sikteAv</code>	Stänger av siktet.
<code>beamsOff()</code>	
<code>bakgrund(svart)</code>	Bakgrundsfärgen blir svart.
<code>setBackground(black)</code>	
<code>bakgrund2(grön,gul)</code>	Bakgrund med övergång från grönt till gult.
<code>setBackgroundV(green, yellow)</code>	
<code>upprepa(4){fram; höger}</code>	Paddan går fram och svänger höger 4 gånger.
<code>repeat(4){forward; right}</code>	
<code>avrunda(3.99, 2)</code>	Avrundar 3.99 till två decimaler, alltså 4.0
<code>slumptal(100)</code>	Ger ett slumptal mellan 0 och 99.
<code>slumptalMedDecimaler(100)</code>	Ger ett slumptal mellan 0 och 99.99999999
<code>systemtid</code>	Ger nuvarande systemklocka i sekunder.
<code>räknaTill(5000)</code>	Kollar hur lång tid det tar för din dator att räkna till 5000.

## Appendix B

# Terminalfönster

### B.1 Vad är ett terminalfönster?

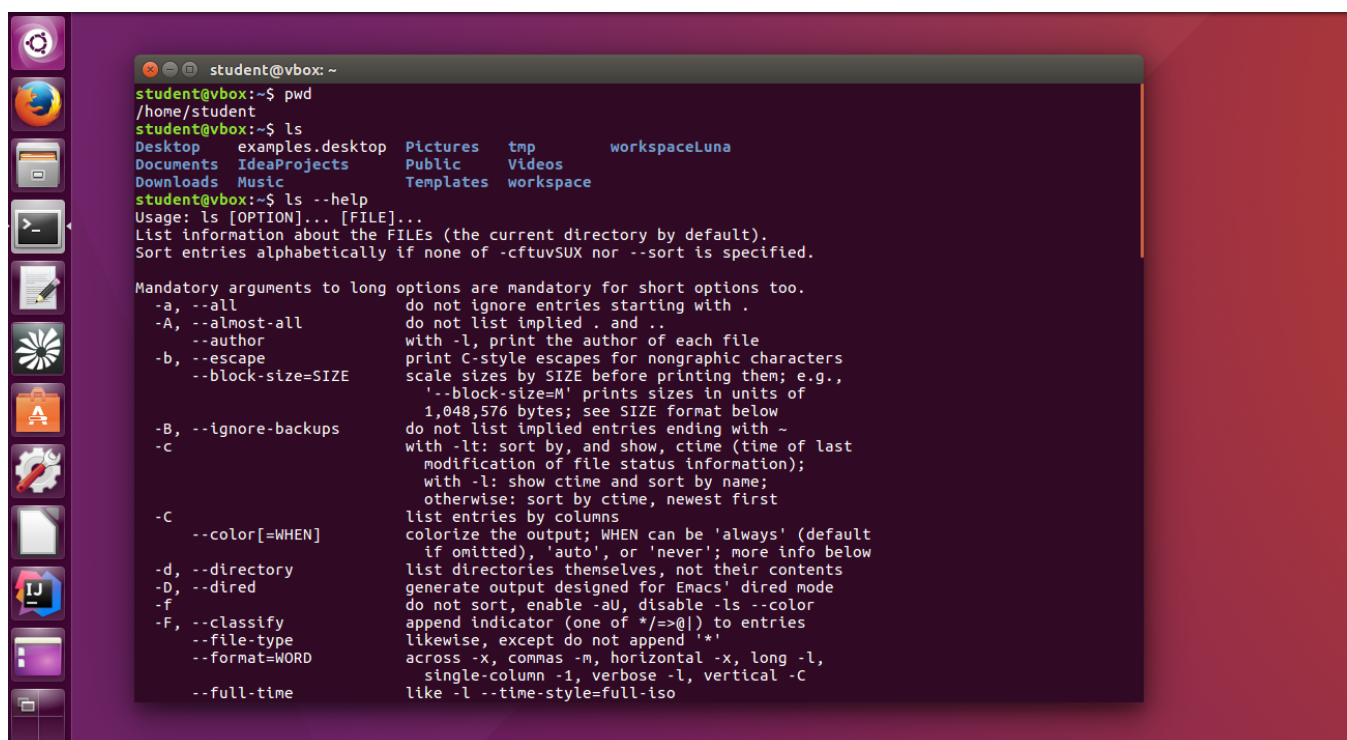
I ett terminalfönster kan man skriva kommandon som kör program och hanterar filer. När man programmerar använder man ofta terminalkommandon för att kompilera och exekvera sina program.

#### Terminal i Linux

I Ubuntu trycker du lättast **Ctrl+Alt+T** eller sök efter "terminal" i app-menyn. Då öppnas ett fönster med en blinkande markör som visar att det är redo att ta emot dina textkommando. Ett exempel på kommando är `ls` som skriver ut en lista med filer i den aktuella katalogen, så som visas i fig. B.1.

Det som visas i ett terminalfönster sköts av ett **kommandoskal** (eng. *command shell*), som är redo att ta emot kommando efter en prompt som slutar med ett `$`-tecken. När du skriver ett kommando och trycker Enter anropar kommandoskalet en kommandotolk som tolkar och utför dina kommandon. Om ett kommando inte kan tolkas, skrivs ett felmeddelande.

Det finns många användbara kortkommando, varav de viktigaste visas i tabell B.1. Det är



Figur B.1: Terminalfönster i Ubuntu öppnas med Ctrl+Alt+T.

bra om du lär dig dessa kortkommandon utantill så att ditt arbete i terminalen går snabbt och smidigt.

pil upp/ner	bläddra i kommandohistoriken
Tab	"auto-complete", fyll i resten baserat på vad du skrivit hittills
Tab Tab	två tryck på Tab listar flera alternativ, om så finnes
Ctrl+A	"ahead", flytta markören till början av raden
Ctrl+E	"end", flytta markören till slutet av raden
Ctrl+K	"kill", ta bort tecken från markören till radens slut
Ctrl+U	"undo", ta bort tecken från markören till början av raden
Ctrl+Y	"yank", sätt in det som senast togs bort
Ctrl+Z	"zleep", stoppa pågående process, skriv sedan bg för bakgrundskörning
Ctrl+L	rensa terminalfönstret
Ctrl+D	avsluta kommandoskalet

**Tabell B.1:** Viktiga kortkommandon i Linux terminalfönster.

Ctrl+C orsakar normalt ett avbrott av pågående process, men om du vill att Ctrl+C ska vara "Copy" som vanligt för att kopiera markerad text, kan du ställa om detta med terminalförnstrrets meny "Edit → Keyboard Shortcuts", eller liknande.

### PowerShell, Cmd och Linux i Microsoft Windows

Det finns flera olika sätt att köra terminalkommando i Windows:

- **Powershell.** I Microsoft Windows finns kommandotolken *Powershell* med speciell kommandosyntax. Den är inte Linux-baserad men det finns alias definierade för några vanliga Linux-kommandon, inkluderat `ls`, `cd` och `pwd`. Du startar Powershell t.ex. genom att trycka på Windows-knappen och skriva `powershell`. Du kan också, medan du bläddrar bland filer, klicka på filnamnsraden överst i filbläddraren och skriva `powershell` och tryck Enter; då startas Powershell i aktuellt katalog.
- **Cmd.** Det finns även i Windows den ursprungliga, gamla kommandotolken *Cmd* med helt andra kommandon. Till exempel skriver man i Cmd kommandot `dir` i stället för `ls` för att lista filer.
- **WSL.** I både Windows 10 och 11 kan du även köra Ubuntu-terminalen med hjälp av Windows Linux Subsystem (WSL), vilket rekommenderas, speciellt om du inte har möjlighet att göra s.k. dual boot<sup>1</sup>.
  - Se vidare här om hur du kan installera WSL under Windows, (WSL2 rekommenderas före WSL1 om din maskin klarar det):  
<https://docs.microsoft.com/en-us/windows/wsl/install>
  - Det finns även ett smidigt tillägg till VS Code som heter Remote-WSL som gör att du kan editera filer i Windows som finns i WSL, se vidare här:  
<https://code.visualstudio.com/docs/remote/wsl-tutorial>
- **Windows Terminal.** Den nya Microsoft-appen *Windows Terminal* rekommenderas oavsett om du använder Powershell, Cmd eller WSL. Läs mer här om hur du installerar Windows Terminal:  
<https://docs.microsoft.com/en-us/windows/terminal/>

<sup>1</sup>Läs mer om dual boot här och be gärna någon om hjälp som gjort det förr:  
<https://www.linuxtechi.com/dual-boot-ubuntu-22-04-and-windows-11/>

## Terminal i Apple macOS/OS X

Apple OS X och macOS är Unix-baserade operativsystem. De flesta vanliga terminalkommandon som fungerar i Linux fungerar också under Apple OS X och macOS. Du startar ett terminalfönster i Apples operativsystem genom att klicka på förstoringsglasat uppe till höger, skriva terminal, och trycka Enter.

## B.2 Vad är en path/sökväg?

När du skriver ett kommando i terminalen, eller kör vilket program som helst på din dator, behöver operativsystemet identifiera i vilken fil programmets maskinkod ligger innan programmet kan köras.

Lokaliseringen av filer sker med hjälp av en **sökväg** (eng. *path*), som anger en position i filsystemet. Ofta betraktas filsystemet som ett upp-och-ned-vänt träd, och kallas därför även "filträdet". Den "översta" positionen kallas "rot" (eng. *root*) och betecknas med ett enkelt snedstreck /. Kataloger som ligger i kataloger utgör förgreningar i trädet. En sökväg pekar ut vägar genom trädet som behövs för att nå "löven", som utgörs av själva filerna.

Du kan se var ett program ligger i Linux med hjälp av kommandot `which` enligt nedan.<sup>2</sup> Listan med kataloger i sökvägen avskiljs med snedstreck.

```
$ which java
/usr/lib/jvm/oracle_jdk8/bin/java
$ which ls
/bin/ls
```

En sökväg kan vara **absolut** eller **relativ**. En absolut sökväg utgår från roten och visar hela vägen från rot till destination, t.ex. `/usr/bin/firefox`, medan en relativ sökväg utgår från aktuellt katalog (där du "står") och börjar *inte* med ett snedstreck.

Alla operativsystem håller reda på en mängd olika sökvägar för att kunna hitta speciella filer i filträdet. Dessa sökvägar lagras i s.k. **miljövariabler** (eng. *environment variables*). Det finns en *speciell* miljövariabel som heter kort och gott **PATH**, i vilken alla sökvägar till de program finns, som ska vara tillgängliga för din användaridentitet direkt för exekvering genom sina filnamn, *utan* att man behöver ange absoluta sökvägar.

Du kan i Linux se vad som ligger i din PATH med kommandot `echo $PATH` medan man i Windows Powershell skriver `$env:Path` där det bara är första bokstaven som ska vara en versal. I Linux separeras katalogerna i sökvägen med kolon, medan Windows använder semikolon.

Ibland kan du behöva uppdatera din PATH för att program som du installerat och ska bli allmänt tillgängliga. Detta görs på lite olika sätt i olika operativsystem, för Linux se t.ex. här: [stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux](https://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux)

När man anger sökvägar finns några tecken med speciell betydelse:

---

<sup>2</sup>Skriv `gcm ls` i Windows Powershell för motsvarighet till `which ls`  
Eller skriv `New-Alias which get-command` för tillgång till kommandot `which` i Powershell.  
[stackoverflow.com/questions/63805/equivalent-of-nix-which-command-in-powershell](https://stackoverflow.com/questions/63805/equivalent-of-nix-which-command-in-powershell)

- ~ "tilde", din hemkatalog
- / "slash", snedstreck anger filträdet rot om det finns i början av sökvägen, men utgör katalogsavskiljare inuti sökvägen
- . en punkt anger aktuell katalog, där du "står"
- .. två punkter anger ett steg "upp" i filträdet
- " omgärda en sökväg med citationstecken, först och sist, om den innehåller annat än engelska bokstäver, t.ex. blanktecken
- \ *backslash+blanktecken* används för att beteckna mellanslag i sökvägar som *inte* omgärdas av citationstecken

### B.3 Några viktiga terminalkommando

I tabell B.2 finns en lista med några viktiga terminalkommando som är bra att lära sig utantill.

En introduktion till LTH:s datorer med exempel på hur du använder vanliga Linux-kommandon finns i denna skrift <http://www.ddg.lth.se/perf/unix/> som används i introduktionsveckan för nybörjare på datateknikprogrammet vid LTH.

På sajten <http://ss64.com/> finns en mer omfattande lista med användbara terminalkommando och tillhörande förklaringar för Linux (Bash), Windows (Powershell, Cmd) och Apple OS X (Bash).

ls	lista filer i aktuell katalog (alltså där du "står")
ls <i>p</i>	lista filer i katalogen <i>p</i>
ls -A	lista alla filer i aktuell katalog, även gömda
man ls	manual för kommandot ls; testa även man för andra kommandon!
cd <i>p</i>	"change directory", ändra aktuell katalog till <i>p</i>
pwd	"print working directory", skriv ut sökväg för aktuell katalog
cp <i>p1 p2</i>	"copy", kopiera filen med path <i>p1</i> till en ny fil kallad <i>p2</i>
mv <i>p1 p2</i>	"move", byt namn på filen <i>p1</i> till <i>p2</i>
rm <i>p</i>	"remove", ta bort filen <i>p</i>
rm -r <i>p</i>	"remove recursive", ta bort katalogen <i>p</i> med allt innehåll; var försiktig!
mkdir <i>p</i>	"make dir", skapa ett en katalog <i>p</i>
cat <i>p1 p2</i>	"concatenate", skriv ut hela innehållet i en eller flera filer <i>p1 p2 etc.</i>
less <i>p</i>	skriv ut innehållet i filen <i>p</i> , en skärm i taget
wget <i>url</i>	ladda ner <i>url</i> , t.ex. wget <a href="http://cs.lth.se/pgk/ws">http://cs.lth.se/pgk/ws</a> -o ws.zip
unzip <i>p</i>	packa upp <i>p</i> , t.ex. unzip ws.zip

**Tabell B.2:** Några viktiga terminalkommando i Linux. Med *p*, *p1*, *p2*, etc. avses en absolut eller relativ sökväg (eng. *path*), se avsnitt B.2.



## Appendix C

# Editera, kompilera och exekvera

### C.1 Vad är en editor?

En editor används för att redigera programkod. Det finns många olika editorer att välja på. Erfarna utvecklare lägger ofta mycket energi på att lära sig att använda favoriteditorns kortkommandon och specialfunktioner, eftersom detta påverkar stort hur snabbt kodredigeringen kan göras.

En bra editor har **syntaxfärgning** för språket du använder, så att olika delar av koden visas i olika färger. Då går det mycket lättare att läsa och hitta i koden.

Nedan listas några viktiga funktioner som man använder många gånger dagligen när man kodar:

- **Navigera.** Det finns flera olika sätt att flytta markören och bläddra genom koden. Alla editorer erbjuder sökmöjligheter, och de flesta editorer har även mer avancerade sökfunktioner där kodmönster kan identifieras och multipla sökträffar markeras över flera kodfiler.
- **Markera.** Att markera kod kan göras på många sätt: med piltangenter+Shift, med olika speciella menyalternativ, med mus + dubbelklick eller trippelklick, etc. I vissa editorer finns även möjlighet att ha multipla markörer så att flera rader kan editeras samtidigt.
- **Kopiera.** Genom Copy-Paste slipper du skriva samma sak många gånger. Kortkommandona Ctrl+C för Copy och Ctrl+V för Paste sitter i fingrarna efter ett tag. Man ska dock vara medveten om att det lätt blir fel när man kopierar en stor del som sedan ska ändras lite; många Copy-Paste-buggar kommer av att man inte är tillräckligt noggrann och ofta är det bättre att skriva från grunden i stället för att kopiera så att du hinner tänka efter medan du skriver.
- **Klipp ut.** Genom Ctrl+X för Cut och Ctrl+V för Paste, kan du lätt flytta kod. Att skriva kod är en stegvis process där man gör många förändringar under resans gång för att förbättra och vidareutveckla koden. Att flytta på kod för att skapa en bättre struktur är mycket vanligt.
- **Formatering.** Med indragningar, radbrytningar och nästlade block i flera nivåer får koden struktur. Många editorer kan hjälpa till med detta och har speciella kortkommandon för att ändra indragningsnivå inåt eller utåt.
- **Parentesmatchning.** Olika former av parenteser, ( { [ ] } ], behöver matchas för att koden ska fungera; annars går kompilatorn ofta helt vilse och konstiga felmeddelanden kan peka på helt fel plats i koden. En bra kodeditor kan hjälpa dig att markera

vilka parentespar som hör ihop så att du undviker att spendera för mycket tid med att leta efter en parentes som saknas eller står i vägen.

### C.1.1 Välj editor

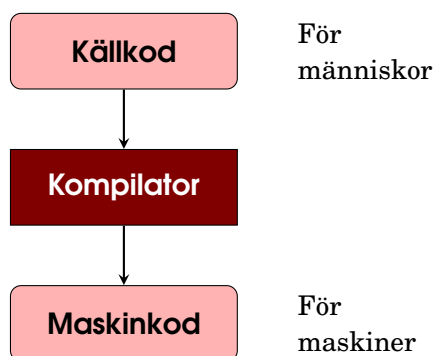
I tabell C.1 visas en lista med några populära editorer. Det är en stor fördel om din favoriteditor finns på flera plattformar så att du har nytta av dina förvärvade färdigheter när du behöver växla mellan olika operativsystem.

I denna kurs rekommenderas Visual Studio **code**, eftersom den är öppen, gratis och finns för Linux, Windows och Mac, och har bra stöd för Scala och Java. Men om du redan är van vid någon annan av editorerna i tabell C.1 så fungerar de också bra.

En integrerad utvecklingsmiljö (eng. *integrated development environment*, *IDE*), se appendix H, erbjuder många avancerade funktioner som hjälper dig att koda effektivt när du väl lärt dig handgreppen. VS code har numera flera IDE-funktioner, och gränsen mellan en renodlad editor och en IDE, så som IntelliJ och Eclipse, är inte längre lika tydlig som förr.

## C.2 Vad är en kompilator?

En **kompilator** (eng. *compiler*) är ett program som läser programtext och översätter den till exekverbar maskinkod, så som visas i figur C.1. Programtexten som kompileras kallas källkod och utgörs av text som följer reglerna för ett programmeringsspråk, till exempel Scala eller Java.



**Figur C.1:** En kompilator översätter från källkod till maskinkod.

Vissa kompilatorer genererar kod som kan köras av en processor direkt, medan andra kompilatorer genererar ett mellanformat som tolkas under exekveringen. Det senare är fallet med Java och Scala, vilket möjliggör att programmet kan kompileras en gång för alla plattformar och sedan kan programmet köras på all de processorer till vilka det finns en s.k. virtuell maskin för Java (eng. *Java Virtual Machine*, *JVM*). Den kod som genereras av en kompilator för JVM kallas **bytekod**.

Om kompileringen inte lyckas skriver kompilatorn ut ett felmeddelande och ingen maskinkod genereras. Det är inte lätt att bygga en kompilator som ger bra felmeddelanden i alla lägen, men felmeddelandet ger oftast goda ledtrådar till felorsaken efter att man lärt sig tolka det programmeringsspråksspecifika vokabulär som kompilatorn använder.

Även om programmet kompilerar utan felmeddelande och genererar exekverbar maskinkod, är det vanligt att programmet ändå inte fungerar som det är tänkt. Ibland är det mycket svårt att lista ut vad problemet beror på och man kan behöva göra omfattande undersökningar av vad som händer under körningen, genom att t.ex. skriva ut olika variablers värden eller på annat sätt ändra koden och se vad som händer. Denna process kallas felsökning

**Tabell C.1:** Några populära editorer. I kursen rekommenderas VS Code.

<i>Editor</i>	<i>Beskrivning</i>
VS Code	<p>Öppen, fri och gratis. Finns för Linux, Windows, &amp; Mac. Är förinstallerad på LTH:s Linux-datorer och startas med kommandot <code>code</code>. Öppen-källkodsprojektet startades av Microsoft och har en aktiv gemenskap med många utvecklare och många användbara tillägg (eng. <i>extensions</i>). Sök efter tillägget <code>scalameta.metals</code> och installera så får du syntaxfärgning och många andra IDE-funktioner för Scala.</p> <p><a href="https://code.visualstudio.com/">https://code.visualstudio.com/</a>  <a href="https://scalameta.org/metals/docs/editors/vscode/#installation">https://scalameta.org/metals/docs/editors/vscode/#installation</a></p>
Gedit	<p>Öppen, fri och gratis. Lätt att lära men inte så avancerad. Är förinstallerad på LTH:s Linux-datorer och startas med kommandot <code>gedit</code>.</p> <p><a href="https://wiki.gnome.org/Apps/Gedit">https://wiki.gnome.org/Apps/Gedit</a></p>
Nano	<p>Öppen, fri och gratis. En simpel editor för enkla småjobb i terminalen. Är förinstallerad på de flesta Linux-system på planeten Jorden. Startas med kommandot <code>nano</code>.</p> <p><a href="https://www.nano-editor.org/">https://www.nano-editor.org/</a></p>
Notepad++	<p>Öppen, fri och gratis. Utvecklad speciellt för Windows men finns även för Linux.</p> <p><a href="https://notepad-plus-plus.org/">https://notepad-plus-plus.org/</a>  <a href="https://snapcraft.io/notepad-plus-plus">https://snapcraft.io/notepad-plus-plus</a></p>
Vim	<p>Öppen, fri och gratis. Hög inlärningströskel. Finns för Linux, Windows, &amp; Mac. Är förinstallerad på LTH:s Linux-datorer och startas med kommandot <code>vim</code>. Med Scala Metals (se länk nedan) får du IDE-liknande funktioner. Du avslutar vim genom att trycka <code>Escape</code> och sedan skriva <code>:q</code> och trycka <code>Enter</code>.</p> <p><a href="http://www.vim.org/">http://www.vim.org/</a>  <a href="https://scalameta.org/metals/docs/editors/vim.html">https://scalameta.org/metals/docs/editors/vim.html</a></p>
Emacs	<p>Öppen, fri och gratis. Hög inlärningströskel. Finns för Linux, Windows, &amp; Mac. Är förinstallerad på LTH:s Linux-datorer och startas med kommandot <code>emacs</code>. Med Scala Metals (se länk nedan) får du IDE-liknande funktioner.</p> <p><a href="http://www.gnu.org/software/emacs/">http://www.gnu.org/software/emacs/</a>  <a href="https://scalameta.org/metals/docs/editors/emacs.html">https://scalameta.org/metals/docs/editors/emacs.html</a></p>
Sublime Text	<p>Sluten källkod. Gratis att prova på, men programmet föreslår då och då att du köper en licens. Finns för Linux, Windows, &amp; Mac. Med Scala Metals (se länk nedan) får du IDE-funktioner.</p> <p><a href="http://www.sublimetext.com/3">http://www.sublimetext.com/3</a>  <a href="https://scalameta.org/metals/docs/editors/sublime.html">https://scalameta.org/metals/docs/editors/sublime.html</a></p>

eller avlusning (eng. *debugging*), och är en väsentlig del av all systemutveckling. Läs mer om debugging i Appendix D.

En uttömmande testning av ett större program, som kör programmets *alla* möjliga exekveringsvägar, är i praktiken omöjlig att genomföra inom rimlig tid, då antalet kombinationsmöjligheter växer mycket snabbt med storleken på programmet. Därför är kompilatorn ett mycket viktigt hjälpmedel. Med hjälp av den analys och de kontroller som görs av kompilatorn kan många buggar, som annars vore mycket svåra att hitta, undvikas och åtgärdas i kompileringsfasen, redan *innan* man exekverar programmet.

## C.3 Java JDK

Scala, Java och flera andra språk använder Java-plattformen som exekveringsmiljö. Om man inte bara vill köra program som andra har utvecklat, utan även utveckla egna program som fungerar i denna miljö, behöver man installera Java Development Kit (JDK). Detta utvecklingspaket innehåller flera delar, bland annat:

- Kompilatorn `javac` kompilerar Java-program till bytekod som lagras i klassfiler med filnamnsändelsen `.class`.
- Exekveringsmiljön Java Runtime Environment (JRE) med kommandot `java` som drar igång den virtuella javamaskinen (Java Virtual Machine) som kan ladda och exekvera bytekod lagrade i klassfiler.
- Programmet `jar` som packar ihop många sammanhörande klassfiler till en enda jar-fil som lätt kan distribueras via nätet och sedan köras med `java`-kommandot på alla maskiner med JRE.
- Programmet `javap` som läser klassfiler och skriver ut vad de innehåller i ett format som kan läsas av människor (ett sådant program kallas *disassembler*).
- I JDK ingår också en mycket stor mängd färdiga programbibliotek med stöd för nätverkskommunikation, filhantering, grafik, kryptering och en massa annat som behövs när man bygger moderna system.

Du kan läsa mer om Java och dess historik här:

[https://en.wikipedia.org/wiki/Java\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Java_(programming_language))

### C.3.1 Kontrollera om du har JDK installerat

Öppna ett terminalfönster (se appendix B) och skriv (observera det avslutande `c:et` i `javac`):

```
javac -version
```

Då ska något som liknar följande skrivas ut, där `x` och `y` är siffror:

```
javac 17.x.y
```

Om utskriften säger att `javac` saknas, installera JDK enl. nedan.

Vi använder alltså JDK 17 i kursen. Det går också bra att använda de äldre versionerna JDK 8 och JDK 11, men JDK 9 eller 10 fungerar inte med alla verktyg vi använder och senare versioner än 17 kan också ge problem. Läs mer under "Verktyg" på kurshemsidan.

### C.3.2 Installera JDK

Det finns flera JDK-distributioner att välja mellan, varav OpenJDK och Oracle JDK är två exempel. Vi använder OpenJDK i kursen, som kan installeras via <https://adoptium.net/temurin/releases/?version=17>.

Om du installerar alla Scala-verktyg med hjälp av Coursier enligt instruktioner på kurs-hemsidan under ”Verktyg”, <http://cs.lth.se/pgk/verktyg> så kommer JDK att installeras automatiskt (om du inte redan har JDK).

## C.4 Scala

Scala använder Java Virtual Machine (JVM) som exekveringsmiljö, men går även att köra i browsern med hjälp av ScalaJS-kompilatorn som kompilerar från Scala till JavaScript. I denna kurs använder vi i Scala på JVM. I en Scala-installation ingår bl.a. kompilatorn `scalac` och även ett interaktivt kommandoskal kallat Scala REPL (se nedan C.4.2) där du kan testa din Scala-kod rad för rad och se vad som händer direkt.

Den officiella hemsidan för Scala finns här: <http://www.scala-lang.org/>

Du hittar mer om Scalas historik och annan bakgrundsinformation här: [en.wikipedia.org/wiki/Scala\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Scala_(programming_language))

### C.4.1 Installera Scala

Scala finns förinstallerat på LTH:s datorer. På kurs-hemsidan under ”Verktyg” finns detaljerade instruktioner om hur du installerar Scala på din egen dator: <http://cs.lth.se/pgk/verktyg>

### C.4.2 Scala Read-Evaluate-Print-Loop (REPL)

För många språk, t.ex. Scala och Python, finns det ett interaktivt program ämnat för terminalen som gör det möjligt att exekvera enstaka programrader och direkt se effekten. Ett sådant program kallas *Read-Evaluate-Print-Loop* (REPL), eftersom det läser och tolkar en rad i taget. Resultatet av evalueringen av din kod skrivs ut i terminalen och därefter är kommandoskalet redo för nästa kodrad.

Kursens övningar bygger till stor del på att du använder Scala REPL för att undersöka principer och begrepp som ingår i kursen genom dina egna kodexperiment. Även när du på labbarna utvecklar större program med en editor och en IDE, är det bra att ha Scala REPL till hands. Då kan du klistra in delar av programmet du håller på att utveckla i Scala REPL och stegvis utveckla delprogram, som till slut fungerar så som du vill.

I Scala REPL får du se typinformation för variabler och metoder, vilket är till stor hjälp när man försöker lista ut vad en kodrad innebär. Genom att öva upp din förmåga att dra nytta av Scala REPL, kommer din produktivitet öka.

Du startar Scala REPL med kommandot `scala` och skriver Scala-kod efter prompten `scala>` och kompilering+exekvering sker när du trycker Enter.

```
> scala
Welcome to Scala 3.1.2 (17.0.2, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> 41 + 1
val res0: Int = 42
```

Varje evaluerat värde sparas i en ny variabel, här `res0`.

Om du skriver en ofullständig rad fortsätter editeringen på nästa rad. Du kan navigera mellan raderna med pil-upp- och pil-ner-tangenterna. När du avslutar med en rad som gör din kod fullständig så kompileras och exekveras alla raderna. Du kan avbryta flerradsediteringen i förtid genom skriva ett semikolon ; och sen trycka Enter. Vill du fortsätta editeringen med en ny rad och förhindra för tidig evaluering så tryck Esc+Enter. Escape-tangenten finns överst till vänster på tangentbordet. Se exempel nedan:

```
scala> def fleraRader = 42 // Esc+Enter ger ny rad
|   + "ny rad".length // fortsättningsrad, avsluta med Enter
```

Beroende på vilket operativsystem du kör så kan även andra tangentkombinationer fungera för att starta ny rad i REPL; prova t.ex. Linux: Left Alt+Enter, Windows: Left Alt + Shift + Enter, VS Code Terminal i Windows Left Alt + Enter, VS Code Terminal i MacOS: Option + Enter.

Många av de kortkommandon som fungerar i terminalens kommandoskal, fungerar också i Scala REPL. Gå gärna igenom listan i tabell B.1 på sidan 459, och testa vad som händer i Scala REPL. Om du tränar upp din fingerfärdighet med dessa kortkommandon, går ditt arbete i Scala REPL väsentligt snabbare.

Med kommandot :help får du se en lista med specialkommandon för Scala REPL:

```
The REPL has several commands available:

:help           print this summary
:load <path>    interpret lines in a file
:quit           exit the interpreter
:type <expression> evaluate the type of the given expression
:doc <expression> print the documentation for the given expression
:imports        show import history
:reset [options] reset the repl to its initial state, forgetting all session entries
:settings <options> update compiler options, if possible
```

Du kan också starta Scala REPL med hjälp av kommandot `scala-cli repl .` med ett blanktecken och en punkt på slutet. Punkten gör att alla `.scala`-filer som finns i aktuell katalog kompileras av Scala CLI och görs tillgänglig för användning i REPL.

### C.4.3 Kompilera och kör med Scala Command Line Interface

Det finns sedan 2022 ett nytt smidigt kommandoradsgränssnitt (eng. *command line interface*) för att kompilera, exekvera och paketera Scala-program som kallas *Scala CLI*. Om du installerar Scala-verktygen enligt instruktioner på kurshemsidan under "Verktyg", <http://cs.lth.se/pgk/verktyg> så medföljer Scala CLI.

Här finns några användbara kommandon:

- Första gången du kör en nyinstallerad Scala CLI-installation så kör detta kommando så att du får tillgång till smidiga kompletteringar med TAB-tangenten:  
`scala-cli install completions`
- Med hjälp av detta kommando kan du förbereda VS Code för samverkan med Scala CLI (notera blanktecken och avslutande punkt):  
`scala-cli setup-ide .`  
Kör ovan kommando innan du startar VS Code första gången med `code .` i aktuell katalog, eller avsluta VS Code och kör ovan kommando och starta VS Code igen med `code .` i aktuell katalog.
- Scala CLI kan köra igång REPL i aktuell katalog med dina Scala- och Java-program automatiskt kompilerade och tillgängliggjorda i REPL med hjälp av nedan kommando.

Med optionen `-S` anger du vilken version av Scala du vill köra:

```
scala-cli repl . -S 3
```

- I stället för att ange Scala-version med optionen `-S` på kommandoraden kan du inuti ditt program, på första raden, skriva denna ”magiska” kommentar:

```
//> using scala "3.1.2"
```

Då kommer Scala CLI att automatiskt välja (och vid behov ladda ned) önskad version av Scala-kompilatorn (notera `>` efter `//`):

- Kompilera alla Scala- och Java-program i aktuell katalog och se eventuella felmeddelanden. Med hjälp av `--watch` (kan förkortas till `-w`) så kompileras alla filer automatiskt om så fort ändringar sparas i VS Code (kortkommando `Ctrl+S`):

```
scala-cli compile . --watch
```

- Kör Scala- och Java-program i aktuell katalog med start av den topp-nivå-**def** som är märkt **@main** (om det finns flera får du en frågan om vilken **@main def** du vill köra):

```
scala-cli run .
```

- Skapa en exekverbar fil:

```
scala-cli package .
```

- Skapa en kopia av ditt projekt med katalogstruktur och filer anpassade för byggverktyget `sbt` (se Appendix F):

```
scala-cli export . --sbt --output ../nameofnewprojdir
```

Ändra katalognamnet `nameofnewprojdir` till valfritt nytt namn på en katalog som inte existerar. Notera de dubbla punkterna som gör att nya katalogen hamnar på samma nivå som ditt nuvarande projekt, och *inte* i din aktuella katalog (för att undvika att dubletter av dina scala-filer ger kompileringsfel).

- Om du skriver `scala-cli help` så får du se vad du mer kan göra.

Läs mer om Scala CLI i Appendix F.2 och här:

<https://scala-cli.virtuslab.org/>







reläer, uppdagades en död nattfjäril ihjälklämd mellan drivankaret och spolen i ett relä, som orsakade att programmet inte kunde exekveras korrekt.

### D.1.1 Olika sorters fel

När man ska lära sig mer om fel i programvarubaserade system, och hur de kan åtgärdas, är det viktigt att noga skilja på **misstag** (eng. *mistake*), **felorsak** (eng. *fault*) och **felyttring** (eng. *failure*). Med "misstag" menar vi här ett fel som begås av människor (utvecklare, systemadministratörer, operatörer, användare, etc.) medan de skapar och använder ett programvarusystem.

Det kan bli fel i olika delar av processen:

- **Kravfel** uppstår medan man tänker ut vad systemet ska göra och då misstar sig angående användarnas behov och önskemål.
- **Designfel** uppkommer när man utformar systemets struktur på ett dåligt sätt.
- **Implementeringsfel** begås när man programmerar och skriver felaktiga kodrader.
- **Testfel** förekommer vid provkörning av systemet då testkoden är felaktig och därför ger falskt alarm om "fel", trots att beteendet egentligen är korrekt.
- **Operatörsfel** sker när systemet lämnas över till de, som ska installera och köra systemet i skarp produktion, och där systemdriften (eng. *operations*, "ops") sköts på ett sätt som får problematiska konsekvenser.
- **Användarfel** händer då användarna ger felaktig indata, eventuellt i strid med riktlinjerna för hur systemet ska användas, som systemet inte klarar att hantera korrekt, varpå mer eller mindre allvarliga felbeteenden hos systemet följer.

I olika delar av utvecklingsprocessen kan alltså misstag begås som, antingen omedelbart, eller någon gång i framtiden, kan orsaka fel. Men det är inte säkert att ett fel någonsin kommer att märkas. Kanske kommer de felaktiga kodraderna, som *skulle* kunna orsaka ett fel, aldrig att exekveras. Eller så kommer ingen användare att någonsin vilja använda systemet så som stipuleras av (onödiga) krav. Det är alltså först när fel *yttrar* sig vid exekvering som misstag märks.

Fel kan också kategoriseras utifrån *hur* de upptäcks i utvecklingsprocessen. Man brukar skilja på fel upptäckta vid granskning, kompileringsfel och exekveringsfel, som diskuteras nedan:

- Fel upptäckta vid **granskning**. Ett effektivt sätt att upptäcka fel är att människor noga läser igenom sin egen, och andras kod och försöker leta efter möjliga problem och brister. Man blir ofta "hemmablind" när det gäller ens egen kod. Därför kan någon annans, oberoende granskning med "nya, friska" ögon vara mycket fruktbar. I samband med kodgranskning kan man med fördel försöka bedöma huruvida koden är lätt att läsa, lätt att ändra i eller om koden har andra viktiga kvaliteter som har betydelse för den framtida utvecklingen av koden. Ofta hittar man vid granskning även enkla programmeringsmisstag, så som felaktiga villkor och loop-räknare som inte räknas upp på rätt sätt etc.
- **Kompileringsfel** uppkommer under kompilering och upptäcks tack vare kontroller som sker av kompilatorn.

Vid kompileringsfel får man också ofta av kompilatorn reda på *var* i koden det är fel och *varför* det är fel, så att sökandet efter felorsaken och åtgärdandet av misstaget underlättas. Men ibland är felmeddelandet från kompilatorn missvisande och pekar på helt fel ställe i koden, så det gäller att inte alltid lita blint på det kompilatorn skriver. Dessutom är felmeddelanden från kompilatorn ofta uttryckta i termer av språkets syntaktiska och semantiska regler och det tar tid att lära sig tolka kompilatorers felmeddelanden. Att skapa kompilatorer som ger bra felmeddelande är ett svårt problem som studeras inom den datavetenskapliga disciplinen *kompilorteknik*, vilken du kan lära mer om i kurser på avancerad nivå.

Olika programmeringsspråk erbjuder olika stora möjligheter att göra kontroller vid kompileringstid. En kompilator för ett språk med ett avancerat typsystem, som till exempel Scala, ger förhållandevis stora möjligheter att identifiera fel redan under kompileringen, medan man med ett språk med ett svagare typsystem, till exempel Javascript, får förlita sig på prestandahämmande kontroller som kompilatorn genererar i maskinkoden eller som du själv väljer att lägga in i källkoden för säkerhets skull.

- **Exekveringsfel**, även kallat körtidsfel (eng. *runtime error*), sker medan programmet körs. Det kan krävas viss, specifik indata under specifika exekveringsomständigheter (en viss processor, en viss minnesstorlek, en viss nätverkskapacitet etc.) för att ett exekveringsfel ska yttra sig. När ett exekveringsfel väl yttrar sig, kan olika saker hända:
  - **Exekveringen ger oönskat resultat.** Det är inte säkert att ett exekveringsfel avbryter exekveringen; det är vanligt att felet ”bara” resulterar i inkorrekt utdata eller på annat sätt ger dålig kvalitet. För att upptäcka detta innan systemet sätts i drift, är det allmän praxis att man skriver noga uttänkta **testfall** och analyserar **testresultat** från exekveringen av testfallen i detalj genom att undersöka utdata i jämförelse med önskat resultat eller med vad som anses vara en tillräckligt hög kvalitetsnivå.
  - **Exekveringen hänger sig** (eng. *hang*). Ibland yttrar sig fel genom att inget alls ser ut att hända under exekveringen, vilket kan beror på t.ex.:
    - \* en **oändlig loop**, som aldrig blir färdig,
    - \* att det går **väldigt långsamt** eftersom bearbetningen av indata tar orimligt lång tid,
    - \* att programmet **väntar på indata** som aldrig kommer,
    - \* att olika jämlöpande delar av programmet väntar på varandra så att ett **dödläge** (eng. *deadlock*) uppstår.
 När exekveringen hänger sig och man inte orkar vänta längre på att något ska hända, är det bara att brutalt avbryta exekveringen genom något lämpligt kommando som erbjuds i din körmiljö.<sup>3</sup> I värsta fall får man stänga av strömmen.
  - **Exekveringen kraschar** (eng. *crash*). Ibland blir det ett plötsligt tvärstopp och exekveringen avbryts med ett körtidsfelmeddelande. Detta kan bero på t.ex.:
    - \* att **minnet är slut**, antingen är det parameterminnet för funktionsanrop (eng. *stack memory*) som tagit slut eller så är minnet för allokering av objekt som skapas under programmets gång (eng. *heap memory*) fullt,
    - \* misstaget att försöka referera en **null-referens** som inte refererar till något objekt, utan har värdet **null**, vilket resulterar i *null pointer exception*,

<sup>3</sup>kill -9 <pid>, Ctrl+C, Ctrl+Shift+C, Ctrl+Z eller något annat beroende på körmiljö.

- \* att ett s.k. **undantag** har ”kastats” (eng. *throw exception*) genom att den som skrivit programmet medvetet kodat så att ett oönskat feltillstånd *ska* orsaka en krasch, om inte undantaget ”fångas” (eng. *catch*) och hanteras av omgivande kod.

När systemet kraschar får man en lista med den aktuella kedjan av funktionsanrop i en **stackspårning** (eng. *stack trace*). Man kan också begära en utskrift av hela innehållet i minnet vid kraschen (eng. *memory dump*), men en sådan kan vara svår att tolka.

När systemet ger oönskade resultat, hänger sig eller kraschar, får man försöka återskapa exekveringsfelet i en omkörning och, med hjälp av instrumentering eller en debugger, försöka lista ut vad som händer precis *innan* exekveringsfelet uppstår, se avsnitt D.5.

I kursen *Programvarutestning* (eng. *Software Testing*) lär du dig mer om systematiska metoder för att testa system så att fel kan förebyggas, identifieras och åtgärdas.

### Bugg eller feature?

När ett (eventuellt) fel upptäcks, kan det vara på sin plats att först ställa sig några grundläggande frågor:

- Är detta verkligen ett ”fel” eller är det egentligen ett avsett beteende? Det är inte alltid självklart om det är en bugg eller en medvetet skapad systemegenskap/funktion (eng. *feature*).
- Är det kanske testfallet som har felaktig testkod, medan koden som testas egentligen fungerar alldeles utmärkt? Sådan problem kan vara speciellt svåra att lösa, då man ofta letar på fel ställe efter orsaken.
- Om buggen rör någon kvalitetsegenskap hos systemet kan man fråga sig: Var går egentligen gränsen för ”fel”? Är detta bra nog givet vad det kostar att förbättra kvaliteten? Kvalitetskrav berör egenskaper hos ett program som kan uttryckas på en glidande skala, där något kan vara mer eller mindre *bra* eller *dåligt* ur olika synvinklar. Sådana krav leder ofta till viktiga men svåra avvägningsbeslut under design och implementation. Dessutom kan testresultat bli svårbedömda och det kan finnas olika åsikter om huruvida ett eventuellt fel är en bugg eller inte.

Här är några exempel på kvalitetskrav:

- **Prestandakrav** (eng. *performance requirements*) avser hur snabbt och effektivt programmet ska arbeta under olika omständigheter.
- **Kapacitetskrav** (eng. *capacity requirements*) avser hur mycket data systemet ska klara av under olika omständigheter.
- **Användbarhetskrav**<sup>4</sup> (eng. *usability requirements*) avser krav på hur lättanvänt systemet ska vara för en given användarkategori.

I kursen *Kravhantering* (eng. *Software Requirements Engineering*) lär du dig mer om att identifiera, specificera och följa upp kvalitetskrav.

---

<sup>4</sup>[sv.wikipedia.org/wiki/Användbarhet](https://sv.wikipedia.org/wiki/Användbarhet)

## Felärendehanteringsverktyg

Det är allmän praxis i industriell systemutveckling att använda sig av ett felärendehanteringsverktyg (eng. *issue tracker*) så att samarbetande utvecklare får stöd i att hålla reda på alla uppkomna fel och problem (eng. *issue*). Många av de populära kodlagringsplatserna som finns på nätet, så som GitLab, GitHub och BitBucket (se avsnitt G.3), erbjuder felärendehanteringsfunktioner. Dessa kan till exempel vara:

- hantering och sammanställning av alla olika ärendetillstånd, så att man kan se vilka issues som är i tillstånden *Open* eller *Closed*,
- tillordning av ärende till specifika personer som ska åtgärda problemet,
- gradering av ärende i olika allvarlighetsgrader,
- meddelandegenerering till inblandade personer när ett ärende kommenteras eller ändrar tillstånd.

## D.2 Att förebygga fel

Även om det nästan är oundvikligt att låta buggar slinka in i koden allteftersom den blir mer och mer komplex, är det ändå viktigt att lägga stor möda vid att försöka undvika att så sker. Det är ofta mycket bättre investerad tid att jobba med buggförebyggande åtgärder medan du skapar koden, än att jaga buggar som skulle kunna ha undvikts med allmän noggrannhet och stramare disciplin i kodningen. Nedan sammanfattas några åtgärder som kan hjälpa till att minska mängden fel.

- **Skapa begriplig kod.** Grunden för att undvika buggar är anstränga sig att skriva begriplig kod som är lätt att läsa. Detta är en ständig kamp; kodens komplexitet växer för varje tillägg och med jämna mellanrum behövs omstruktureringar (eng. *refactoring*) för att bibehålla en god struktur som underlättar begripligheten och gör utvidgningar lättare.
- **Tänk ut bra namn.** En viktig pusselbit för att skapa begriplig kod är att tänka ut bra namn. Detta kan vara förvånansvärt svårt och kan kräva mycket diskussioner och tankemöda. Om du inser att ett namn är illa valt är det förmodligen värt jobbet att omstrukturera koden och införa ett bättre namn, speciellt om andra ännu inte vant sig alltför mycket vid begreppet.
- **Kontrollera parametrar och variabler.** Ofta känner man till vilka villkor som måste gälla för olika variablers värden. Till exempel vet man ofta att en viss funktionsparameter av heltalstyp inte får vara negativ. Då kan man säkerställa detta genom att lägga in kontroller av att villkoret är uppfyllt. Vid villkor som gäller parametrar, brukar man i Scala anropa `require`, till exempel: `require(x >= 0, "x must be positive")`. Det finns också en metod `assert` som fungerar på samma vis<sup>5</sup>; medan `require` används för att kontrollera parametrar, brukar `assert` användas för att kontrollera generella villkor som ska gälla, till exempel `assert(x + y > n, "overflow")`. Fördelen med att lägga in kontroller av villkor är att villkorsbrott upptäcks direkt och felsökningen blir lättare.

<sup>5</sup>[stackoverflow.com/questions/26140757/what-to-choose-between-require-and-assert-in-scala](https://stackoverflow.com/questions/26140757/what-to-choose-between-require-and-assert-in-scala)

- **Kontrollera typer.** Med *typannoteringar* får du hjälp av kompilatorn att kontrollera dina hypoteser om vilka typer olika värden har. I Scala kan du nästan var du vill i ett uttryck lägga till ett kolon och en typ för att begära att kompilatorn kontrollerar typen. Till exempel kan du skriva `(xs + f(42)) : Set[Int]` för att säkerställa att uttrycket `xs + f(42)` verkligen ger en mängd med heltal. Även om du sällan i Scala behöver ange typer explicit, tack vare kompilatorns typinferens, bidrar det till läsbarheten och skapar säkrare kod om du på lämpliga ställen ändå anger de typer som du förväntar dig, speciellt vid i komplicerade uttryck eller långa kedjor av metodanrop, och när metoders returtyper inte är uppenbara. Dessutom kan kompilatorn ibland undvika att gå vilse i speciellt svåra typhärledningar, om du hjälper den på traven med explicita typannoteringar.
- **Hantera saknade värden.** Det är mycket vanligt att man måste hantera situationer där ett värde saknas, inte kan beräknas, eller inte finns tillgängligt av andra orsaker. Man kan hålla reda på att ett värde saknas genom att representera detta med speciella värden, t.ex. `-1` eller `null`. Men den strategin leder mycket lätt till buggar, då man lätt glömmer att på andra ställen i koden kontrollera dessa speciella värden. Med sådana speciella värden får man heller ingen hjälp av kompilatorn att upptäcka att man missat att ta hand om dem. Om man istället hanterar eventuellt saknade värden med `Option` (se kapitel 10), så får man hjälp vid kompileringstid och slipper exekveringsfel och besvärlig felsökning. Det blir dessutom väldigt tydligt för alla som läser din kod, inklusive du själv, att ett värde kan saknas.
- **Hantera undantag.** När undantag uppstår, t.ex. att en fil inte kan läsas eller det blir division med noll, avbryts exekveringen och programmets användare kan inte använda programmet längre, vilket i värsta fall kan få ödesdigra konsekvenser. Därför vill man hantera undantagssituationer på ett sådant sätt att programmet blir robust och inte kraschar. Detta kan man med fördel göra genom att kapsla in undantaget i ett värde av typen `Try`, se kapitel 10. I likhet med `Option` för saknade värden, blir det tydligt i koden att ett värde av typen `Try` kan innebära ett lyckat resultat (`Success`), eller så fallerar beräkningen (`Failure`) med en inkapslad, förhindrad krasch.
- **Granska kod.** Det är allmän praxis i industriell programvaruutveckling att göra kodgranskningar, vid vilka en grupp människor noga studerar någon annans kod och ger kommentarer och identifierar potentiella problem. Ofta har man en checklista att utgå ifrån medan man läser koden, som innehåller punkter man vill kontrollera speciellt, t.ex. begriplighet, namngivning, kontroller av parametrar, hantering av saknade värden och undantag, etc. Många organisationer har en överenskommen kodningsstandard med riktlinjer för kodens utseende och stil som alla ska följa om inte speciella skäl finns. Att sådana stilriktlinjer följs kan kontrolleras genom granskningar. Det finns också verktygsstöd för att göra sådana kontroller. Ett exempel på kodningsriktlinjer för Scala finns på den officiella dokumentationssajten<sup>6</sup>.
- **Testa kod.** Det är allmän praxis i industriell programvaruutveckling att genomföra tester på flera olika nivåer. Man kombinerar ofta **enhetstest** (eng. *unit test*) av enskilda delar av koden, med **funktionstest** (eng. *feature test*) för att se så att indata i en tänkt användningssituation ger önskat resultat, och **systemtest** (eng. *system test*) för att se att alla delar fungerar tillsammans under realistiska omständigheter.

---

<sup>6</sup><https://docs.scala-lang.org/style/>

- **Lär av användarnas upplevelser.** När koden sätts i produktion finns möjlighet att lära sig genom återkoppling från användare. Hur systemet används och hur användarna upplever det att använda systemet är mycket viktig information när man ska besluta om hur koden bäst ska utveckla vidare. Från användarna kan man få reda på både okända buggar och få briljanta idéer till nya värdefulla funktioner. En mjukvaruutvecklande organisations innovationsförmåga beror i stor utsträckning på dess förmåga att kontinuerligt leverera kod som får allt fler funktioner som användarna gillar, utan att för många irriterande eller ödesdigra buggar.

## D.3 Vad är debugging?

När en felyttring identifierats, t.ex. genom testning eller slutanvändare rapporterar om problem, vidtar sökandet efter den bakomliggande felorsaken, så att vi förstår *varför* det blev fel och sedan kan *åtgärda* misstaget. Denna process kallas **avlusning** (eng. *debugging*).

### D.3.1 Hur hittas felorsaken?

Första steget i avlusningsprocessen är att hitta den bakomliggande felorsaken. Detta kan vara mycket svårt, speciellt om systemet är stort och komplicerat.

När du stirrar dig blind på koden utan att hitta felorsaken, kan det bero på att du har en felaktig hypotes om vad koden egentligen gör. Du är övertygad om att en viss sak händer, men *egentligen* är det *inte* det du *tror* händer som *verkligen* händer. Exempelvis kanske du antar att en räknare räknas upp i en loop, men i själva verket saknas uppräknningen. Om du oflekterat accepterar ditt felaktiga antagande, är det stor risk att du letar på fel ställe i koden.

Följande åtgärder är ofta lämpliga när man jagar buggar:

- **Återskapa buggen med ett minimalt testfall.** När du upptäckt en felyttring är det viktigt att kunna återskapa felet, så att koden som körs precis *innan* buggen uppstår kan felsökas. Allra bäst är det om du kan skapa ett **minimalt testfall** där precis den minimala indata och de enskilda förutsättningar nedtecknas, som ska gälla för att buggen ska uppstå. Beskrivningen av det minimala testfallet är första pusselbiten i det detektivarbete som vidtar under felsökningen.
- **Formulera och verifiera hypoteser om buggen.** En grundläggande princip vid felsökning är att uttryckligen formulera hypoteser som du har om vad som sker i systemet medan buggen uppstår och sedan *verifiera* att de verkligen stämmer, genom olika undersökningar av det exekverande systemet. Du ska alltså tydligt beskriva hur du tror att koden fungerar och sedan med olika former av instrumentering, t.ex. genom utskrifter i terminalen av variablers värden, kontrollera att så verkligen är fallet. Detta kan göras med instrumentering enligt nedan.
- **Instrumentering med utskrifter, "print-debugging".**

För att verifiera din hypotes om vad som leder fram till buggen, behöver du kontrollera vad som händer. Det kan du göra genom att på väl valda ställen ligga in `println`-utskrifter i koden där värden på intressanta variabler skrivs ut. Det kan behövas lite klurighet för att hitta precis rätt utskrifter; om man skriver ut allt som händer i alla loopar drunknar man i all information, men skriver man ut för lite förbiser man kanske den falsifierade hypotesen och får ingen hjälp att knäcka bugg-gåtan.



Du kan även använda en avlusare (eng. *debugger*), som normalt ingår i en integrerad utvecklingsmiljö, för att instrumentera din kod. Se vidare i avsnitt D.5 om hur du använder avlusarna i Eclipse och IntelliJ IDEA.

## D.4 Åtgärda fel

Ofta är det det svåraste att *hitta* buggen, medan själva buggrättningen visar sig trivial. Har du, till exempel, väl hittat den saknade uppräkningsvariabeln av din loop-variabel är det uppenbart vad du ska göra.

Men ibland är det riktigt knepigt att åtgärda felet. Nedan sammanfattas några av de situationer som kan uppkomma, som gör att felrättningen blir extra svår.

- Kanske är själva algoritmen i grunden feltänkt och en helt ny algoritm behöver konstrueras. Att skapa nya algoritmer från grunden kan visa sig mycket svårt i en del fall. I fortsättningskurser får du lära dig mer om algoritmkonstruktionens ädla konst.
- Kanske algoritmen fungerar för olika normalfall, medan ovanliga undantagsfall inte hanteras korrekt. Att på ett bra sätt hantera alla upptänkliga fall kan visa sig väldigt knepigt. Tyvärr är det ofta undantagsfall i kombination med buggar som öppnar för säkerhetsluckor redo att utnyttjas av elaka hackare för att krascha systemet eller smitta ner det med virus.
- Kanske är problemet i sig väldigt svårt att lösa på ett korrekt sätt. Algoritmen kan vara riktigt knepig med många villkor, loopar och nästlade datastrukturer. Blir det fel i en sådan algoritm kan det ta lång tid att få ändringar att fungera och alla villkor, loopar och nästlade datastrukturer att passa ihop igen efter felrättningen.
- Medan man rättar en bug kan man råka att, av misstag, skapa nya buggar. Risken för detta är speciellt stor om koden är komplex. Ibland låter man till och med bli att åtgärda ett fel om systemet ändå fungerar hjälpligt i andra avseenden och risken är för stor att nya buggar skapas. Då behöver systemet strukturerats om så att det blir lättare att ändra i.
- Kanske växer exekveringstiden exponentiellt med datamängden. Det kan då i praktiken vara omöjligt att skriva ett program som i alla lägen blir färdigt inom rimlig tid. Då får man försöka tänka ut kluriga genvägar till suboptimala lösningar som ändå duger, vilket ibland kräver mycket avancerad programmeringsteknik.

Det finns ingen allena rådande snabbfix att ta till när man stöter på svåra fel. Att bli en produktiv systemutvecklare, som framgångsrikt reder ut allehanda buggar, handlar i stor utsträckning om att kombinera en bred allmänbildning inom datavetenskap med ett livslångt lärande, där varje bugg du hittar och åtgärdar ger dig nya kunskaper och erfarenheter inför framtiden. Även om din bugg är irriterande, försök se den som en ny chans till ökad lärdom!

## D.5 Använda en debugger

Med en professionell integrerad utvecklingsmiljö kommer ofta en avancerad debugger, som kan hjälpa dig att följa exekveringen och se vad som händer medan koden kör. Normalt ingår dessa funktioner i en debugger:

- **Sätta brytpunkter.** Med hjälp av debuggern kan du sätta så kallade *brytpunkter* på speciella ställen i koden. Detta görs ofta genom att du markerar en kodrad i marginalen varpå en brytpunktssymbol placeras där. När exekveringen når brytpunkten avbryts exekveringen och du kan stega dig vidare eller inspektera variablers värden vid brytpunkten.
- **Stegad exekvering.** När du nått en brytpunkt kan du med hjälp av debuggern stega dig fram genom koden rad för rad och se vad som händer. Om du kommer till ett funktionsanrop kan du välja att stega in i koden som implementerar funktionen eller bara köra funktionen i ett svep och stega till nästa rad som kommer efter funktionsanropet. Det kan kräva många omkörningar från en viss brytpunkt, innan man hittar vilka funktioner som inte verkar relevanta alls för buggen och bara kan stegas över, eller vilka funktioner som utgör gåtans lösning och som du vill stega in i och undersöka närmare. Stegar man djupt ner i funktionsanropskedjan, går man lätt vilse och får börja om. (Det går inte att stega bakåt...)
- **Inspektera variabler.** Medan du stegar dig fram i koden kan du inspektera variablers värden. I ett område på skärmen presenterar debuggern både enkla värden så som heltal och strängar, men även datastrukturer, så som vektorer och listor, kan inspekteras genom att debuggern låter dig bläddra bland arrayer och objektreferenser. Ett program kan ha väldigt många variabler och djupa strukturer av objekt som refererar till nya objekt. Det är ofta ett knepigt detektivarbete att försöka lista ut hur olika variabelvärden relaterar till orsaken bakom buggen som du letar efter. Du behöver ofta växla mellan att läsa koden, stega dig fram, sätta nya brytpunkter och inspektera variabler för att förstå vad som händer.

I Kojo Desktop (se appendix [A](#)) finns lättanvända debug-funktioner. Man kan till exempel följa stegen i exekveringen med hjälp av den brandgula play-knappen "Kör och spåra programmet" (kortkommando: Alt+Enter). Då öppnas ett nytt fönster som visar exekveringsstegen. Man kan klicka på ett steg och få information om parametrar vid funktionsanrop etc.

Du kan läsa mer om hur man använder en avancerad debugger i en professionell integrerad utvecklingsmiljö i appendix [H](#).



## Appendix E

# Dokumentation

Dokumentation hjälper andra att använda din kod, men underlättar även för dig själv när du vid ett senare tillfälle ska erinra dig hur den fungerar och hur du ska använda och bygga vidare på din kod. Modern systemutveckling baseras ofta på öppen källkod och färdiga api (eng. *application programming interface*), där kvaliteten på dokumentationen är avgörande för hur lätt det är att komma igång med att använda koden.

Nedan listas exempel på olika typer av dokumentation<sup>1</sup>:

- **Kravdokumentation** beskriver det övergripande målet med mjukvaran, samt funktionella krav och kvalitetskrav som uppfylls av systemet.
- **Designdokumentation** beskriver arkitekturen, hur koden är organiserad i moduler, och den interna systemstrukturen t.ex. i form av klasser, objekt och deras relation.
- **Slutanvändardokumentation** kan t.ex. vara manualer för användning av systemet och installationsanvisningar.
- **Teknisk dokumentation** kan t.ex. vara api-dokumentation som beskriver vilka funktioner som ingår i ett programbibliotek. Sådan dokumentation genereras ofta med hjälp av ett **dokumentationsverktyg** (se avsnitt E.1). Andra typer av teknisk dokumentation är instruktioner om hur man bygger koden med eventuellt tillhörande byggverktygskonfigurationsfiler; ofta beskrivs byggförfarandet steg för steg i en textfil med namnet README. (Läs mer om byggverktyg i appendix F.)

Det är en stor utmaning att hålla dokumentationen uppdaterad allteftersom koden utvecklas. Även om man får hjälp att generera en navigerbar sajt av ett dokumentationsverktyg, måste själva *innehållet* i de manuellt författade dokumentationskommentarerna vara i överensstämmelse med den aktuella versionen av koden. Uppdateras koden, måste man alltså vara noga med att uppdatera dokumentationskommentarerna, annars uppstår stor förvirring.

Detta problem är så pass allvarligt att man ska tänka sig noga för hur man kan formulera dokumentationskommentarerna på ett framtidssäkert sätt, och hur omfattande de ska vara i förhållande till den framtida arbetsinsatsen med att hålla dem uppdaterade. Desto mer omfattande kommentarer desto mer jobb att hålla dem uppdaterade.

Det är i praktiken svårt att uppnå en optimal balans mellan bra och många kommentarer som *hjälper* användaren, och å andra sidan svårunderhållna och föråldrade kommentarer som *stjälper* användare.

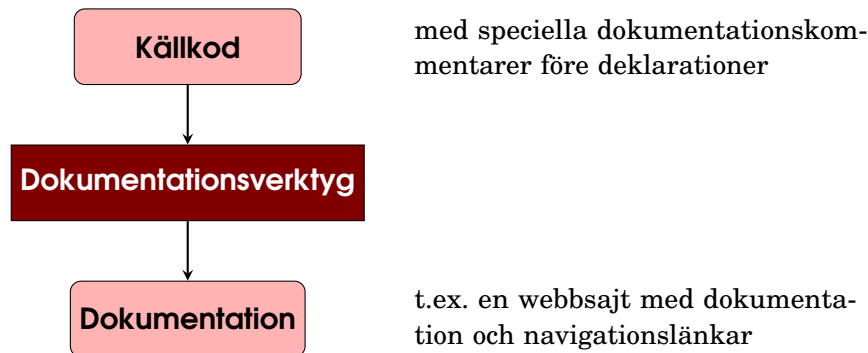
---

<sup>1</sup>[en.wikipedia.org/wiki/Software\\_documentation](https://en.wikipedia.org/wiki/Software_documentation)

## E.1 Vad gör ett dokumentationsverktyg?

Ett dokumentationsverktyg genererar teknisk dokumentation av koden baserat på speciella **dokumentationskommentarer** som skrivs i koden omedelbart före deklARATIONER av det som ska dokumenteras. Dessa dokumentationskommentarer skrivs enligt en speciell syntax som dokumentationsverktyget kan tolka.

Utdata från ett dokumentationsverktyg utgörs typiskt av en webbsajt med ändamålsenlig formatering och navigationslänkar, se figur E.1.



**Figur E.1:** Ett dokumentationsverktyg läser koden och dokumentationskommentarer och genererar dokumentation, t.ex. i form av en webbsajt.

## E.2 scaladoc

Med Scala-installationen följer dokumentationsverktyget `scaladoc`, som genererar en webbsajt med ändamålsenlig layout och specialfunktioner för att söka, filtrera och navigera i dokumentationen.

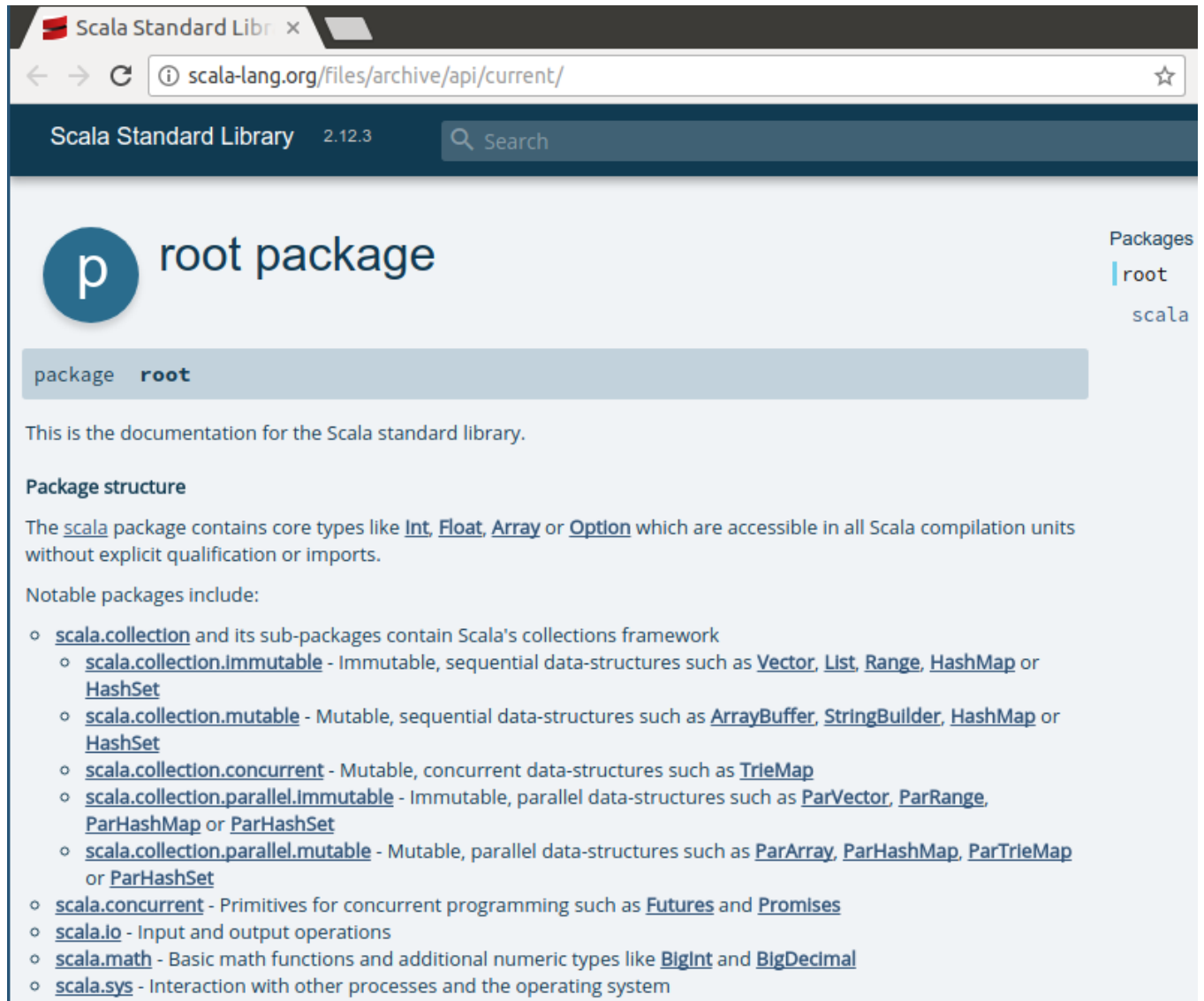
Dokumentationen av stora bibliotek kan bli omfattande och det krävs träning i att använda dokumentationssajter för att få maximal nytta av dem. I efterföljande avsnitt beskrivs först hur du använder dokumentation som är genererad med `scaladoc`. Därefter visas hur du själv kan generera dokumentation för din egen kod.

### E.2.1 Använda dokumentation från `scaladoc`

Dokumentationen av Scalas standardbibliotek är genererad med `scaladoc` och att navigera i denna ger bra träning i hur man använder avancerad api-dokumentation. Du hittar dokumentationen för Scalas standardbibliotek här:

<http://scala-lang.org/api/current>

När du surfar dit möts du av dokumentationen för *root package*, som ger en översikt av olika paket i standardbiblioteket. I sökrutan uppe till vänster kan du skriva början på namnet på klasser, traits, eller objekt som du letar efter, så som visas i figure E.2.



The screenshot shows a web browser window with the URL `scala-lang.org/files/archive/api/current/`. The page title is "Scala Standard Library 2.12.3". The main heading is "root package" with a large blue circle containing a white 'p'. A sidebar on the right lists "Packages" with "root" and "scala". Below the heading, a light blue bar contains the text "package root". The main content area states: "This is the documentation for the Scala standard library." followed by a section "Package structure". It explains that the `scala` package contains core types like `Int`, `Float`, `Array` or `Option` accessible in all Scala compilation units without explicit qualification or imports. It then lists notable packages including `scala.collection` and its sub-packages, `scala.concurrent`, `scala.io`, `scala.math`, and `scala.sys`.

Scala Standard Library 2.12.3 Search

# p root package

Packages  
| root  
scala

package root

This is the documentation for the Scala standard library.

## Package structure

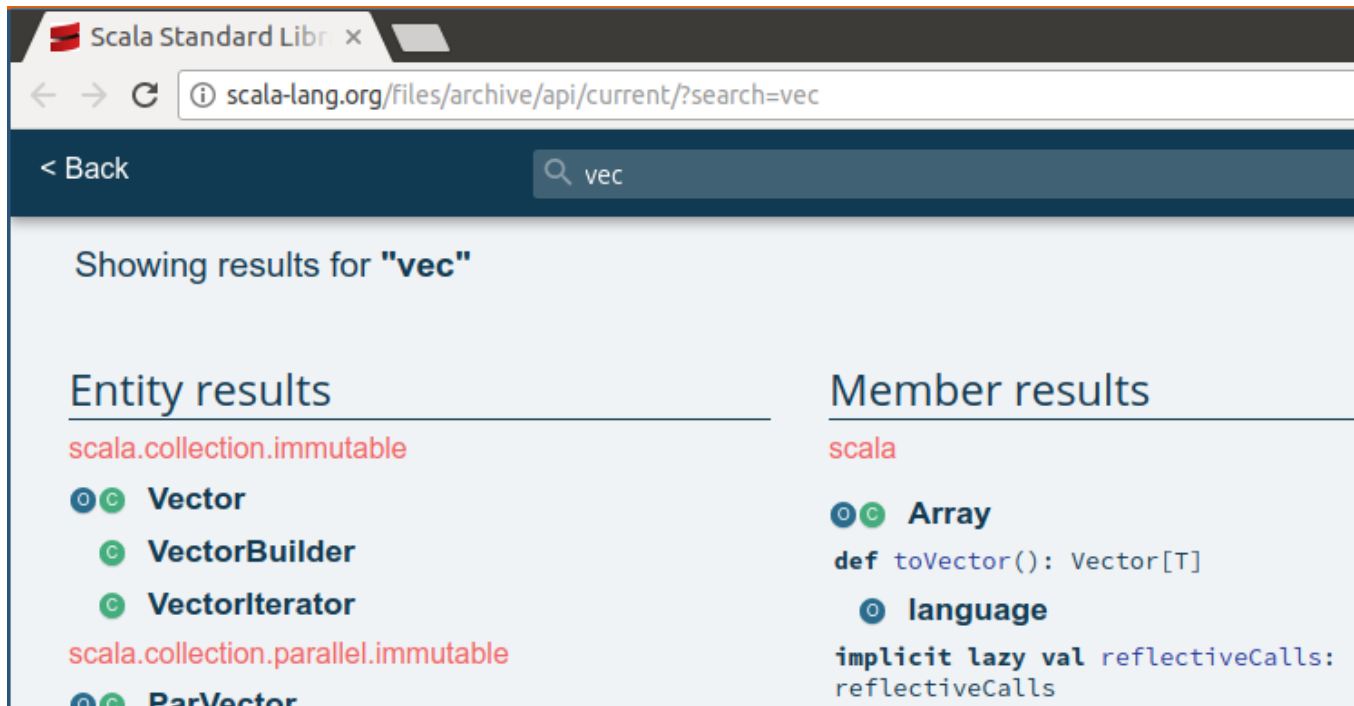
The `scala` package contains core types like `Int`, `Float`, `Array` or `Option` which are accessible in all Scala compilation units without explicit qualification or imports.

Notable packages include:

- `scala.collection` and its sub-packages contain Scala's collections framework
  - `scala.collection.immutable` - Immutable, sequential data-structures such as `Vector`, `List`, `Range`, `HashMap` or `HashSet`
  - `scala.collection.mutable` - Mutable, sequential data-structures such as `ArrayBuffer`, `StringBuilder`, `HashMap` or `HashSet`
  - `scala.collection.concurrent` - Mutable, concurrent data-structures such as `TrieMap`
  - `scala.collection.parallel.immutable` - Immutable, parallel data-structures such as `ParVector`, `ParRange`, `ParHashMap` or `ParHashSet`
  - `scala.collection.parallel.mutable` - Mutable, parallel data-structures such as `ParArray`, `ParHashMap`, `ParTrieMap` or `ParHashSet`
- `scala.concurrent` - Primitives for concurrent programming such as `Futures` and `Promises`
- `scala.io` - Input and output operations
- `scala.math` - Basic math functions and additional numeric types like `BigInt` and `BigDecimal`
- `scala.sys` - Interaction with other processes and the operating system

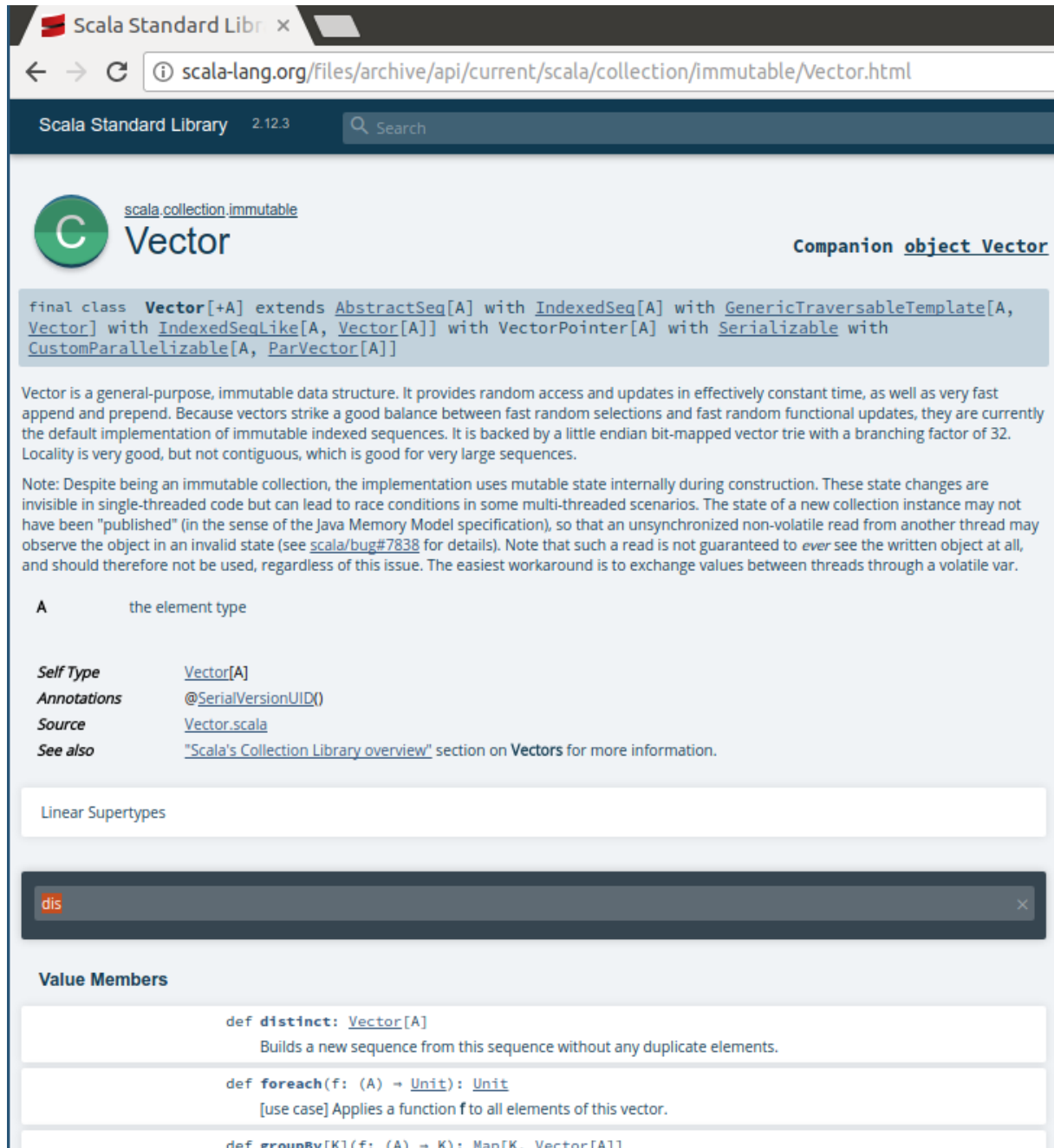
**Figur E.2:** Förstasidan med dokumentationen av standardbiblioteket i Scala.

Genom att skriva text i sökrutan får du en filtrerad lista på allt som har ett namn som börjar på det söker efter. I figur E.3 visas en sökning efter `vec`.



**Figur E.3:** Sökning efter innehåll som börjar på vec.

Om du klickar på Vector får du se dokumentationen i figur E.4



The screenshot shows the Scala Standard Library documentation for the `Vector` class. The browser address bar shows the URL `scala-lang.org/files/archive/api/current/scala/collection/immutable/Vector.html`. The page title is "Scala Standard Library 2.12.3" with a search bar. The main heading is "Vector" with a green circular icon containing a 'C' and the text "scala.collection.immutable". To the right, it says "Companion object Vector".

The class signature is: `final class Vector[+A] extends AbstractSeq[A] with IndexedSeq[A] with GenericTraversableTemplate[A, Vector] with IndexedSeqLike[A, Vector[A]] with VectorPointer[A] with Serializable with CustomParallelizable[A, ParVector[A]]`

A paragraph describes `Vector` as a general-purpose, immutable data structure that provides random access and updates in effectively constant time, as well as very fast append and prepend. It is the default implementation of immutable indexed sequences, backed by a little endian bit-mapped vector trie with a branching factor of 32. Locality is very good, but not contiguous, which is good for very large sequences.

A note states: "Despite being an immutable collection, the implementation uses mutable state internally during construction. These state changes are invisible in single-threaded code but can lead to race conditions in some multi-threaded scenarios. The state of a new collection instance may not have been 'published' (in the sense of the Java Memory Model specification), so that an unsynchronized non-volatile read from another thread may observe the object in an invalid state (see [scala/bug#7838](#) for details). Note that such a read is not guaranteed to ever see the written object at all, and should therefore not be used, regardless of this issue. The easiest workaround is to exchange values between threads through a volatile var."

Below the note, it says "A" is "the element type".

Metadata includes:
 

- Self Type: `Vector[A]`
- Annotations: `@SerialVersionUID()`
- Source: `Vector.scala`
- See also: "Scala's Collection Library overview" section on `Vectors` for more information.

There is a section for "Linear Supertypes" which is currently empty.

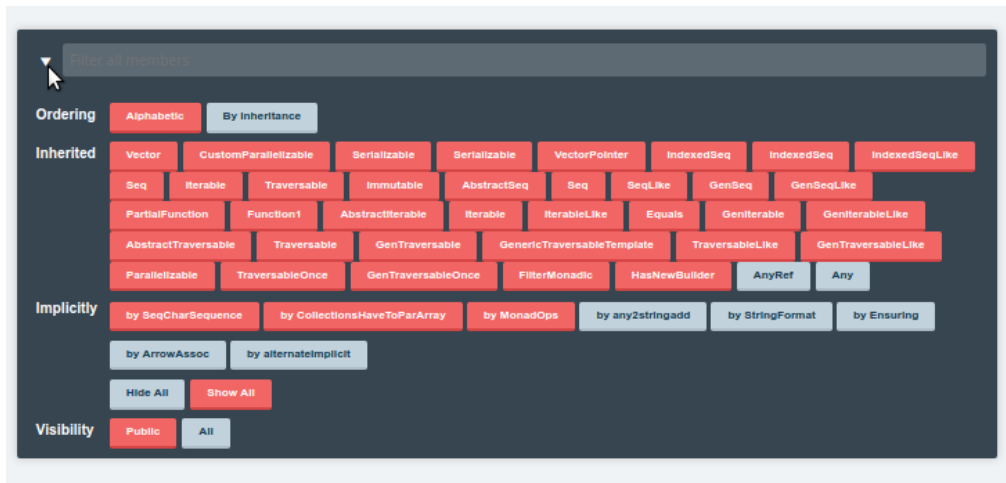
A dark grey search bar with the text "dis" is visible.

The "Value Members" section lists:
 

- `def distinct: Vector[A]`: Builds a new sequence from this sequence without any duplicate elements.
- `def foreach(f: (A) => Unit): Unit`: [use case] Applies a function `f` to all elements of this vector.
- `def groupBy[K](f: (A) => K): Map[K, Vector[A]]`

**Figur E.4:** En del av dokumentationen av klassen `Vector`

Genom att skriva text i den nedre, mörkgrå sökrutan kan du filtrera listan med klass-medlemmar. Om klickar på länken **object** `Vector`, eller på den runda, gröna ikonen med ett stort C, får du se kompanjonsobjektets medlemmer. Du kan få fram en filtreringsruta med fler möjligheter genom att klicka på expanderingspilen i den mörkgrå sökrutan så som visas i figur E.5.



**Figur E.5:** Expanderad sökruta med extra filtreringsmöjligheter.

## E.2.2 Skriva dokumentationskommentarer

Verktöget scaladoc läser kommentarer som börjar med `/**` och slutar med `*/` och associeras till efterföljande deklaration. Notera de dubbla asteriskerna. Alla rader som följer efter `/**` ska, enligt konventionen för Scalas dokumentationskommentarer, börja med en asterisk `*` med indrag med flera blanksteg så att den hamnar under *andra* asterisken i öppningskommentaren, som nedan:

```
/** Först kommer en sammanfattning på en enda rad.
 *
 * Sedan kommer eventuellt en mer detaljerad beskrivning,
 * som kan vara flera rader lång.
 */
```

Dokumentationskommentaren slutar med `*/` rakt under asterisk-kolumnen.

- Med `@constructor` i början på en rad ges en speciell kommentar om konstruktörer.
- Med `@param` i början på en rad ges en speciell kommentar angående parametrar.
- Med `@return` i början av en rad ges en speciell kommentar angående vad som returneras vid metदानrop.
- Länkar skrivs inom dubbla hakparenteser, enligt exempel nedan.

```
/** A person who uses our application.
 *
 * @constructor create a new person with a name and age.
 * @param name the person's name
 * @param age the person's age in years
 */
class Person(name: String, age: Int)

/** Factory for [[mypackage.Person]] instances. */
object Person:
  /** Creates a person with a given name and age.
   *
   * @param name their name
   * @param age the age of the person to create
   */
  def apply(name: String, age: Int) = ???

  /** Creates a person with a given name and birthdate
   *
   * @param name their name
   * @param birthDate the person's birthdate
   * @return a new Person instance with the age determined by the
   *         birthdate and current date.
   */
  def apply(name: String, birthDate: java.util.Date) = ???
```

Läs mer här om hur du skriver dokumentationskommentarer:

<https://docs.scala-lang.org/scala3/guides/scaladoc/>

Läs mer om dokumentation här:

<https://docs.scala-lang.org/scala3/scaladoc.html>

<https://scala-cli.virtuslab.org/docs/commands/doc>

### E.2.3 Generera dokumentation

Du genererar dokumentation enklast med hjälp av körverktyget `scala-cli`. I terminalen skriv

```
scala-cli doc . -o api
```

När `scala-cli` är färdig med att generera dokumentationen så meddelas vilken katalog som dokumentationen ligger i. Du läser dokumentationen genom att öppna filen `index.html` i en webbläsare.

Mer att läsa om att generera dokumentation finns här:

<https://scala-cli.virtuslab.org/docs/commands/doc>



## Appendix F

# Byggverktyg

### F.1 Vad gör ett byggverktyg?

Ett **byggverktyg** (eng. *build tool*) används för att

- ladda ner,
- kompilera,
- testköra,
- paketera och
- distribuera

programvara. Ett stort utvecklingsprojekt kan innehålla många hundra kodfiler och under utvecklingens gång vill man kontinuerligt testköra systemet för att kontrollera att allt fortfarande fungerar; även den kod som inte ändrats, men som kanske ändå påverkas av ändringen. Ett byggverktyg används för att *automatisera* denna process.

Ett viktigt begrepp i byggsammanhang är **beroende** (eng. *dependency*). Om koden X behöver annan kod Y för att fungera, sägs kod X ha ett beroende till kod Y.

I konfigurationsfiler, som är skrivna i ett format som byggverktyget kan läsa, specificeras de beroenden som finns mellan olika koddelar. Byggverktyget analyserar dessa beroenden och, baserat på ändringstidsmarkeringar för kodfilerna, avgör byggverktyget vilken delmängd av kodfilerna som behöver **omkompileras** efter en ändring. Detta snabbar upp kompileringen avsevärt jämfört med en total omkompilering från grunden, som för ett stort projekt kan ta många minuter eller till och med timmar. Efter omkompilering av det som ändrats, kan byggverktyget instrueras att köra igenom testprogram och rapportera om testernas utfall, men även ladda upp körbara programpaket till t.ex. en webbserver.

En vanlig typ av beroende är färdiga programbibliotek som utnyttjas av systemet under utveckling, vilket i praktiken ofta innebär att en sökväg till den kompilerade koden för programbiblioteket behöver göras tillgänglig. I JVM-sammanhang innebär detta att sökvägen till alla nödvändiga jar-filer behöver finnas på sökvägslistan kallad **classpath**.

Många byggverktyg kan utföra så kallad **beroendeupplösning** (eng. *dependency resolution*), vilket innebär att nätverket av beroenden analyseras och rätt uppsättning programpaket görs tillgänglig under bygget. Detta kan även innebära att programpaket som är tillgängliga via nätet automatiskt laddas ned inför bygget, t.ex. via lagringsplatser för öppen källkod.

Även om man bara har ett litet kodprojekt med några få kodfiler, är det ändå smidigt att använda ett byggverktyg. Man kan nämligen göra så att byggverktyget är aktivt i

bakgrunden och, så fort man sparar en ändring av koden, gör omkompilering och rapporterar eventuella kompileringsfel.

Det är klokt att kompilera om ofta, helst vid varje liten ändring, och rätta eventuella fel *innan* nya ändringar görs, eftersom det är mycket lättare att klura ut ett enskilt problem efter en mindre ändring, än att åtgärda en massa svåra följdfe, som beror på en sekvens av omfattande ändringar, där misstaget begicks någon gång långt tidigare.

En integrerad utvecklingsmiljö, så som VS Code eller IntelliJ IDEA, bygger om koden kontinuerligt och kan ofta konfigureras att kommunicera med flera olika byggverktyg. Exempelvis kan du med VS Code välja om du vill att Scala CLI eller sbt ska användas för att bygga ditt projekt.

Det finns många olika byggverktyg. Några allmänt kända byggverktyg listas nedan så att du ska känna igen vilket byggverktyg som används i öppen-källkods-projekt som du stöter på, t.ex. på GitHub.

- **Scala CLI.** Verktöget Scala CLI (Command Line Interface) är öppen källkod utvecklad av VirtusLab<sup>1</sup> för att kompilera och köra Scala- och Java-program och innehåller också grundläggande byggverktygsfunktioner, så som att köra testfall, paketera jar-filer och skapa dokumentation. Kommandot `scala-cli` övertog år 2023 rollen som det officiella `scala`-kommandot. Detta är det enklaste och rekommenderade sättet att bygga system med Scala-kod. Grundläggande användning av Scala CLI beskrivs i Appendix C.4.3, medan en mer utförlig beskrivning återfinns nedan i avsnitt F.2.

- **sbt.** Även kallad *Scala Build Tool*. Används för att bygga Java- och Scala-program i samexistens, men även för att automatisera en mängd andra saker. sbt är utvecklat i Scala och konfigurationsfilerna, som heter `build.sbt`, innehåller Scala-kod som styr byggprocessen. sbt är avancerat och klarar bygga system som består av många projekt (eng. *multi-project build*). sbt är det i särklass vanligaste byggverktyget för Scala och många öppen-källkodsprojekt använder sbt. Läs mer om sbt i avsnitt F.3 nedan.

Efter kritik om att sbt är komplicerat så har flera alternativa byggverktyg för Scala utvecklats, däribland `bleep`<sup>2</sup> och `mill`<sup>3</sup>.

- **Apache Maven,** `mvn` är också skriven i Java och är en efterföljare till ant. Maven används av många Java-utvecklare. Konfigurationsfilerna heter `pom.xml` och innehåller en s.k. projektobjektmodell specificerad i XML enligt speciella regler.
- **gradle** bygger vidare på idéerna från ant och maven och är skrivet i Java och Groovy. Konfigurationsfilerna skrivs i Groovy och heter `build.gradle`.
- **Apache ant.** Detta byggverktyg är utvecklat i Java som ett alternativ till make och används fortfarande i många Java-projekt, även om Maven och Gradle är vanligare numera. Konfigurationsfilerna heter `build.xml` och skrivs i det standardiserade språket XML enligt speciella regler.
- **make.** Detta anrika byggverktyg har varit med ända sedan 1970-talet och används fortfarande för att bygga många system under Linux, och är populärt vid utveckling med programspråken C och C++. En konfigurationsfil för make heter `Makefile` och har en egen, speciell syntax.

---

<sup>1</sup><https://scala-cli.virtuslab.org/>

<sup>2</sup><https://bleep.build/docs/>

<sup>3</sup>[https://mill-build.com/mill/Intro\\_to\\_Mill.html](https://mill-build.com/mill/Intro_to_Mill.html)

## F.2 Scala Command Line Interface *scala-cli*

Utvecklingen av [Scala CLI](#) påbörjades 2022 av [Virtuslab](#). Scala CLI blir under 2023 det officiella byggverktyget för enkla Scala-projekt och kommer medfölja [installationen av Scala](#). Scala CLI kan även installeras separat och köras med kommandot `scala-cli`, men blir någon gång under 2023 liktydigt med kommandot `scala` i terminalen.<sup>4</sup>

Efter nyinstallation av Scala CLI kan du ange följande kommando för att, en gång för alla, få tillgång till kompletteringar av optioner med Tab-tangenten i terminalen:

```
scala-cli install-completions
```

Innan du börjar skriva källkod i en ny katalog i VS Code kan du konfigurera VS Code att vara redo för att använda Scala CLI som byggverktyg i aktuell katalog med följande kommando (vänta med att öppna VS Code till efter att du kört kommandot):

```
mkdir minNyaKatalog  
cd minNyaKatalog  
scala-cli setup-ide .  
code .
```

### F.2.1 Grundläggande byggfunktioner i Scala CLI

De grundläggande funktionerna sammanfattas nedan (se även Appendix [C.4.3](#)):

---

<sup>4</sup>I skrivande stund har gamla `scala`-kommandot ännu inte uppgaderats, men det förväntas ske i början av hösten. När så väl sker kan alla förekomster av `scala-cli` i detta stycke ersättas med det kortare kommandot `scala`, efter uppdatering.

<code>scala-cli repl</code>	Starta Scala REPL. Det går även bra med enbart <code>scala-cli</code>
<code>scala-cli repl hello.scala</code>	Starta Scala REPL med kompillerade koden i <code>hello.scala</code> på classpath.
<code>scala-cli repl .</code>	Starta repl med kodfiler i aktuell katalog tillgängliga på classpath.
<code>scala-cli compile hello.scala</code>	Kompilera koden i filen <code>hello.scala</code>
<code>scala-cli compile .</code>	Kompilera alla kodfiler i aktuell katalog.
<code>scala-cli run hello.scala</code>	Kompilera koden i filen <code>hello.scala</code> och kör igång eventuellt huvudprogram om kompileringen gick bra.
<code>scala-cli run .</code>	Kompilera och kör alla kodfiler i aktuell katalog.
<code>scala-cli run . --list-main-class</code>	Lista alla huvudprogram.
<code>scala-cli run . -M mypkg.myMain</code>	Kör ett specifikt huvudprogram. Förk. <code>-M</code> kan även skrivas <code>--main-class</code>
<code>scala-cli package .</code>	Paketera all kompillerade kodfiler i en körbar fil.

## F.2.2 Använda optioner för att styra Scala CLI

Det finns en mängd olika optioner som du kan lägga till för att styra vad Scala CLI ska göra. Se exempel nedan och förklaringar i efterföljande tabell:

```
scala-cli run . -S 3.3 -0 -unchecked --dep se.lth.cs::introprog::1.3.1 -w
```

Här förklaras några vanliga optioner som kan användas vid både kompilering och exekvering:

<code>--scala 3.3</code>	Använd version 3.3 av Scala. Optionen <code>--scala</code> kan förkortas med <code>-S</code>
<code>--watch</code>	Upprepa kommando vid sparad ändring. Optionen <code>--watch</code> förkortas med <code>-w</code>
<code>--jar introprog.jar</code>	Lägg till en jar-fil på classpath.
<code>--dep se.lth.cs::introprog::1.3.1</code>	Lägg till ett beroende på classpath.
<code>--scalac-option -unchecked</code>	Lägg till en kompilator-option som ger extra varningar vid osäker kod. Optionen <code>--scalac-option</code> kan förkortas med <code>-0</code>

Fördelen med att explicit ange en viss Scala-version är att byggprocessen blir *upprepningsbar* även på en annan dator som kanske råkar ha en annan Scala-version installerad. Det går att ”spika fast” Scala-versionen till en ännu mer precis version, t.ex. 3.2.2. Om inte den versionen av kompilatorn finns installerad på datorn så kommer Scala CLI att automatiskt ladda ner och använda den explicit efterfrågade versionen under byggandet.

Det går också att be om den absolut mest rykande färska kompilatorversionen om man vill använda det allra senaste i Scala-språkets utveckling med 3.nightly. Speciellt kräver s.k. experimentella funktioner att du använder nightly-versionen<sup>5</sup>.

### F.2.3 Generera dokumentation med Scala CLI

Scala CLI kan också skapa dokumentation baserat på dokumentationskommentarer (se vidare Appendix E), enligt nedan. Med optionen `--output` kan du ange destinationskatalog och med `--force` så skrivs ev. gammal dokumentation över.

```
scala-cli doc . --output apidoc --force
```

### F.2.4 Paketering av exekverbar fil med Scala CLI

Scala CLI kan paketera din kod i en exekverbar fil så här:

```
scala-cli package . --force --standalone --output myapp
```

Här förklaras några vanliga optioner som kan användas vid paketering:

<code>--output</code>	Ange namn på utfilen med paketerad kod. Optionen <code>--output</code> kan förkortas med <code>-o</code> .
<code>--force</code>	Skriv över utfilen om den redan finns. Optionen <code>--force</code> kan förkortas med <code>-f</code> .
<code>--standalone</code>	Skapa en självständig, exekverbar jar-fil med din kod och dess beroenden.
<code>--library</code>	Skapa en jar-fil med din kod för användning av andra program.
<code>--assembly</code>	Skapa en fet jar-fil med din kod och alla dess beroenden för användning av andra program.

### F.2.5 Optioner som användningsdirektiv i ”magiska” kommentarer

I stället för att använda optioner i terminalen så kan du ge dessa som s.k. användningsdirektiv (eng. *using directives*) i ”magiska” kommentarer som börjar med `//>` `using` i början av valfri kod-fil.

Om du har flera kodfiler i samma katalog brukar man skapa en speciell fil som vanligtvis kallas `project.scala` och i den samla alla användningsdirektiv som styr byggandet. Här visas ett exempel hur det kan se ut:

```
//> using scala 3.3
//> using option -unchecked -deprecation
//> using option -Wunused:all -Wvalue-discard -Ysafe-init
```

<sup>5</sup><https://stackoverflow.com/questions/40622878/how-do-i-use-a-nightly-build-of-scala>

```
//> using dep se.lth.cs::introprog::1.3.1
```

De kompilatoroptioner som föreslås för att få extra varningar ovan har följande betydelser:

- unchecked           Extra varningar vid flera fall av osäker kod.
- deprecation       Förklaring vid användning av utgående funktioner.
- Wunused:all       Varning om deklarationer ej används.
- Wvalue-discard    Varning vid förlorat värde.
- Ysafe-init        Varna vid risk för ej initialiserade värden.

Du hittar mer information om Scala CLI här: <https://scala-cli.virtuslab.org/>

## F.3 Scala Build Tool sbt

Byggverktyget sbt är skrivet i Scala och är det mest använda byggverktyget bland Scala-utvecklare. Med sbt kan du skriva byggkonfigurationsfiler i Scala och även styra byggprocessen via ett interaktivt kommandoskal i terminalfönstret. Med inkrementell (stegvis) kompilering och parallellkörning av byggprocessens olika delar, kan den snabbas upp avsevärt.

### F.3.1 Installera sbt

sbt finns förinstallerat på LTH:s datorer och körs igång med kommandot sbt i terminalen.

Om du vill installera sbt på din egen dator, säkerställ först att du har java på din dator med terminalkommandot `java -version`. Om java saknas, följ instruktionerna i avsnitt C.3.2 på sidan 466. Följ sedan instruktionerna här för att installera sbt: <http://www.scala-sbt.org/download.html>

- **Linux.** Om du surfar till ovan sida från en Linux-dator syns några terminalkommando som du använder för att installera sbt i terminalen.
- **Windows.** Om du surfar till ovan sida från en Windows-dator visas en länk till en .msi-fil. Ladda ner och dubbelklicka på den. Innan du kör igång med sbt i en Windowsterminal är det bra att skriva `chcp 65001` för att särskilda tecken (t.ex. ÅÄÖ) ska fungera som de ska.
- **macOS.** Följ instruktionerna under rubriken *Manual Installation*.

När du kör sbt första gången kommer ytterligare filer att laddas ner och installeras och delar av denna process kan ta lång tid. Ha tålamod och avbryt inte körningen, även om inget speciellt ser ut att hända på ett bra tag.

### F.3.2 Använda sbt

sbt är konstruerat för att klara mycket stora projekt, men det är enkelt att använda sbt även om du bara har ett litet projekt med någon enstaka kodfil. Med sbt installerat, är det bara att köra igång sbtoch skriva run enligt nedan

```
> sbt
sbt> run
```

i terminalen i den katalog där dina kodfiler ligger. sbt letar då upp och kompilerar alla de .scala-filer som ligger i katalogen och, om det bara finns ett objekt med main-metod, kör sbt igång denna main-metod direkt, förutsatt att kompileringen kan avlutas utan fel. Även .java-filer kompileras automatiskt om de ligger i samma katalog.

Om du enbart skriver sbt körs det interaktiva kommandoskalet igång, där du kan köra kommando så som compile och run. Om du skriver ett ~ före kommandot run, enligt nedan kommer sbt vara aktivt i bakgrunden medan du editerar och så fort du sparar en ändring kommer omkompilering av ändrade kodfiler ske, varefter main-metoden exekveras om kompileringen lyckades.

```
> sbt
[info] Set current project to hello (in build file:/home/bjornr/hello/)
> ~run
[info] Running hello
Hello, World!
[success] Total time: 0 s, completed Aug 9, 2016 9:50:16 PM
1. Waiting for source changes... (press enter to interrupt)
[info] Compiling 1 Scala source to /home/bjornr/hello/target/scala-2.10/classes...
[info] Running hello
Hello again, World!
[success] Total time: 1 s, completed Aug 9, 2016 9:50:45 PM
2. Waiting for source changes... (press enter to interrupt)
```

I ovan körning gör sbt en omkompilering, efter att en ändring av utskriftssträngen sparats.

```
// in file hello.scala

@main def run = println("Hello again, World!") // add 'again'; Ctrl+S
```

## Katalogstruktur

Om man har kod i underkataloger förutsätter sbt att du följer en viss, specifik katalogstruktur. Denna katalogstruktur används även av andra byggverktyg, så som Maven, och fungerar även i många utvecklingsmiljöer så som Eclipse och IntelliJ.

Det blir också mindre rörigt och lättare för alla att hitta i projektets kataloger om dina kodfiler placeras i en given struktur som är allmänt accepterad. Placera därför gärna dina kodfiler i underkataloger enligt strukturen som visas i figur [F.1](#).

```

src/
  main/
    resources/
      <files to include in main jar here>
    scala/
      <main Scala sources>
    java/
      <main Java sources>
  test/
    resources
      <files to include in test jar here>
    scala/
      <test Scala sources>
    java/
      <test Java sources>

```

**Figur F.1:** Katalogstrukturen i ett sbt-projekt. Bara de kataloger som har något innehåll behöver finnas.

Lägg enligt denna struktur dina `.scala`-filer i underkatalogen `src/main/scala/` och dina `.java`-filer i underkatalogen `src/main/java/`. Om du lägger kod i någon av katalogerna `src/test/scala/` respektive `src/test/java/` kommer denna kod köras när du skriver sbt-kommandot `test`. Om du lägger filer i underkatalogen `src/main/resources/` kommer dessa att paketeras med i jar-filen som skapas när du kör sbt-kommandot `package`.

Om du använder t.ex. `package x.y.z`; i din Java-kod, måste även strukturer på underkataloger matcha och kodfilen alltså ligga i `src/main/java/x/y/z/`.

I Scala är det egentligen inte nödvändigt att koden ligger i samma katalog som de kompillerade `.class`-filerna, men det kan vara bra att följa paketstrukturen även för Scala-källkoden; speciellt om du senare vill kunna köra din kod med Eclipse, som kräver denna överensstämmelse mellan paket och källkodskataloger, inte bara för Java, utan även för Scala.

### Konfigurera dina byggen i filen `build.sbt`

Om du vill göra inställningar och även hjälpa andra att kunna återskapa dina byggen, så skapa en konfigurationsfil med namnet `build.sbt` och placera den i projektets baskatalog. Figur F.2 visar en byggkonfigurationsfil som specificerar vilken version av Scala-kompilatorn du använder, så att andra ska kunna bygga din kod under samma förutsättningar som du.

```
scalaVersion := "3.2.2"
```

**Figur F.2:** Exempel på konfigurationsfil för sbt. Filen ska ha namnet `build.sbt` och vara placerad i projektets baskatalog.

Här är ett exempel på en mer omfattande `build.sbt`:

```

scalaVersion    := "3.2.2"
scalacOptions   := Seq("-unchecked", "-deprecation") //mer info vid kompilering

fork            := true    // kör i en egen JVM, bra om ljud och grafik används
connectInput    := true    // koppla indata till rätt JVM vid fork
outputStrategy  := Some(StdoutOutput) // koppla utdata till rätt JVM vid fork

ThisBuild / useSuperShell := false // stänger av rör(l)ig progressinformation

```



Du kan läsa mer om alla möjligheter med sbt och hur man skapar mer avancerade byggkonfigurationsfiler här: <http://www.scala-sbt.org>

### Fixera versionen för sbt i project/build.properties

Om du skapar en katalog project (om den inte redan finns) kan du i en fil med namnet build.properties fixera versionen av sbt genom att låta filen ha detta innehåll (notera punkten och avsaknaden av citationstecken):

```
sbt.version=1.8.2
```

På så sätt riskerar du inga inkonsekvenser mellan en gammal build.sbt vid framtida uppdatering av sbt, ovan inställning garanterar att ditt bygge alltid kommer att byggas med denna version av sbt, och andra kan bygga din kod under samma förutsättningar som du.

### Lägga till kursbiblioteket introprog som ett beroende

Med följande text i build.sbt får du automatisk nedladdning och tillgång till kursens Scala-bibliotek introprog med bl.a. klassen PixelWindow för grafiska fönster:

```
scalaVersion := "3.2.2"
libraryDependencies += "se.lth.cs" %% "introprog" % "1.3.1"
```

Ändra ev. versionsnummer till senaste versionen. Notera de dubbla procent-tecknen före biblioteksnamnet, som används för Scala-bibliotek som kors-publicerats för olika versioner av Scala, t.ex. 3, 2.12 och 2.13, vilket gör att rätt biblioteksversion för rätt kompilatorversion laddas ned.

Du kan läsa mer om introprog här:

- Kod: <https://github.com/lunduniversity/introprog-scalalib>
- Dokumentation: <http://cs.lth.se/pgk/api>

### Lägga till andra beroenden

I filen build.sbt kan man lägga till många beroenden till flera olika kodbibliotek. Det finns på nätplatsen *Maven Central* en mycket omfattande koddatabas, som är sökbar här <http://search.maven.org>, med en massa användbara öppen-källkodsprojekt. Du kan be sbt att ladda ner den färdigkompilerade koden till vilket som helst av projekten på *Maven Central* och automatiskt lägga till jar-filen till classpath så att koden blir tillgänglig direkt i ditt program.

Till exempel kan du lägga till Java-biblioteket jline som gör det möjligt att göra terminalinläsning från tangentbordet med många bra finesser, t.ex. kommandohistorik med pil-upp, bara genom att lägga till denna rad i din build.sbt och den specifika version du önskar (notera enkla procent-tecken för Java-bibliotek):

```
libraryDependencies += "org.jline" % "jline" % "3.20.0"
```

Du kan läsa mer om jline här: <https://jline.github.io/>

## Appendix G

# Versionshantering och kodlagring

### G.1 Vad är versionshantering?

**Versionshantering**<sup>1</sup> (eng. *version control* eller *revision control*) av mjukvara innebär att hålla koll på olika versioner av koden i ett utvecklingsprojekt allteftersom koden ändras. Versionshantering är en deldisciplin inom **konfigurationshantering** (eng. *software configuration management*) som inbegriper allt i processen för att identifiera, besluta, genomföra och följa upp ändringar.

En viktig del av versionshantering är att *lagra* olika versioner av koden allt eftersom den utvecklas, så att tidigare versioner kan *återskapas* vid behov. Ett bra verktygsstöd och en väldefinierad arbetsprocess för versionshanteringen, som alla i utvecklingsprojektet följer, möjliggör att flera utvecklare kan *arbeta parallellt* med att sammanfoga (eng. *merge*) varandras tillägg och ändringar i den gemensamma kodbasen utan att det blir kaos och förvirring.

God versionshantering är helt avgörande för utvecklarnas produktivitet, speciellt för stora projekt med många utvecklare som jobbar parallellt mot en omfattande kodbas med många olika interna och externa komponenter. Men även ett litet projekt med en enda utvecklare kan ha god nytta av ett versionshanteringsverktyg och ett disciplinerat förfarande för att namnge versioner, t.ex. för att kunna återskapa tidigare versioner av projektets olika kodfiler när en ändring visar sig mindre lyckad.

Det finns flera olika modeller för hur kodlagringen sker:

- **lokal**; alla utvecklare jobbar i samma, lokala filsystem där alla olika versioner lagras.
- **centraliserad**; ett repositorium (förk. repo), alltså en databas med koden, finns centralt på en server som alla jobbar mot med hjälp av en versionshanteringsklient.
- **distribuerad**; alla utvecklare har sitt eget lokala repo och varje utvecklare initierar enskilt delning av ändringar mellan olika repo.

### G.2 Versionshanteringsverktyget Git

Det finns många olika versionshanteringsverktyg<sup>2</sup> som använder olika modeller för kodlagring; lokal, centraliserad, distribuerad eller kombinationer därav. På senare tid har verktyget **Git**<sup>3</sup> fått en stark ställning, speciellt i öppen-källkodsvärlden. Git utvecklades ursprungligen

---

<sup>1</sup>[en.wikipedia.org/wiki/Version\\_control](https://en.wikipedia.org/wiki/Version_control)

<sup>2</sup>[https://en.wikipedia.org/wiki/List\\_of\\_version\\_control\\_software](https://en.wikipedia.org/wiki/List_of_version_control_software)

<sup>3</sup>[https://en.wikipedia.org/wiki/Git\\_\(software\)](https://en.wikipedia.org/wiki/Git_(software))

av Linus Torvalds för att versionshantera Linuxkärnan, men har växt till ett omfattande öppen-källkodsprojekt med stor spridning och många användare och bidragsgivare.

Git är skapad för **distribuerad** versionshantering där var och en kan jobba snabbt och smidigt i sitt eget lokala repo, utan att behöva vänta på att en klient ska synkronisera koden med ett centralt repo på en server över nätverket. Ändringar delas mellan repo på begäran av enskilda utvecklare.

Varje ny version av koden lagras som en avgränsad mängd ändringar sedan förra versionen, en s.k. **commit**<sup>4</sup>, och hanteras internt av Git i en lokal databas i katalogen `.git` som ligger överst i din projektkatalog. Genom olika kommandon i terminalen, eller via en klient med ett grafiskt användargränssnitt, kan din kod överföras till och från den lokala koddatabasen, alternativt delas med andra repon via nätet.

Det finns en välskriven bok kallad ”*Pro Git*” som förklarar Git på djupet och är tillgänglig fritt här: <https://git-scm.com/book/en/v2>. Läs kapitel 1 och 2 så får du en bra grund att stå på.

Dessa termer är bra att kunna utantill innan du kör igång med Git:

- **repo** (*substantiv*: ett repositorium, *eng. a repository*) En koddatabas med ändringshistorik.
- **commit** (*substantiv*: en inlämning, *verb*: att lämna in). En avgränsad mängd nya ändringar lämnas in i det lokala repot. Repots ändringshistorik utgörs av sekvensen av alla inlämningar.
- **push** (*substantiv*: en leverans, *verb*: att leverera, att trycka upp). En eller flera inlämningar trycks upp till ett annat repo.
- **pull** (*substantiv*: en hämtning, *verb*: att hämta, att dra ner). En eller flera inlämningar dras ner från ett annat repo.
- **merge** (*substantiv*: en sammanfogning, *verb*: att sammanfoga). En eller flera inlämningar slås samman till en ny inlämning.
- **merge conflict** (*substantiv*: en sammanfogningskonflikt, *eng. a merge conflict*) Problem vid sammanfogning; ändringar kan inte enkelt sammanfogas på ett entydigt sätt.
- **pull request** (förk. PR, *substantiv*: en hämtningsbegäran, *verb*: att begära en hämtning). Utvecklare A ber en annan utvecklare B att hämta en eller flera inlämningar från A:s repo och sammanfoga med B:s repo.

### G.2.1 Installera git

Git finns förinstallerat på LTH:s Linuxdatorer. Du kan kolla om Git redan finns på din maskin genom att skriva `git help` i terminalen.

Det finns bra instruktioner om hur du installerar Git på din egen maskin här: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

VS Code har speciellt stöd för git och du kan inne ifrån VS Code göra t.ex. `add`, `commit`, `push` och `pull` via editorns grafiska gränssnitt. Läs mer här: <https://code.visualstudio.com/docs/editor/versioncontrol>

Det finns även många andra grafiska användargränssnitt till git, t.ex. **GitHub Desktop** (Windows/Mac) eller **GitKraken** (Linux/Windows/Mac). Se fler exempel här: <https://git-scm.com/downloads/guis>

<sup>4</sup>På svenska kan t.ex. ”inlämning” användas, men låneordet `commit` är redan etablerat.

## G.2.2 Anpassa Git

Innan du börjar använda git, konfigurera ditt namn och din email med nedan terminalkommando, där du anger ditt namn i stället för Förnamn Efternamn och din mejladress i stället för mejladr@plats.se. Namnet och mejladressen kommer lagras i varje commit som du gör så att det går att se vem som har gjort en given ändring.

```
> git config --global user.name "Förnamn Efternamn"
> git config --global user.email mejladr@plats.se
```

Läs mer om hur du gör andra inställningar här, t.ex. hur du anger vilken editor som git startar när du ska skriva commit-beskrivningar:

<https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>

## G.2.3 Använda git

Nedan listas några vanliga terminalkommandon i Git.

- Skapa ett repo i en katalog:

```
> cd myproject
> git init
```

- Se vilka filer som ändrats och ännu ej lämnats in:

```
> git status
> git status -s
```

- Se vilka ändringar som gjorts i filer som ännu ej lämnats in:

```
> git diff
```

- Se vilka inlämningar som finns i ändringshistoriken:

```
> git log
> git log --oneline -5
```

- Lägg till filer som ska ingå i nästa inlämning och gör sedan inlämningen; ge inlämningen en bra beskrivning som förklarar vad inlämningen omfattar:

```
> git add *.scala
> git commit -m 'initial project version'
```

- Ångra alla tillägg inför inlämning (ändringarna finns kvar och kan läggas till igen om du vill):

```
> git reset
```

- Du kan skippa de senaste, ännu ej commitade, ändringar i filen filename, och göra "undo", med kommandot `git checkout` på filen enligt nedan. Gör bara detta om du är helt säker på att du vill ångra dina senaste ändringar.

**WARNING!** Dina senaste ändringar i filen förloras för alltid; kan ej ångras!

```
> git checkout filename
```

- Man vill förhindra versionshantering av vissa filer, t.ex. binärkodsfiler så som **.class**-filer och andra genererade filer. Detta gör du genom att skapa en fil med namnet `.gitignore` och lägga in filändelser enligt nedan syntax, där `**/` avser alla kataloger och underkataloger och `*` kan vara vilken början på ett filnamn som helst. Symbolen `#` föregår en kommentarsrad.

```
# this is my .gitignore

# Java / Scala
**/*.class

# Sbt
**/target
```

## G.3 Kodlagringsplatser på nätet

Många utvecklare använder kodlagringsplatser på nätet ("i molnet") (eng. *code hosting*) för att underlätta samarbete kring kod och för att dela med sig av öppen källkod. Det finns många olika kodlagringsplatser som kan användas gratis under vissa förutsättningar eller mot betalning med tillhörande extratjänster.

**OBS!** Du får *inte* lagra dina lösningar på kursens laborationer i ett öppet repo. Om du vill använda en kodlagringsplats måste du säkerställa att dina lösningar förblir i ett stängt repo utan att någon annan kan komma åt det.

Nedan beskrivs några vanliga nätplatser för öppen och sluten kodlagring, som alla är Git-baserade:

- **GitHub**, <https://github.com>, är en av de mest populära kodlagringsplatserna för öppen källkod, men har även blivit en populär plats för jobbsökande utvecklare att visa upp sina kodarbetsprover för framtida arbetsgivare. GitHub är gratis att använda för dig som privatperson. Många företag betalar GitHub för att lagra sin stängda kod med tilläggstjänster för att testa, bygga och driftsätta kod etc. Koden som styr själva kodlagringsplatsen GitHub är stängd, till skillnad från GitLab. GitHub köptes 2018 av Microsoft för 65 miljarder kronor.
- **GitLab**, <https://gitlab.com>, erbjuder gratis kodlagring för öppen källkod, men det är även gratis för privatpersoner och gemenskapsprojekt att ha stängda repo. Företag kan betala för stängd kodlagring med extratjänster för att testa, bygga och driftsätta kod etc. GitLab är i sig ett öppen-källkodsprojekt och koden som styr kodlagringsplatsen är öppen och fri. Detta innebär att du själv kan ladda ner koden och starta en kodlagringsplats. LTH har en GitLab-baserad kodlagringsplats här: <https://git.cs.lth.se>
- **BitBucket**, <https://bitbucket.org>, är en populär kodlagringsplats både för öppen och stängd källkod och drivs av det australiensiska företaget Atlassian. Det är gratis för privatpersoner och små team att ha både öppna och slutna repon, men bara om det är få bidragsgivare. Kostnader tillkommer om antalet bidragsgivare kommer över en viss nivå. Universitetsanställda och studenter kan få mer gynnsamma villkor efter ansökan. Atlassian erbjuder en hel verktygssvit för att hantera buggar och samarbeta över nätet. BitBucket stödjer, förutom Git, även andra versionshanteringsverktyg.

### Använda kodlagringsplatser

Om du inte redan gjort det är det bra om du registrerar ditt användarnamn, förslagsvis fornamnefternamn som ett ord utan svenska tecken med små bokstäver, på någon eller alla av ovan sajter, dels för att paxa ditt namn och dels för börja samarbeta med utvecklare världen över. Det är bra att välja *ett* användarnamn för *alla* kodlagringsplatser på nätet, speciellt om du jobbar med öppen källkod där ditt namn kommer associeras med alla de kodbidrag du gör under ditt yrkesliv.

Om du inte vet vilken sajt du ska välja, börja med <https://github.com>. Om du vill att även kodlagringssajten ska drivas av öppen källkod, testa <https://gitlab.com>.

Med en Git-baserad kodlagringsplats får du möjlighet att synka ditt lokala repo mot en server på nätet med hjälp av git-kommandon i terminalen eller via en Git-klient med grafiskt användargränssnitt, se avsnitt G.2.1.

Innan du börjar använda en kodlagringsplats är det bra att sätta sig in i begreppen nedan.

- **clone** (*substantiv*: en klon är kopia av ett (nätlagrat) repo, *verb*: att klona, att skapa en kopia). Genom att klona ett repo som ligger på en nätlagringsplats kan du bygga, undersöka och vidareutveckla koden lokalt på din dator. Om du har rättigheter att lämna in kod till det centrala originalet kan du pusha dina commits direkt via terminalkommando eller Git-klient.
- **fork** (*substantiv*: en förgrening av ett helt repo, *verb*: att förgrena ett repo, att "forka"). Genom att förgrena ett repo skapar du en kopia, normalt även den nätlagrad på en kodlagringsplats, som du kan utveckla separat från originalet. Det blir då möjligt för dig att lämna in ändringar och trycka upp dem, även om du inte har rättigheter att leverera ("pusha") till originalet. Gör en ändringsbegäran (Pull Request, PR) om du vill bidra med dina ändringar, så kan ägaren av originalet sedan välja att sammanfoga ("merga") dina ändringar med originalet. Många nätlagringsplatser, så som GitHub, har en speciell knapp som du trycker på för att enkelt skapa en fork av ett repo under din användare.
- **upstream** (*preposition*: uppströms, *substantiv*: uppströmsrepo) Ett uppströmsrepo utgör original till ett förgrenat repo (en "fork").
  - Här beskrivs hur du länkar en förgrening uppströms:  
<https://help.github.com/articles/configuring-a-remote-for-a-fork/>
  - Här beskrivs hur du synkar en förgrening uppströms:  
<https://help.github.com/articles/syncing-a-fork/>

Om du vill bidra till ett öppen-källkodsprojekt, börja med att forka repot på kodlagringsplatsen och sedan klona repot till din lokala dator. Därefter kan du commita ändringar och pusha till din fork och slutligen göra en pull request från din fork till upstream. Läs om hur ett bidrag kan gå till i avsnitt 0.3.

Här följer några användbara kommandon:

- Skapa en lokal kopia av ett fjärran (eng. *remote*) repo; här visas hur du klonar kursens repo från GitHub:

```
$ git clone --depth 1 https://github.com/lunduniversity/introprog
```

- Dra ner nya inlämningar från ett fjärran repo:

```
$ git pull
```

- Trycka upp nya lokala inlämning till ett fjärran repo:

```
$ git push
```



## Appendix H

# Integrerad utvecklingsmiljö

## H.1 Vad är en integrerad utvecklingsmiljö?

En integrerad utvecklingsmiljö (eng. *integrated development environment*, *IDE*) samlar ett flertal verktyg för att skapa, köra och testa program, inklusive en avancerad **editor** och speciella felsökningsverktyg. Det finns flera utvecklingsmiljöer att välja mellan, som kan användas för både Scala och Java.

En IDE ger stöd för **kodkomplettering** (eng. *code completion*) där tillgängliga metoder visas i en lista och resten av ett namn kan fyllas i automatiskt efter att du skrivit de första bokstäverna i namnet. En IDE kan hjälpa dig med formatering och även skapa skelettkod utifrån **kodmallar** (eng. *code templates*). Med **felindikering** (eng. *error highlighting*) får du understrykning av vissa fel direkt i koden och ibland kan du även få hjälp med förslag på åtgärder för att rätta till enkla fel. Funktioner för **avlusning** (eng. *debugging*) hjälper dig att felsöka medan du kör din kod. Med funktioner för **omstrukturering** (eng. *refactoring*) av kod får du hjälp av editorn i samarbete med kompilatorn att göra omfattande strukturförändringar i många kodfiler samtidigt, t.ex. namnbyten med hänsyn taget till språkets synlighetsregler.

Alla dessa avancerade funktioner kan öka produktiviteten avsevärt, men samtidigt tar de tid att lära sig och en IDE kan kräva mycket datorkraft och viss väntetid jämfört med en vanlig, fristående editor. I början kan all funktionalitet upplevas som överväldigande och det kan vara svårt att hitta i alla menyer och inställningar. Det är därför många som föredrar en fristående, snabbstartad keditor före en fullfjädrad, tungrodd IDE, speciellt om det rör ett mindre program. Å andra sidan kan en IDE med kodkomplettering vara till stor hjälp när man ska lära sig ett nytt api och experimentera med en okänd kodmassa. Med tiden har hanliga editorer, så som VS Code, fått allt fler funktioner som tidigare bara fanns i en fullfjädrad IDE<sup>1</sup>, och den praktiska skillnaden allt mindre mellan en ”vanlig” editor och en IDE blir allt mindre.

I kursen använder vi flera utvecklingsmiljöer. På första labben använder vi Kojo (se appendix A) som är en IDE speciellt anpassad på nybörjare. Sedan använder vi en editor t.ex. VS Code, gärna i kombination med byggverktyget sbt. Du kan under andra halvan av kursen välja att gå över från VS Code till att använda en (annan) IDE, men det går utmärkt att fortsätta med VS Code som numera har en bra debugger för Scala genom tillägget Metals. Om du vill använda en IDE i stället för VS Code så rekommenderas IntelliJ IDEA med Scala-plugin. Om du redan lärt dig Eclipse på djupet och verkligen vill fortsätta med denna IDE, välj då ScalaIDE – dock har denna IDE inte hängt med i den tekniska utvecklingen och ligger kvar på Scala 2.12.

---

<sup>1</sup>Se t.ex. LSP: [https://en.wikipedia.org/wiki/Language\\_Server\\_Protocol](https://en.wikipedia.org/wiki/Language_Server_Protocol)



## H.2 Visual Studio Code med tillägget Scala Metals

Visual Studio Code<sup>2</sup>, förkortat VS Code eller bara code, är en gratis utvecklingsmiljö som är mestadels öppen källkod<sup>3</sup>. Projektet startades och leds av Microsoft och har en aktiv gemenskap med många utvecklare och många användbara tillägg (eng. *extensions*).

VS Code kallas ofta för ”bara” en editor, men har genom åren utvecklats till en fullfjädrad IDE med bl.a. inbyggd debugger och stöd för många olika språk via ett omfattande bibliotek av tillägg.

- Läs mer om hur man använder VS Code här:  
<https://code.visualstudio.com/docs>
- Läs mer om hur du använder Scala i VS Code här:  
<https://scalameta.org/metals/docs/editors/vscode>

Det finns många användbara kortkommandon som gör dig snabbare och snabbare när du kodar, allteftersom du lär dig nya kortkommandon. Ett bra tips är att du lär dig minst ett nytt kortkommando om dagen och efter ett tag kan du riktigt många. Här finns en sammanfattning av de viktigaste kortkommandona för VS Code för Linux:

<https://code.visualstudio.com/shortcuts/keyboard-shortcuts-linux.pdf>

Byt ut linux mot windows eller macos i adressen ovan för motsvarande plattform.

### H.2.1 Installera VS Code och Metals

VS Code är förinstallerad på LTH:s datorer, men du behöver själv installera Scala-tillägget **Metals** första gången du kör igång VS Code på LTH:s datorer. Läs om installation av Metals här:

<https://marketplace.visualstudio.com/items?itemName=scalameta.metals>

Läs mer om hur du installerar VS Code på din egen dator här:

<https://code.visualstudio.com>

Mer information om installation av verktyg finns på kursens hemsida:

<https://cs.lth.se/pgk/verktyg>

### H.2.2 Köra program i VS Code

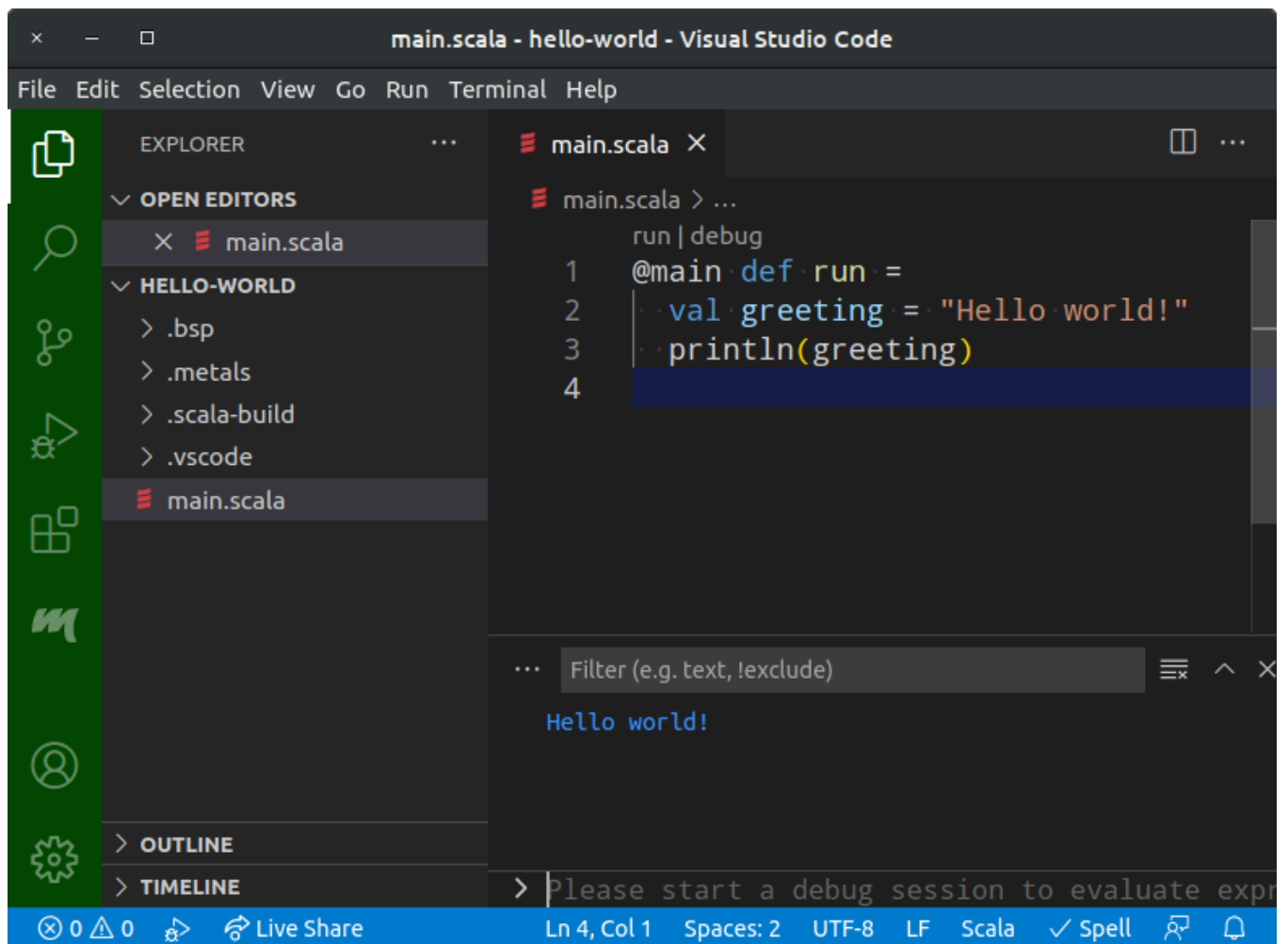
Det finns olika sätt att köra igång huvudprogrammet i ett projekt i VS Code:

1. Använd `scala-cli run .` i ett separat terminalfönster. Läs mer om `scala-cli` i Appendix C.4.3.
2. Kör igång `sbt` i ett separat terminalfönster och kör kommandot `run` inifrån `sbt`. Detta kräver att du har en giltig `build.sbt`, se Appendix F.
3. Köra igång program inifrån VS Code. Detta kräver att du öppnat katalogen med din kod med File-menyns ”Open Folder”, eller genom att du startar VS Code med `code .` överst i din projektkatalog (du ser att detta är gjort om nedre meddelandefältet är blått i stället för lila). Du behöver *innan* du startar VS Code en första gång köra `scala-cli setup-ide .` (se Appendix C.4.3), eller skapa en giltig `build.sbt`-fil (se Appendix F) som du importerar i VS Code när frågan dyker upp i nedre högra hörnet.

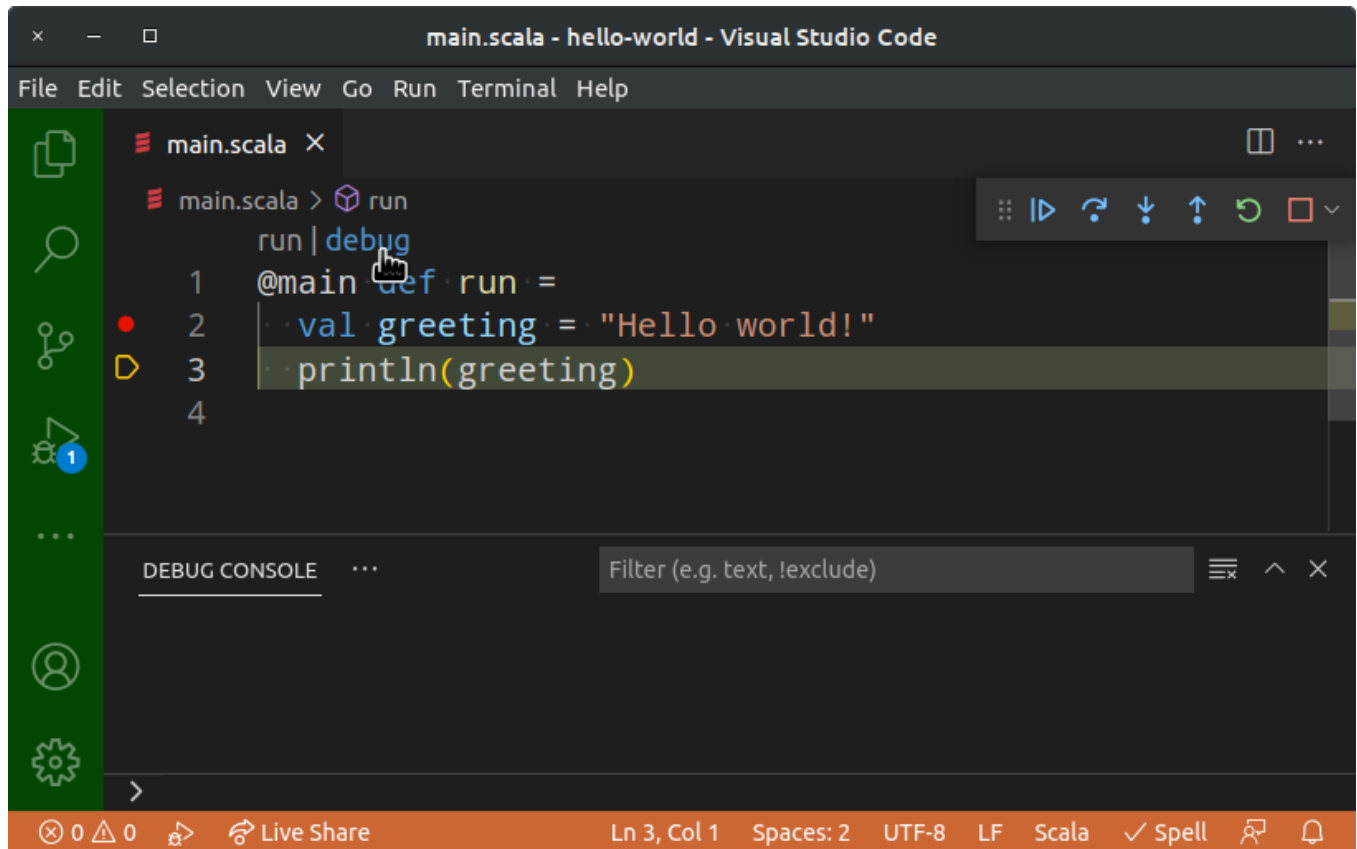
---

<sup>2</sup>[en.wikipedia.org/wiki/Visual\\_Studio\\_Code](https://en.wikipedia.org/wiki/Visual_Studio_Code)

<sup>3</sup>Varianten VS Codium <https://vscodium.com/> är helt fri från stängd källkod och telemetri.



**Figur H.1:** Kör program genom att klicka på RUN ovanför huvudprogrammet.



**Figur H.2:** Debuggern i VS Code. Nederkanten är orange när debuggern kör.

4. Kombinera Scala CLI eller sbt och VS Code. Kör detta kommando i ett separat terminalfönster: `scala-cli compile . -w` där `-w` betyder *watch* och gör så att ändringar bevakas. Om du istället använder sbt kör `sbt ~compile` i ett separat terminalfönster (notera tilde-tecknet som gör att ändringar bevakas). Vid ändringsbevakning kommer kompileringsfel visas där varje gång du sparar en ändring i VS Code med `Ctrl+S`. När alla kompileringsfel är åtgärdade och du är redo att testköra så klickar du på `run`.

Du ser att VS Code är beredd att köra igång ditt program genom att det (efter ett tag) kommer upp en extra rad ovanför ditt huvudprogram med texten `run | debug` och då kan du klicka på `run` för att köra ditt program. Utdata från körningen visas i en flik under koden. Observera att det kan ta lite tid för VS Code att förbereda allt som behövs för att kunna köra ditt program. Håll koll på om VS Code håller på med dessa förberedelser i det blåa meddelandefältet längst ned till höger. När allt är klart efter att du startat VS Code står det "Index complete!" bredvid en raketsymbol i meddelandefältet.

Om något krånglar och du inte får fram `run | debug` ovanför din `@main`-funktion, trots du har startat VS code enligt ovan, så prova att under Metals-fliken (ikonen med det stiliserade M:et i det gröna verktygsfältet) klicka på någon av "Restart build server" eller "Import build" (den senare tar längre tid men börjar om helt) och vänta tills det står "Index Complete!" i det blå meddelandefältet och då ska `run | debug` synas ovanför din `@main`-funktion.

### H.2.3 Använda debuggern i VS Code

Innan du börjar använda debuggern, läs först om allmän felhantering i Appendix D.

Du kan aktivera debuggern i VS Code för dina Scala-program genom att klicka på "debug" ovanför din `main`-metod, förutsatt att du har tillägget Metals installerad i VS Code. Du behöver även köra `scala-cli setup-ide .` en första gång (se Appendix C.4.3), eller ha en

giltig `build.sbt`-fil (se Appendix F) som du importerar i VS Code när frågan dyker upp i nedre högra hörnet.

Figur H.2 på sidan 504 visar hur det kan se ut när debuggern i VS Code är aktiverad. När debuggern är igång får det nedersta meddelandefältet en orange färg (istället för blå). Till vänster om radnummerkolumnen kan du klicka för att aktivera och avaktivera brytpunkter. Aktiverade brytpunkter visas som en röd prick i marginalen till vänster. Den ihåliga gula pilen i marginalen pekar på den rad som kommer att exekveras närmast. Notera panelen med olika knappar i överkanten av editorfönstret. Med dessa knappar kan du styra exekveringen enligt följande (lär dig gärna kortkommandona så blir du snabbare):

- **Fortsätt.** Den blåa play-knappen kör vidare till nästa brytpunkt eller tills programmet är klart om brytpunkt ej påträffas. Kortkommando "Continue": F5.
- **Stega över.** Den blåa böjda framåtpilen kör en rad i taget *utan* att hoppa in i funktioner. Kortkommando "Step Over": F10.
- **Stega in.** Den blåa nedåtpilen kör vidare en rad i taget och hoppar in i funktioner om raden innehåller funktionsanrop. Kortkommando "Step Into": F11.
- **Stega ut.** Den blåa uppåtpilen kör klar innevarande funktion. Kortkommando "Step Out": Shift+F11.
- **Kör igen.** Den gröna återstartsikonen kör om ditt program. Kortkommando "Restart": Ctrl+Shift+F5.
- **Avbryt.** Den röda stoppknappen avbryter denna debuggingsession. Kom ihåg att avbryta innan du startar en ny debuggingsession, annars kan det lätt bli förvirrande med många samtidigt pågående körningar. Kortkommando "Stop": Shift+F5.

Figur H.3 på sidan 506 visar hur VS Code presenterar anropsstacken och värdet på de variabler som syns där exekveringen befinner sig för tillfället. Du får fram detta genom att klicka på ikonerna med en lus och en playknapp i det vertikala, gröna verktygsfältet längst till vänster. I en blå ring står en etta om du har startat en debuggingsession. Om det står en tvåa eller mer så har du flera sessioner igång och då kan det vara klokt att avsluta alla utom en, så att inte förvirring uppstår om vilken session som är den aktuella.

Mycket av konsten i debugging handlar om att undersöka variabelers värde under exekveringen för att ta reda på om din hypotes om vad som händer under exekvering verkligen stämmer, eller om något egentligen inte fungerar så som du antar. Detta kan du med fördel göra genom att placera brytpunkter på relevanta ställen. Även vid användning av en debugger kan du ha stor nytta av att göra `println` av intressanta uttryck för att i detalj undersöka vad som egentligen händer. Läs mer om debugging i Appendix D.

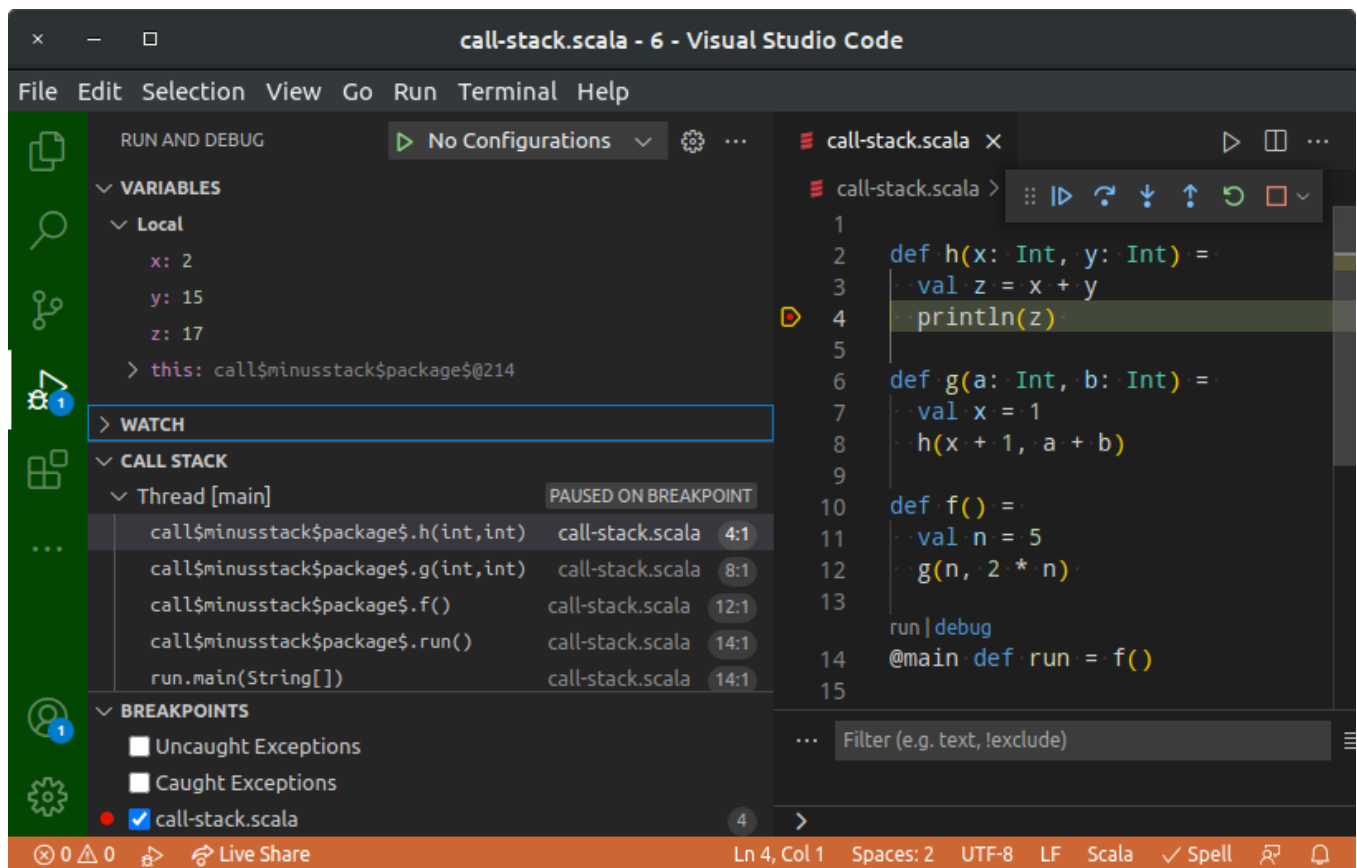
### H.3 JetBrains IntelliJ IDEA med Scala-plugin

IntelliJ IDEA<sup>4</sup> är en professionell IDE som stödjer många olika programmeringsspråk. IntelliJ är skriven i Java och utvecklas av det tjeckiska företaget JetBrains.

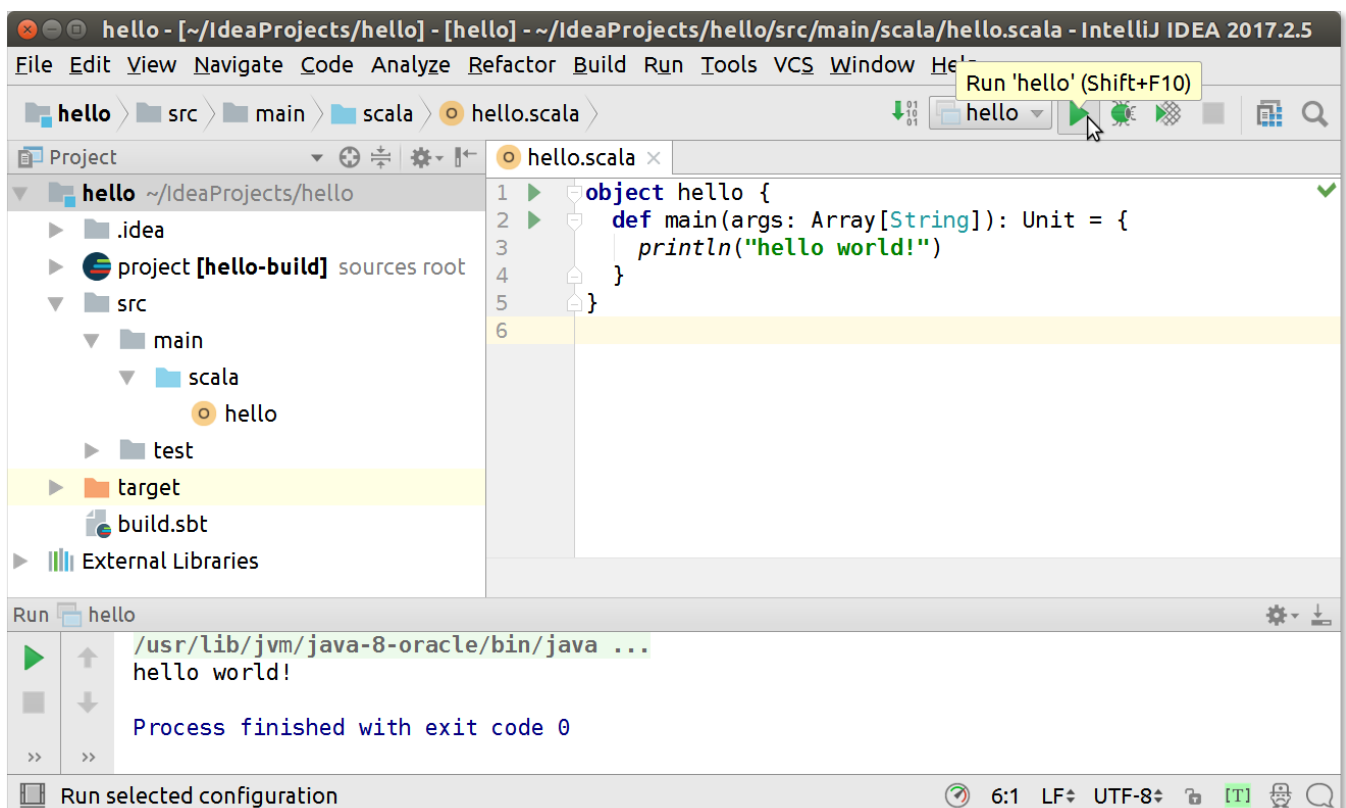
IntelliJ IDEA finns i två varianter: en gratis gemenskapsvariant med öppen källkodslicens (eng. *Community edition*), samt en betalvariant med sluten källkod och support-tjänster.

IntelliJ IDEA är en omfattande och avancerad programmeringsmiljö med många funktioner och inställningar. Det finns även en omfattande uppsättning insticksmoduler och

<sup>4</sup>[en.wikipedia.org/wiki/IntelliJ\\_IDEA](https://en.wikipedia.org/wiki/IntelliJ_IDEA)



**Figur H.3:** Anropsstack och variabler i VS Code.



**Figur H.4:** Den integrerade utvecklingsmiljön IntelliJ IDEA.

tillägsprogram som underlättar utveckling av t.ex. mobilappar, webbprogram, databaser och mycket annat.

Till IntelliJ IDEA finns en insticksmodul (eng. *plug-in*) som stöd för Scala med tillhörande standardbibliotek och byggverktyget sbt, med mera. Scala-insticksmodulen kan inkluderas genom att välja Scala i en av de dialoger som visas vid första körningen, enligt instruktioner nedan.

I detta avsnitt ges länkar till installation samt tips om hur du kommer igång med att använda IntelliJ IDEA med Scala. Det går ganska snabbt att lära sig grunderna, men det kräver en viss ansträngning att lära sig de mer avancerade funktionerna. Det finns omfattande resurser på nätet som hjälper dig vidare.

Google tillkännagav 2013 att företaget övergår från Eclipse till IntelliJ som den officiellt understödda utvecklingsmiljön för Android och 2014 lanserades utvecklingsmiljön Android Studio<sup>5</sup> som bygger vidare på IntelliJ.

### H.3.1 Installera IntelliJ IDEA

IntelliJ med Scala-plugin är förinstallerat på LTH:s datorer och startas med kommandot `idea` i ett terminalfönster.

Du kan installera IntelliJ på din egen dator genom att följa instruktionerna för ditt operativsystem (Windows/macOS/Linux) här:

<https://www.jetbrains.com/help/idea/run-for-the-first-time.html>

Du behöver Scala-plugin som du kan välja under installationen av IntelliJ, men det går också att installera plugin för Scala i efterhand, se vidare här:

<https://www.jetbrains.com/help/idea/discover-intellij-idea-for-scala.html>

---

<sup>5</sup>[en.wikipedia.org/wiki/Android\\_Studio](https://en.wikipedia.org/wiki/Android_Studio)

## Appendix I

# Skapa webb-appar med ScalaJS

- Lär mer här: <http://scala-js.org/>
- Exempel: <https://github.com/bjornregnell/kapten-alloc-web>
- Exempel på konfiguration av bygge:
  - Lägg in plugin för ScalaJS i `project/plugins.sbt`:  
<https://github.com/bjornregnell/kapten-alloc-web/blob/master/project/plugins.sbt>
  - Lägg till ScalaJS-biblioteket i `build.sbt`:  
<https://github.com/bjornregnell/kapten-alloc-web/blob/master/build.sbt>
  - Lägg till `<script>`-tag i `index.html`:  
<https://github.com/bjornregnell/kapten-alloc-web/blob/master/index.html>
  - Skapa webbappen med `sbt fastLinkJS` vid utveckling och när klar så skapa optimerad app med `sbt fullLinkJS`

## **Appendix J**

# **Introduktion till Java**



## J.1 Teori

### J.1.1 Övning scalajava och labb javatext

- Övning scalajava:
  - Översätta Java till Scala och från Scala till Java
  - Undersöka autoboxning (eng. *autoboxing*)
  - Använda **import** `scala.jdk.CollectionConverters.*`
- Laboration javatext:
  - Gör ett textspel för terminalen huvudsakligen i Java men vissa delar i Scala, enligt krav, tips och inspiration i labb-instruktionerna.
- Tips: Scala CLI och sbt kan blanda `.scala` och `.java` i samma projekt.

---

### J.1.2 "Hello world!" i Java.

Ett minimalt huvudprogram i Java:

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

---

### J.1.3 Testa Java i jshell

- Java har en motsvarighet till Scalas REPL: kommandot `jshell`

```
> jshell  
| Welcome to JShell -- Version 11.0.11  
| For an introduction type: /help intro  
  
jshell> /help intro  
|  
|                                     intro  
|                                     =====  
|  
| The jshell tool allows you to execute Java code, getting immediate results.  
| You can enter a Java definition (variable, method, class, etc), like: int x = 8  
| or a Java expression, like: x + x  
| or a Java statement or import.  
| These little chunks of Java code are called 'snippets'.  
|  
| There are also the jshell tool commands that allow you to understand and  
| control what you are doing, like: /list  
|  
| For a list of commands: /help  
  
jshell>
```

## J.1.4 Grundläggande likheter och skillnader Java–Scala

### Några likheter:

- Kompilerar till bytekod som kör på JVM på många olika plattformar.
- Statisk typning: ger snabb maskinkod, kompilatorn kan ge stöd vid förändring av kod (s.k. refactoring) och hittar många buggar redan vid kompilering.

### Liknande men **viss skillnad**:

#### Java

- **Objektorientering**, men inte "äka" (eng. *pure*) eftersom inte alla värden är objekt
- Primitiva typer är inte objekt; representeras effektivt, normalt **utan boxning**
- Visst stöd för **funktionsprogrammering**

#### Scala

- **Äkta objektorienterat** eftersom alla värden är objekt, även funktioner
- AnyVal-instanser är äkta objekt men representeras ändå effektivt, normalt **utan boxning**
- Omfattande stöd för **funktionsprogrammering**

## J.1.5 Huvudprogram i Scala och Java

### Scala 2

```
object Main {
  def main(args: Array[String]): Unit = {
    println("Hello!")
  }
}
```

### Java

```
public class JMain {
  public static void main(String[] args){
    System.out.println("Hello!");
  }
}
```

### Scala 3

```
@main
def sumFirst(n: Int, xs: Int*): Unit = println(xs.take(n).sum)
```

## J.1.6 Loopa genom argumenten i ett Java-huvudprogram

```
> code HelloJavaArgs.java
```

```
public class HelloJavaArgs {
  public static void main(String[] args) {
    int i = 0;
    while (i < args.length) {
      System.out.println(args[i]);
      i = i + 1;
    }
  }
}
```

```
    }  
  }  
}
```

Kompilera och kör:

```
1 > javac HelloJavaArgs.java  
2 > java HelloJavaArgs hej gurka tomat  
3 hej  
4 gurka  
5 tomat
```

---

### J.1.7 HIGHSCORE implementerad i Java

```
import java.util.Scanner;  
  
public class HighScore {  
    public static void main(String[] args){  
        Scanner scan = new Scanner(System.in);  
        System.out.println("Hur många poäng fick du?");  
        int points = scan.nextInt();  
        System.out.println("Vad var highscore före senaste spelet?");  
        int highscore = scan.nextInt();  
        if (points > highscore) {  
            System.out.println("GRATTIS!");  
        } else {  
            System.out.println("Försök igen!");  
        }  
    }  
}
```

---

### J.1.8 Några saker som finns i Scala men inte i Java

- **case**-klasser
- Lokala funktioner
- Metoder som operatorer
- Infix operatornotation
- Defaultargument
- Namngivna argument
- Engångsinitialisering: **val**
- Fördröjd initialisering: **lazy val**
- Enhetlig access för **def**, **val**, **var**
- Egna setters med **def** namn\_ =
- Namnanrop, fördröjd evaluering
- Matchning, mönster och garder

- Klassparametrar, primärkonstruktor
- Singelobjekt: **object**
- Kompanjonsobjekt
- Inmixning: **trait**
- **for-*yield***-uttryck
- Block är uttryck; slipper **return**
- Tomma värdet () av typen Unit
- Option, Some, None (Java har Optional som ger en del, men inte allt...)
- Try, Success, Failure
- Samlingarna i Scalas standardbibliotek, speciellt de **oföränderliga** samlingarna Vector, Map, Set, List, etc.
- Innehållslighet med == för oföränderliga strukturer, inkl. < <= > >= på strängar
- **Enhetlig** användning av samlingar **inkl. Array** (förutom innehållslighet för Array)
- Kontextuella abstraktioner **given using**
- Mer precis synlighet **private**[mypack]
- Namnändring vid **import**
- Flexibel filstruktur och filnamngivning
- Flexibel nästling av klasser, objekt, traits
- Typ-alias och abstrakta typer med **type**
- Extensionsmetoder **extension**
- ...

### J.1.9 Några saker som finns i Java men inte i Scala

- + Snabbare kompilering
- + Mognare verktygsstöd
- Variabeldeklaration utan initialisering
- Förändringsbara parametrar
- C-liknande prefix- och postfix-inkrementering och -dekrementering: `i++ ++i i-- --i`
- C-liknande **for**-sats
- Semikolon krävs efter alla satser
- **return** krävs i alla metoder som har returvärde
- Nyckelordet **void**
- Parenteser efter alla metoder
- Specialsyntax för indexering av array [] ej som i andra samlingar
- Hoppa ut ur loop med **break**  
[docs.oracle.com/javase/tutorial/](https://docs.oracle.com/javase/tutorial/)
- **switch** "faller igenom" om du glömmer skriva **break**
- Kontrollerade undantag (eng. *checked exceptions*) och **throws**
- ...

### J.1.10 Begränsningar med funktionsprogrammering i Java

Av alla dessa funktionsprogrammeringskoncept i Scala...

- **överlagring**
- **anonyma funktioner**
- mönstermatchning
- funktioner etc. på toppnivå
- utelämna tom parameterlista (enhetlig access)
- defaultargument
- namngivna argument
- lokala funktioner
- funktioner som äkta värden
- klammerparentes vid ensam paramenter
- multipla parameterlistor
- egendefinierade kontrollstrukturer
- namnanrop (fördröjd evaluering)
- stegade funktioner ("Curry-funktioner")
- fångad variabelrymd i funktionsobjekt ("closure")
- ad hoc polymorfism ("typklasser")
- kontextuella abstraktioner

...kan man i Java endast göra: **överlagring** ("overloading") och **anonyma funktioner** ("lambda") där det senare har starka begränsningar.

Läs mer om Java här:

[https://en.wikipedia.org/wiki/Java\\_version\\_history](https://en.wikipedia.org/wiki/Java_version_history) [https://en.wikipedia.org/wiki/Anonymous\\_function#Java\\_Limitations](https://en.wikipedia.org/wiki/Anonymous_function#Java_Limitations)

### J.1.11 Grundtyper i Scala och primitiva typer Java

<b>Grundtyp i</b> <b>Scala</b>	<b>Antal</b> bitar	<b>Omfång</b> minsta/största värde	<b>primitiv typ i</b> <b>Java</b>
Byte	8	$-2^7 \dots 2^7 - 1$	byte
Short	16	$-2^{15} \dots 2^{15} - 1$	short
Char	16	$0 \dots 2^{16} - 1$	char
Int	32	$-2^{31} \dots 2^{31} - 1$	int
Long	64	$-2^{63} \dots 2^{63} - 1$	long
Float	32	$\pm 3.4028235 \cdot 10^{38}$	float
Double	64	$\pm 1.7976931348623157 \cdot 10^{308}$	double

### J.1.12 Javas switch-sats

De flesta C-liknande språk (men inte Scala) har en **switch**-sats som man kan använda istället för (vissa) nästlade if-else-satser:

```
import java.util.Scanner;
```

```
public class Switch {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        System.out.println("Skriv grönsak:");  
        String g = scan.next();  
        switch (g) {  
            case "gurka":  
                System.out.println("gott!");  
                break;  
            case "tomat":  
                System.out.println("gott!");  
                break;  
            case "broccoli":  
                System.out.println("ganska gott...");  
                break;  
            default:  
                System.out.println("mindre gott...");  
                break;  
        }  
    }  
}
```

switch i Java har stora begränsningar (fungerar t.ex. bara på primitiva typer och några till, tex String); i framtiden planerar man att anamma en del av det **match** i Scala kan.

### J.1.13 Javas switch-sats utan break

Saknad **break**-sats ”faller igenom” till efterföljande gren:

```
import java.util.Scanner;  
  
public class SwitchNoBreak {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        System.out.println("Skriv grönsak:");  
        String g = scan.next();  
        switch (g) {  
            case "gurka":  
            case "tomat":  
                System.out.println("gott!");  
                break;  
            case "broccoli":  
                System.out.println("ganska gott...");  
                break;  
            default:  
                System.out.println(" mindre gott...");  
                break;  
        }  
    }  
}
```

```
}  
}
```

En glömd **break** kan ge svårhittad bugg...

---

#### J.1.14 Javas switch-sats med glömd break

```
import java.util.Scanner;  
  
public class SwitchForgotBreak {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        System.out.println("Skriv grönsak:");  
        String g = scan.next();  
        switch (g) {  
            case "gurka":  
                System.out.println("gott!");  
            case "tomat":  
                System.out.println("gott!");  
                break;  
            case "broccoli":  
                System.out.println("ganska gott...");  
                break;  
            default:  
                System.out.println("mindre gott...");  
                break;  
        }  
    }  
}
```

```
> java SwitchForgotBreak  
Skriv grönsak:  
gurka  
gott!  
gott!
```

### J.1.15 Syntax för variabeldeklaration i Scala och Java

Exempel på variabeldeklarationer i

#### Scala

```
var i1: Int = 0
var i2 = 0
var i3 = 0: Int
var p1: Point = new Point(0, 0)
var p2 = new Point(0, 0)
var (x, y) = (0, 0)
val a = 0
final val Constant = 42
```

- i2 härledd typ; går ej i Java 8,9 men finns med **var** i Java 10
- i3 typ varhelst i uttryck; går ej i Java
- (x, y) mönster i init; går ej i Java
- **val** ger "engångsinit"; ingen exakt motsvarighet i Java men **final** kan ofta användas i stället

#### Java

```
int i1 = 0;
var i2 = 0; // från Java 10
int i4;
Point p1 = new Point(0, 0);
var p2 = new Point(0, 0);
final int CONSTANT = 42;
```

- i4 ej explicit init; går ej i Scala

### J.1.16 For-sats i Scala och Java

#### Scala

```
val s = "Abbasillen"

// Loopa över index framlänges:

for i <- 0 until s.length do
  println(s(i))

// Loopa över index baklänges:

for i <- s.length-1 to 0 by -1 do
  println(s(i))
```

I Scala är s.indices att föredra!

#### Java

```
String s = "Abbasillen";

// Loopa över index framlänges:

for (int i = 0; i < s.length(); i++) {
  System.out.println(s.charAt(i));
}

// Loopa över index baklänges:

for (int i = s.length()-1; i >= 0; i--) {
  System.out.println(s.charAt(i));
}
```



### J.1.17 For-sats i Scala med indices

#### Scala

```
val s = "Abbasillen"

// Loopa över index framlänges:

for i <- s.indices do
  println(s(i))

// Loopa över index baklänges:

for i <- s.indices.reverse do
  println(s(i))
```

#### Java

```
String s = "Abbasillen";

// Loopa över index framlänges:

for (int i = 0; i < s.length(); i++) {
    System.out.println(s.charAt(i));
}

// Loopa över index baklänges:

for (int i = s.length()-1; i >= 0; i--) {
    System.out.println(s.charAt(i));
}
```

---

### J.1.18 For-satser och arrayer i Java

En for-sats i Java har följande struktur:

```
for (initialisering; slutvillkor; inkrementering) {
    sats1;
    sats2;
    ...
}
```

En primitiv heltals-array deklareras så här i Java:

```
int[] xs = new int[42]; // rymmer 42 st heltal, init 0:or
int[] ys = {10, 42, -1}; // initiera med 3 st heltal
```

Exempel på for-sats: fyll en array med 1:or

```
for (int i = 0; i < xs.length; i++){
    xs[i] = 1; // indexera med [i]
}
```

---

### J.1.19 Implementation av SEQ-COPY i Java med for-sats

```

1 public class SeqCopyForJava {
2
3     public static int[] arrayCopy(int[] xs){
4         int[] result = new int[xs.length];
5         for (int i = 0; i < xs.length; i++){
6             result[i] = xs[i];
7         }
8         return result;
9     }
10
11     public static String test(){
12         int[] xs = {1, 2, 3, 4, 42};
13         int[] ys = arrayCopy(xs);
14         for (int i = 0; i < xs.length; i++){
15             if (xs[i] != ys[i]) {
16                 return "FAILED!";
17             }
18         }
19         return "OK!";
20     }
21
22     public static void main(String[] args) {
23         System.out.println(test());
24     }
25 }

```

Lite syntax och semantik för Java:

- En Java-klass med enbart statiska medlemmar motsvarar ett singelobjekt i Scala.
- Typen kommer **före** namnet.
- Man **måste** skriva **return**.
- Man **måste** ha semikolon efter varje sats.
- Metodnamn **måste** följas av parenteser; om inga parametrar finns används ()
- En array i Java är inget vanligt objekt, men har ett "attribut" length som ger antal element.
- **Övning:** skriv om med **while**-sats i stället; har samma syntax i Scala & Java.

### J.1.20 Element för element med speciell for-each-sats i Java

#### Scala

```

val s = "Abbasillen"

// Loopa över alla tecken:

for ch <- s do println(ch)

```

#### Java

```

String s = "Abbasillen";

// Loopa över alla tecken:

for (char ch: s.toCharArray()) {
    System.out.println(ch);
}

```

s.foreach(println) går ej i Java men från Java 8 finns metoden chars som ger en IntStream och då kan man:

```
str.chars().forEachOrdered(i -> System.out.println((char) i));
```

### J.1.21 Typisk utformning av Java-klass

Typisk "anatomy" hos en Java-klass:

```

public class Klassnamn {
    attribut, normalt privata
    konstruktorer, normalt publika
    metoder: publika getters, och vid förändringsbara objekt även setters
    metoder: privata abstraktioner för internt bruk
    metoder: publika abstraktioner tänkta att användas av klientkoden
}

```

### J.1.22 Statiska medlemmar i Java

- Man kan **inte** deklarera explicita singelobjekt i Java och det finns inget nyckelord **object**.
- I stället kan man deklarera **statiska medlemmar** i en klass med Java-nyckelordet **static**.
- Exempel på hur vi ska göra detta inuti en klassen JPerson:

```
public static final int ADULT_AGE = 18;
```

- Effekten blir den samma som ett singelobjekt i Scala:
  - Alla statiska medlemmar i en Java-klass allokeras automatiskt och hamnar i en egen singular ”klassinstans” som existerar oberoende av de dynamiska instanserna.
  - De statiska medlemmarna accessas med punktnotation genom klassnamnet, **utan new**:

```
System.out.println(JPerson.ADULT_AGE);
```

---

### J.1.23 Exempel: oföränderlig klass i Scala och Java

#### Scala:

```
class Person(val name: String, val age: Int):  
  def isAdult = age >= Person.AdultAge  
  
object Person:  
  val AdultAge = 18
```

#### Java:

```
public class JPerson {  
    private String name;  
    private int age;  
    public static final int ADULT_AGE = 18;  
  
    public JPerson(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

```

    public boolean isAdult() {
        return age >= ADULT_AGE;
    }
}

```

Lär dig detta mönster för en typisk Java-klass utantill så du snabbt får grejerna på plats!

### Övning:

Gör Person + JPerson **förändringsbara** så att namnet och åldern går att uppdatera och följande krav uppfylls:

- namnet ska ges vid konstruktion,
- åldern ska initieras till 0 vid konstr.,
- åldern ska aldrig kunna bli negativ.

#### J.1.24 Exempel: Scala-klassen Complex

```

1 class Complex private (val re: Double, val im: Double):
2   def r = math.hypot(re, im)
3   def fi = math.atan2(im, re)
4   def +(other: Complex) = new Complex(re + other.re, im + other.im)
5   override def toString = s"$re + $im${Complex.imSymbol}"
6
7 object Complex:
8   var imSymbol = 'i'
9   def apply(re: Double = 0, im: Double = 0) = new Complex(re, im)
10  def real(r: Double) = apply(re = r)
11  def imag(i: Double) = apply(im = i)
12  val zero = apply()

```

Scala: <https://github.com/lunduniversity/introprog/blob/master/compendium/examples/complex7.scala>

Java: <https://github.com/lunduniversity/introprog/blob/master/compendium/examples/JComplex.scala>

#### J.1.25 Exempel: Motsvarande Java-klassen JComplex

```

1 public class JComplex { // man kan ej deklarerera klassparametrar i Java
2   private double re; // initialiseras i konstruktorn nedan
3   private double im; // initialiseras i konstruktorn nedan
4   public static char imSymbol = 'i'; // publikt förändringsbart attribut (ovanlig
5
6   public JComplex(double real, double imag){ // konstruktor, körs vid new
7     re = real;
8     im = imag;

```

```

9      }
10
11 // en "getter" som ger attributvärdet, hindrar förändring av re
12     public double getRe(){
13         return re;
14     }
15
16 // ej bruklig formattering i Java, så metoder blir minst 3 rader
17     public double getIm(){ return im; }
18
19     public double getR(){
20         return Math.hypot(re, im);      // Math med stort M i Java
21     }
22
23     public double getFi(){
24         return Math.atan2(re, im);
25     }
26
27 // Javametodnamn får ej ha operatortecken t.ex. +, därav namnet add
28     public JComplex add(JComplex other){
29         return new JComplex(re + other.getRe(), im + other.getIm());
30     }
31
32     @Override public String toString(){
33         return re + " + " + im + imSymbol;
34     }
35 }

```

### J.1.26 Exempel: Använda JComplex i Scala-kod

```

1 $ javac JComplex.java
2 $ scala
3 Welcome to Scala 2.12.9 (OpenJDK 64-Bit Server VM, Java 1.8.0_222).
4 Type in expressions for evaluation. Or try :help.
5
6 scala> val jc1 = JComplex(3, 4)
7 jc1: JComplex = 3.0 + 4.0i
8
9 scala> val polarForm = (jc1.getR, jc1.getFi)
10 polarForm: (Double, Double) = (5.0,0.6435011087932844)
11
12 scala> val jc2 = JComplex(1, 2)
13 jc2: JComplex = 1.0 + 2.0i
14
15 scala> jc1 add jc2
16 res0: JComplex = 4.0 + 6.0i

```

### J.1.27 Exempel: Använda JComplex i Java-kod

```
public class JComplexTest {
    public static void main(String[] args){
        JComplex jc1 = new JComplex(3,4);
        String polar = "(" + jc1.getR() + ", " + jc1.getFi() + ")";
        System.out.println("Polär form: " + polar);
        JComplex jc2 = new JComplex(1,2);
        System.out.println(jc1.add(jc2));
    }
}
```

- I Java måste man skriva **new**.
- I Java måste man skriva **tomma parentes-par** efter metodnamnet vid **anrop av parameterlösa metoder**.
- **Tupler finns inte** i Java, så det går inte på ett enkelt sätt att skapa par av värden som i Scala; ovan görs polär form till en sträng för utskrift.
- **Operatornotation för metoder finns inte** i Java, så man måste i Java använda punktnotation och skriva: `jc1.add(jc2)`

### J.1.28 Exempel: Förändringsbar klass i Scala och Java

#### Scala:

```
class MutablePerson(var name: String):
    private var _age = 0

    def age: Int = _age

    def age_=(a: Int): Unit =
        if (a >= 0) _age = a else _age = 0
        // eller hellre kasta undantag?

    def isAdult: Boolean =
        age >= MutablePerson.AdultAge

object MutablePerson:
    val AdultAge = 18
```

#### Java:

```
public class JMutablePerson {
    private String name;
    private int age = 0;
    public static final int ADULT_AGE = 18;
```

```
public JMutablePerson(String name) {  
    this.name = name;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public int getAge() {  
    return age;  
}  
  
public void setAge(int age) {  
    if (age >= 0) {  
        this.age = age;  
    } else {  
        this.age = 0;  
    }  
}  
  
public boolean isAdult() {  
    return age >= ADULT_AGE;  
}  
}
```

---

### J.1.29 Scalas "case-klass-godis" finns inte i Java

En oföränderlig datatyp implementeras i **Scala** helst som en **case**-klass:

```
case class Person(name: String, age: Int):  
  def isAdult = age >= Person.AdultAge  
  
object Person:  
  val AdultAge = 18
```

En oföränderlig datatyp i **Java** med **motsvarande** funktionalitet kräver egen implementation av dessa metoder:

- en getter för varje attribut
- equals
- hashCode (förklaras i forts.kurs)
- apply  
(men man kallar nog den create el. likn.; namnet måste ju skrivas)
- toString
- copy  
(men det finns ju inte namngivna parametrar och default-argument så denna blir osmidig)
- unapply  
(men det finns ju inte mönstermatchning så denna struntar man nog i)

---

### J.1.30 Repetition: Den primitiva typen Array i JVM

- Primitiva arrayer (Array i Scala, [] i Java) har **fördelar**:<sup>1</sup>
  - Det är den snabbaste indexerbara datastrukturen i JVM: att läsa och uppdatera ett element på en viss plats är mycket effektivt om man vet platsens index.
  - Fungerar lika bra med både primitiva värden och objektreferenser
- ... men också **nackdelar**:
  - Man måste bestämma sig för antalet element som ska allokeras när man gör **new**.
  - Man kan ta i lite extra när man allokerar om man behöver plats för fler senare, men då måste man hålla reda på hur många platser man använder och veta var nästa lediga plats finns.
  - Det är krångligt att stoppa in (eng. *insert*) och ta bort (eng. *delete*) element.
  - Vill man ha fler platser måste man allokera en helt ny, större array och kopiera över alla befintliga element.

---

### J.1.31 Syntax för Array i Scala och Java

---

<sup>1</sup>[stackoverflow.com/questions/2843928/benefits-of-arrays](https://stackoverflow.com/questions/2843928/benefits-of-arrays)



**Scala**

```

var xs = Array(42, 43, 44)

val n = xs.length

var strings = new Array[String](42)
// eller      Array.ofDim[String](42)
// eller      Array.fill(42)(null: String)

strings(0) = "first"

strings(1) = "second"

```

**Java**

```

int[] xs = new int[]{42, 43, 44};

// samma som ovan, men kortare:
int[] xs2 = {42, 43, 44};

int n = xs.length; // EJ length()

String[] strings = new String[42];

strings[0] = "first";

strings[1] = "second";

```

**J.1.32 Exempel: Polygon med primitiv array i Java**

```

1 public class Polygon {
2     private Point[] vertices; // array med hörnpunkter
3     private int n;           // antalet hörnpunkter
4
5     /** Skapar en polygon */
6     public Polygon() {
7         vertices = new Point[1];
8         n = 0;
9     }
10
11     ...

```

**J.1.33 Polygon med primitiv array i Java: stoppa in sist och vid behov skapa mer plats**

Implementera:

```

private void extend() // dubbla storleken
public void addVertex(int x, int y) // lägg till hörnpunkt

```

```

1     private void extend(){
2         Point[] oldVertices = vertices;
3         vertices = new Point[2 * vertices.length]; // skapa dubbel plats
4         for (int i = 0; i < oldVertices.length; i++) { // kopiera
5             vertices[i] = oldVertices[i];
6         }
7     }
8
9     public void addVertex(int x, int y) {
10         if (n == vertices.length) extend();

```

```

11     vertices[n] = new Point(x, y);
12     n++;
13 }

```

### J.1.34 Polygon med primitiv array i Java: stoppa in mitt i på angiven plats

Implementera:

*/\*\* Sätt in hörnpunkt på plats pos \*/*

**public void** insertVertex(**int** pos, **int** x, **int** y)

```

1     public void insertVertex(int pos, int x, int y) {
2         if (n == vertices.length) extend();    // utöka vid behov
3         for (int i = n; i > pos; i--) {        // flytta element bakifrån
4             vertices[i] = vertices[i - 1];
5         }
6         vertices[pos] = new Point(x, y);
7         n++;
8     }

```

### J.1.35 Scanna filer och strängar med java.util.Scanner

- I Scala kan man läsa från fil så här (se quickref sid 3 längst ner):

```
val names = scala.io.Source.fromFile("src/names.txt").getLines.toVector
```

- Klassen java.util.Scanner kan också läsa från fil (se Java Snabbref sid 4):

```

def readFromFile(fileName: String): Vector[String] = {
    val file = new java.io.File(fileName)
    val scan = new java.util.Scanner(file)
    val buffer = scala.collection.mutable.ArrayBuffer.empty[String]
    while (scan.hasNext) {
        buffer += scan.next
    }
    scan.close
    buffer.toVector
}

```

- Med **new** java.util.Scanner(System.in) kan man även scanna tangentbordet.
- Med **new** java.util.Scanner("hej 42") kan man även scanna en sträng.
- Scanna Int och Double med metoderna nextInt och nextDouble.

Se doc: [docs.oracle.com/javase/8/docs/api/java/util/Scanner.html](https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)

### J.1.36 Exempel: Scanner

```
1 scala> val scan = new java.util.Scanner("hej 42 42.0 42 slut")
2
3 scala> scan.hasNext
4 res0: Boolean = true
5
6 scala> scan.hasNextInt
7 res1: Boolean = false
8
9 scala> scan.next
10 res2: String = hej
11
12 scala> scan.hasNextInt
13 res3: Boolean = true
14
15 scala> scan.nextInt
16 res4: Int = 42
17
18 scala> while (scan.hasNext) println(scan.next)
19 42.0
20 42
21 slut
```

---

### J.1.37 Använda Java-samlingar i Scala med CollectionConverters

Med hjälp av **import** `scala.jdk.CollectionConverters.*` får du smidig **interoperabilitet** med Java och dess standardbibliotek, speciellt metoderna **asJava** och **asScala**:

```
1 scala> import scala.jdk.CollectionConverters.*
2
3 scala> Vector(1,2,3).asJava
4 res0: java.util.List[Int] = [1, 2, 3]
5
6 scala> val xs = new java.util.ArrayList[String]()
7 xs: java.util.ArrayList[String] = []
8
9 scala> xs.add("hej")
10 res1: Boolean = true
11
12 scala> xs.asScala
13 res2: scala.collection.mutable.Buffer[String] = Buffer(hej)
```

Läs mer här:

<https://docs.scala-lang.org/overviews/collections-2.13/conversions-between-java-and-scala-collections.html>

---

### J.1.38 Generiska samlingar i Java

- Från och med version 5 av Java (2004) så introducerades **generics** vilket möjliggör skapandet av klasser som kan erbjuda generell behandling av olika typer av objekt.
- Generiska klasser i Java känns igen med syntaxen `Klassnamn<Typ>`, till exempel `ArrayList<Point>`
- Fördjupning: [docs.oracle.com/javase/tutorial/extra/generics/intro.html](https://docs.oracle.com/javase/tutorial/extra/generics/intro.html), mer om detta i fördjupningskursen.

---

### J.1.39 Om ArrayList i Java

`java.util.ArrayList` liknar `scala.collection.mutable.ArrayBuffer` som båda har dessa fördelar:

- Lagrar sina element internt i snabbindexerade primitiva arrayer.
- Fungerar för alla typer av objekt.
- Utökar samlingens storlek av sig själv vid behov.

Det finns dock vissa nackdelar med `ArrayList` i Java (som inte gäller för `ArrayBuffer` i Scala):

- Fungerar **inte** rakt av med primitiva typer `int`, `double`, `char`, ... (men det finns sätt komma runt detta, tack vare s.k. wrapper-klasser och autoboxning; mer om detta snart)
- Namnet `ArrayList` är inte helt lyckat, eftersom ordet "lista" normalt används för länkade snarare än array-liknande strukturer.

---

### J.1.40 Polygon med ArrayList i Java

Klassen `Polygon`, nu med ett attribut av typen `ArrayList<Point>`:

```
public class Polygon {
    private ArrayList<Point> vertices; // lista med hörnpunkter

    /** Skapar en polygon */
    public Polygon() {
        vertices = new ArrayList<Point>();
    }

    ...
}
```

Det behövs inget attribut `n` eftersom vi inte själva behöver hålla reda på antalet allokerade platser: allokering, insättning, och utökning av antalet platser sköts helt automatiskt av `ArrayList`-klassen vid behov.

---

### J.1.41 Några viktiga operationer på ArrayList<E>

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html>

```
/** Tar reda på elementet på plats pos */
E get(int pos);

/** Läger in objektet obj sist */
void add(E obj);
```

```
/** Lägger in obj på plats pos; efterföljande flyttas */  
void add(int pos, E obj);  
  
/** Tar bort elementet på plats pos och returnerar det */  
E remove(int pos);  
  
/** Tar reda på antalet element i listan */  
int size();
```

Lär dig vad som finns om ArrayList i snabbreferensen för Java

Överkurs för den nyfikne: kolla implementation av ArrayList här:

<http://www.docjar.com/html/api/java/util/ArrayList.java.html>

### J.1.42 Övning ArrayList: new och add

Skriv Java-kod som skapar en lista med element av typen Point och lägger in tre punkter i listan med koordinaterna:

(50, 50), (50,10) och (30, 40).

Lösning:

```
ArrayList<Point> vertices = new ArrayList<Point>();  
vertices.add(new Point(50, 50));  
vertices.add(new Point(50, 10));  
vertices.add(new Point(30, 40));
```

### J.1.43 For-each-sats i Java:

- Antag att vi vill gå igenom alla element i en lista.

```
ArrayList<String> words = new ArrayList<String>();
```

- Det finns två olika typer av **for**-satser i Java som kan göra detta:
  - Vanlig **for**-sats:

```
for (int i = 0; i < words.size(); i++) {  
    System.out.println(i + ": " + words.get(i));  
}
```

- Så kallad **for-each-sats** med denna syntax:

```
for (Elementtyp element: samling) { ... }
```

Exempel:

```
for (String s: words) {  
    System.out.println(s);  
}
```

Men vi får ingen indexvariabel då...

---

#### J.1.44 Polygon med ArrayList: metoderna blir enklare

```
public void addVertex(int x, int y) {
    vertices.add(new Point(x, y));
}

public void move(int dx, int dy) {
    for (Point p: vertices){
        p.move(dx, dy);
    }
}

public void insertVertex(int pos, int x, int y) {
    vertices.add(pos, new Point(x, y));
}

public void removeVertex(int pos) {
    vertices.remove(pos);
}
```

Se hela lösningen här: [compendium/examples/scalajava/list/Polygon.java](https://compendium/examples/scalajava/list/Polygon.java)

---

#### J.1.45 Polygon med ArrayList: iterera över alla hörnpunkter i draw med indexering

```
public void draw(SimpleWindow w) {
    if (vertices.size() == 0) {
        return;
    }
    Point start = vertices.get(0);
    w.moveTo(start.getX(), start.getY());
    for (int i = 1; i < vertices.size(); i++) {
        w.lineTo(vertices.get(i).getX(),
                 vertices.get(i).getY());
    }
    w.lineTo(start.getX(), start.getY());
}
```

Övning: Skriv om med for-each-sats.

---

#### J.1.46 Polygon med ArrayList: iterera över alla hörnpunkter i draw med foreach-sats

```
public void draw(SimpleWindow w) {  
    if (vertices.size() == 0) {  
        return;  
    }  
    Point start = vertices.get(0);  
    w.moveTo(start.getX(), start.getY());  
    for (Point p: vertices){  
        w.lineTo(p.getX(), p.getY());  
    }  
    w.lineTo(start.getX(), start.getY());  
}
```

Se hela lösningen här: [compendium/examples/scalajava/list/Polygon.java](https://compendium/examples/scalajava/list/Polygon.java)

---

### J.1.47 Övning ArrayList: implementera metoden hasVertex

Skriv kod som implementerar denna metod i klassen Polygon:

```
/** Undersöker om polygonen har någon hörnpunkt med koordinaterna x, y. */  
public boolean hasVertex(int x, int y) {  
    ???  
}
```

### J.1.48 Lösning ArrayList: implementera metoden hasVertex

```
public boolean hasVertex(int x, int y) {  
    for (Point p: vertices) {  
        if (p.getX() == x && p.getY() == y) {  
            return true;  
        }  
    }  
    return false;  
}
```

### J.1.49 For-each-sats med array

For-each-sats fungerar även med primitiv array:

```
String[] stringArray = {"hej", "på", "dej"};  
for (String s: stringArray) {  
    System.out.println(s);  
}
```

### J.1.50 Generiska klasser (t.ex. ArrayList) med primitiva typer

Detta går tyvärr **INTE** i Java:

```
ArrayList<int> list = new ArrayList<int>();
```

- Hur gör man om man vill ha heltalselement (eller andra primitiva värden) i en generisk samling?
- Javas lösning på problemet består av två delar:
  - Klasser som packar in primitiva typer, (eng. *wrapper classes*)
  - Speciella regler för implicita konverteringar, s.k. "auto-boxing" (eng. *Boxing / Unboxing conversions*)

Ofta fungerar det fint, men det finns fallgropar.

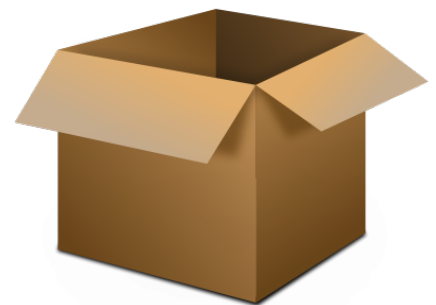
(Om du är nyfiken på alla intrikata detaljer, se [Java tutorial](#) och [Javaspecifikationen](#).)

### J.1.51 Wrapper-klassen Integer

En skiss av klassen Integer

(ligger i paketet `java.lang` och importeras därmed implicit):

```
public class Integer {  
    private int value;  
  
    public static final MIN_VALUE = -2147483648;  
    public static final MAX_VALUE = 2147483647;  
  
    public Integer(int value) {  
        this.value = value;  
    }  
  
    public int intValue() {  
        return value;  
    }  
    ...  
}
```



Javadoc för klasen Integer finns här:

<http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html>



### J.1.52 Wrapper-klasser i java.lang

Primitiv typ	Inpackad typ
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

### J.1.53 Övning: primitiva versus inpackade typer

Med papper och penna:

- Deklarera en variabel med namnet gurka av den primitiva heltalstypen och initiera den till värdet 42.
- Deklarera en referensvariabel med namnet tomat av den inpackade ("wrappade") heltalstypen och initiera den till värdet 43.
- Rita hur det ser ut i minnet.

### J.1.54 Exempel: Lista med heltal utan autoboxning

```
import java.util.ArrayList;
import java.util.Scanner;

public class TestIntegerList {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv heltal med blank emellan. Avsluta med <CTRL+D>");
        while (scan.hasNextInt()) {
            int nbr = scan.nextInt();
            Integer obj = new Integer(nbr);
            list.add(obj);
        }
        System.out.println("Dina heltal i omvänd ordning:");
        for (int i = list.size() - 1; i >= 0; i--) {
            Integer obj = list.get(i);
            int nbr = obj.intValue();
            System.out.println(nbr);
        }
    }
}
```

```
    }
}
```

Koden finns här: [compendium/examples/scalajava/TestIntegerList.java](#)

---

### J.1.55 Specialregler för wrapper-klasser

- Om ett int-värde förekommer där det behövs ett Integer-objekt, så lägger kompilatorn **automatiskt** ut kod som skapar ett Integer-objekt som packar in värdet.
- Om ett Integer-objekt förekommer där det behövs ett int-värde, lägger kompilatorn **automatiskt** ut kod som anropar metoden intValue().

Samma gäller mellan alla primitiva typer och dess wrapper-klasser:

boolean ⇔ Boolean

byte ⇔ Byte

short ⇔ Short

char ⇔ Character

int ⇔ Integer

long ⇔ Long

float ⇔ Float

double ⇔ Double

---

### J.1.56 Exempel: Lista med heltal och autoboxning

```
import java.util.ArrayList;
import java.util.Scanner;

public class TestIntegerListAutoboxing {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv heltal med blank emellan. Avsluta med <CTRL+D>");
        while (scan.hasNextInt()) {
            int nbr = scan.nextInt();
            list.add(nbr); // motsvarar: list.add(new Integer(nbr));
        }
        System.out.println("Dina heltal i omvänd ordning:");
        for (int i = list.size() - 1; i >= 0; i--) {
            int nbr = list.get(i); // motsvarar: int nbr = list.get(i).intValue();
            System.out.println(nbr);
        }
    }
}
```

Koden finns här: [scalajava/generics/TestIntegerListAutoboxing.java](#)

---

### J.1.57 Fallgropar vid autoboxning

- Jämförelser med `==` och `!=`  
[compendium/examples/scalajava/generics/TestPitfall1.java](#)
- Kompilatorn hittar inte förväxlad parameterordning, t.ex. `add(pos, item)` i fel ordning:  
`add(item, pos)`  
[compendium/examples/scalajava/generics/TestPitfall2.java](#)

### J.1.58 Referenslikhet eller innehållslikhet i Scala och Java

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturlikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.
- I Scala finns flera metoder som testar likhet:
  - metoden `eq` testar **referenslikhet** och `r1.eq(r2)` ger **true** om `r1` och `r2` refererar till **samma** instans.
  - metoden `ne` testar referensolikhet och `r1.ne(r2)` ger **true** om `r1` och `r2` refererar till **olika** instanser.
  - metoden `==` som anropar metoden `equals` som default testar referenslikhet men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturlikhet.
- Scalas **standardbibliotek** och **grundtyperna** `Int`, `String` etc. testar **innehållslikhet** genom metoden `==`
- I **Java** är det **annorlunda**: symbolen `==` är ingen metod i Java utan **specialsyntax** som vid instansjämförelse alltid testar **referenslikhet**, medan metoden `equals` kan överskuggas med valfri likhetstest.

### J.1.59 Fallgrop med samlingar: metoden `contains` kräver implementation av `equals`

Antag att vi vill implementera `hasVertex()` i klassen `Polygon` genom att använda metoden `contains` på en lista. Hur gör vi då?

```
public boolean hasVertex(int x, int y) {
    return vertices.contains(new Point(x, y)); // FUNKAR INTE om ...
    // ... inte Point har en equals som kollar innehållslikhet
}
```

Vi behöver implementera metoden `equals(Object obj)` i klassen `Point` som kollar innehållslikhet och ersätter den `equals` som finns i `Object` som kollar referenslikhet, eftersom metoden `contains` i klassen `ArrayList` anropar `equals` när den letar igenom listan efter lika objekt.

Se exempel här: [compendium/examples/scalajava/generics/TestPitfall3.java](http://compendium/examples/scalajava/generics/TestPitfall3.java)

Det krävs ofta även att man även ersätter `hashCode`, mer om det i fortsättningskursen.

---

### J.1.60 Fullständigt recept för `equals`

För den nyfikne inför fortsättningskursen efter jul:

Läs om fallgropar för att implementera `equals` i **Java** här:

[www.artima.com/lejava/articles/equality.html](http://www.artima.com/lejava/articles/equality.html)

Läs receptet för att implementera `equals` i **Scala** här:

[www.artima.com/pins1ed/object-equality.html#28.4](http://www.artima.com/pins1ed/object-equality.html#28.4)

---

### J.1.61 Villkorsuttryck i Java

Det går att använda villkorsuttryck i Java, men med syntax från språket C:

#### Scala

```
var r = math.random()  
var answer = if r > 0.5 then 42 else 0
```

#### Java

```
double r = Math.random();  
int answer = (r > 0.5) ? 42 : 0;
```

---

### J.1.62 Typtest och typkonvertering

#### Scala

```
var x = "hej"  
  
var isString = x.isInstanceOf[String]  
  
var y = 42  
  
var z = y.asInstanceOf[Double]
```

#### Java

```
String x = "hej";  
  
boolean isString = x instanceof String;  
  
int y = 42;  
  
double z = (double) y;
```

Detta görs ju i Scala bäst med **match** och typmönster!

---

### J.1.63 Regler för överskuggning i Java

<http://docs.oracle.com/javase/tutorial/java/IandI/override.html>

---

### J.1.64 Fånga undantag i Scala och Java

Typisk skillnad mellan Scala och Java:

konstruktioner som är **uttryck** i Scala är ofta **satser** i Java.

**Scala**

```
val a = try 2 / 0 catch
  case e: ArithmeticException => 0
```

```
val b = try 4 / 2 catch
  case e: ArithmeticException => 0
```

**Java**

```
int a;
try {
    a = 2 / 0;
} catch (ArithmeticException e) {
    a = 0;
}
```

```
int b;
try {
    b = 4 / 2;
} catch (ArithmeticException e) {
    b = 0;
}
```

**J.1.65 Gränssnittet List i Java**

- I Java finns inte **trait** och inmixning.
- I stället finns **interface** som liknar **trait** men är mer begränsad vad gäller vilka medlemmar som får finnas.
- Man kan bara göra **extends** på exakt en annan klass, men man kan i Java göra **implements** på flera **interface**.
- Exempel:

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

- Att implementera ett gränssnitt innebär att uppfylla ett kontrakt som utlovar att vissa speciella metoder finns tillgängliga.
- Gränssnittet List uppfylls av en av dess implementationer ArrayList

på liknande sätt i Scala där gränssnittet Seq uppfylls av Vector etc.

```
List<String> xs = new ArrayList<String>();
```

- Liknande exempel från övning Hangman:
 

```
Set<Character> found = new HashSet<Character>();
```
- En Scala-trait med enbart abstrakta medlemmar kompileras till ett Java-interface i JVM bytekode.
- Mer om gränssnitt i Java i fördjupningskursen.

**J.1.66 Det går inte att skapa generisk Array i Java**

- I Java kan man **inte** skapa en primitiv array av godtycklig typ enligt generisk typparameter: `T[] xs = new T[42]`



```
        int eqTest3 =  
            (new String("hej")).compareTo(new String("hej"));  
        System.out.println(eqTest1);  
        System.out.println(eqTest2);  
        System.out.println(eqTest3);  
    }  
}
```

```
1 $ javac StringEqTest.java  
2 $ java StringEqTest  
3 false  
4 true  
5 0
```

---

## J.2 Övning java

### Mål

- ☐ Kunna förklara och beskriva viktiga skillnader mellan Scala och Java.
- ☐ Kunna översätta enkla algoritmer, klasser och singletonobjekt från Scala till Java och vice versa.
- ☐ Känna till vad en case-klass innehåller i termer av en Javaklass.
- ☐ Kunna använda Javatyperna List, ArrayList, Set, HashSet och översätta till deras Scalamotsvarigheter med CollectionConverters.
- ☐ Kunna förklara hur autoboxning fungerar i Java, samt beskriva fördelar och fallgropar.

### Förberedelser

- ☐ Studera teori i början av detta Appendix.

### J.2.1 Grunduppgifter; förberedelse inför laboration

**Uppgift 1.** *Översätta metoder från Java till Scala.* I denna uppgift ska du översätta en Java-klass som används som en modul<sup>2</sup> och bara innehåller statistiska metoder och inget förändringsbart tillstånd som kan ändras utifrån. (I nästa uppgift ska du sedan översätta klasser med förändringsbara tillstånd.)

Vi börjar med att göra översättningen från Java till Scala rad för rad och du ska behålla så mycket som möjligt av syntax och semantik så att Scala-koden blir så Java-lik som möjligt. I efterföljande deluppgift ska du sedan omforma översättningen så att Scala-koden blir mer idiomatisk<sup>3</sup>.

a) Studera klassen Hangman nedan. Du ska översätta den från Java till Scala enligt de riktlinjer och tips som följer efter koden. Läs igenom alla riktlinjer och tips innan du börjar.

```

1  import java.net.URL;
2  import java.util.ArrayList;
3  import java.util.Set;
4  import java.util.HashSet;
5  import java.util.Scanner;
6
7  public class Hangman {
8      private static String[] hangman = new String[]{
9          " ===== ",
10         " |/   | ",
11         " |    0  ",
12         " |   -|-  ",
13         " |   / \\  ",
14         " |       ",
15         " |       ",
16         " ===== RIP  :("};
17
18     private static String renderHangman(int n){
19         StringBuilder result = new StringBuilder();

```

<sup>2</sup>[en.wikipedia.org/wiki/Modular\\_programming](https://en.wikipedia.org/wiki/Modular_programming)

<sup>3</sup>[sv.wikipedia.org/wiki/Idiom\\_%28programmering%29](https://sv.wikipedia.org/wiki/Idiom_%28programmering%29)



```
20     for (int i = 0; i < n; i++){
21         result.append(hangman[i]);
22         if (i < n - 1) {
23             result.append("\n");
24         }
25     }
26     return result.toString();
27 }
28
29 private static String hideSecret(String secret,
30                                 Set<Character> found){
31     String result = "";
32     for (int i = 0; i < secret.length(); i++) {
33         if (found.contains(secret.charAt(i))) {
34             result += secret.charAt(i);
35         } else {
36             result += '_';
37         }
38     }
39     return result;
40 }
41
42 private static boolean foundAll(String secret,
43                                 Set<Character> found){
44     boolean foundMissing = false;
45     int i = 0;
46     while (i < secret.length() && !foundMissing) {
47         foundMissing = !found.contains(secret.charAt(i));
48         i++;
49     }
50     return !foundMissing;
51 }
52
53 private static char makeGuess(){
54     Scanner scan = new Scanner(System.in);
55     String guess = "";
56     do {
57         System.out.println("Gissa ett tecken: ");
58         guess = scan.next();
59     } while (guess.length() != 1);
60     return Character.toLowerCase(guess.charAt(0));
61 }
62
63 public static String download(String address, String coding){
64     String result = "lackalänga";
65     try {
66         URL url = new URL(address);
67         ArrayList<String> words = new ArrayList<String>();
```

```
68         Scanner scan = new Scanner(url.openStream(), coding);
69         while (scan.hasNext()) {
70             words.add(scan.next());
71         }
72         int rnd = (int) (Math.random() * words.size());
73         result = words.get(rnd);
74     } catch (Exception e) {
75         System.out.println("Error: " + e);
76     }
77     return result;
78 }
79
80 public static void play(String secret){
81     Set<Character> found = new HashSet<Character>();
82     int bad = 0;
83     boolean won = false;
84     while (bad < hangman.length && !won){
85         System.out.print("\nFelgissningar: " + bad + "\t");
86         System.out.println(hideSecret(secret, found));
87         char guess = makeGuess();
88         if (secret.indexOf(guess) >= 0) {
89             found.add(guess);
90         } else {
91             bad++;
92             System.out.println(renderHangman(bad));
93         }
94         won = foundAll(secret, found);
95     }
96     if (won) {
97         System.out.println("BRA! :)");
98     } else {
99         System.out.println("Hängd! :(");
100     }
101     System.out.println("Rätt svar: " + secret);
102     System.out.println("Antal felgissningar: " + bad);
103 }
104
105 public static void main(String[] args){
106     if (args.length == 0) {
107         String runeberg =
108             "http://runeberg.org/words/ord.ortsnamn.posten";
109         play(download(runeberg, "ISO-8859-1"));
110     } else {
111         int rnd = (int) (Math.random() * args.length);
112         play(args[rnd]);
113     }
114 }
115 }
```

*Riktlinjer och tips för översättningen:*

1. Skriv Scala-koden med en texteditor i en fil som heter `hangman1.scala` och kompilera med `scalac hangman1.scala` i terminalen; använd alltså *inte* en IDE, så som Eclipse eller IntelliJ, utan en "vanlig" texteditor, t.ex. VS code.
2. Översätt i denna första deluppgift rad för rad så likt den ursprungliga Java-kodens utseende (syntax) som möjligt, med så få ändringar som möjligt. Du ska alltså ha kvar dessa Scalaovanligheter, även om det inte alls blir som man brukar skriva i Scala:
  - (a) långa indrag,
  - (b) onödiga semikolon,
  - (c) onödiga `()`,
  - (d) onödiga `{}`,
  - (e) onödiga `System.out`, och
  - (f) onödiga **return**.
3. Försök också i denna deluppgift göra så att betydelsen (semantiken) så långt som möjligt motsvarar den i Java, t.ex. genom att använda **var** överallt, även där man i Scala normalt använder **val**.
4. En Javaklass med bara statiska medlemmar motsvarar ett singletonobjekt i Scala, alltså en **object**-deklaration innehållande "vanliga" medlemmar.
5. För att tydliggöra att du använder Javas Set och HashSet i din Scala-kod, använd följande import-satser i `hangman1.scala`, som därmed döper om dina importerade namn och gör så att de inte krockar med Scalas inbyggda Set. Denna form av import går inte att göra i Java.

```
import java.util.{Set => JSet};
import java.util.{HashSet => JHashSet};
```

6. Javas `i++` fungerar inte i Scala; man får istället skriva `i += 1` eller mindre vanliga `i = i + 1`.
  7. Typparametrar i Java skrivs inom `<>` medan Scalas syntax för typparametrar använder `[]`.
  8. Till skillnad från Java så har Scalas metoddeklarationer ett tilldelningstecken `=` efter returtypen, före kroppen.
  9. Du kan ladda ner Java-koden till Hangman-klassen nedan från kursens repo<sup>4</sup>. I samma bibliotek ligger även lösningarna till översättningen i Scala, men kolla *inte* på dessa förrän du gjort klart översättningarna och fått dem att kompilera och köra felfritt! Tanken är att du ska träna på att läsa felmeddelande från kompilatorn och åtgärda dem i en upprepad kompilera-testa-rätta-cykel.
- b) Skapa en ny fil `hangman2.scala` som till att börja med innehåller en kopia av din direkt-översatta Java-kod från föregående deluppgift. Omforma koden så att den blir mer som man brukar skriva i Scala, alltså mer Scala-idiomatisk. Försök förenkla och förkorta så mycket du kan utan att göra avkall på läsbarheten.

*Tips och riktlinjer:*

1. Kalla Scala-objektet för hangman. När man använder ett Scalaobjekt som en modul (alltså en samling funktioner i en gemensam, avgränsad namnrymd) har man gärna liten begynnelsebokstav, i likhet med konventionen för paketnamn. Ett paket är ju också en slags modul och med en namngivningskonvention som är gemensam kan man

<sup>4</sup>[github.com/lunduniversity/introprog/blob/master/compendium/examples/scalajava/Hangman.java](https://github.com/lunduniversity/introprog/blob/master/compendium/examples/scalajava/Hangman.java)

senare, utan att behöva ändra koden som använder modulen, ändra från ett singelobjekt till ett paket och vice versa om man så önskar.

2. Gör alla metoder publikt tillgängliga och låt även strängvektorn `hangman` vara publikt tillgänglig. Deklarera `hangman` som en **val** och konstruera den med `Vector`. Eftersom `Vector` är oföränderlig och man inte kan ärva från singelobjekt och `hangman` är deklarerad med **val** finns inga speciella risker med att göra den konstanta vektorn publik om vi inte har något emot att annan kod kan läsa (och eventuellt göra sig beroende av) vår hänggubbetext.
3. I metoden `renderHangman`, använd `take` och `mkString`.
4. I metoden `hideSecret`, använd `map` i stället för en **for**-sats.
5. Det går att ersätta metoden `foundAll` med det kärnfulla uttrycket `(secret forall found)` där `secret` är en sträng och `found` är en mängd av tecken (undersök gärna i REPL hur detta fungerar). Skippa därför den metoden helt och använd det kortare uttrycket direkt.
6. I metoden `makeGuess`, i stället för `Scanner`, använd `scala.io.StdIn.readLine`.
7. Om du vill träna på att använda rekursion i stället för imperativa loopar: Gör metoden `makeGuess` rekursiv i stället för att använda **do-while**.
8. I metoden `download`, i stället för `java.net.URL` och `java.util.ArrayList`, använd `scala.io.Source.fromURL(address, coding).getLines.toVector` och gör en lokal import av `scala.io.Source.fromURL` överst i det block där den används. Det går inte att ha lokala **import**-satser i Java.
9. Låt metoden `download` returnera en `Option[String]` som i fallet att nedladdningen misslyckas returnerar `None`.
10. I metoden `download`, i stället för **try-catch** använd `scala.util.Try` och dess smidiga metod `toOption`.
11. Om du vill träna på att använda rekursion i stället för imperativa loopar: Använd, i stället för **while**-satsen i metoden `play`, en lokal rekursiv funktion med denna signatur:

```
def loop(found: Set[Char], bad: Int): (Int, Boolean)
```

Funktionen `loop` returnerar en 2-tupel med antalet felgissningar och **true** om man hittat alla bokstäver eller **false** om man blev hängd.

**Uppgift 2.** Översätta mellan klasser i Scala och klasser i Java. Klassen `Point` nedan är en modell av en punkt som kan sparas på begäran i en lista. Listan är privat för kompanjonsobjektet och kan skrivas ut med en metod `showSaved`. I koden används en `ArrayBuffer`, men i framtiden vill man, vid behov, kunna ändra från `ArrayBuffer` till en annan sekvenssamlingsimplementation, t.ex. `ListBuffer`, som uppfyller egenskaperna hos supertypen `Buffer`, men har andra prestandaegenskaper för olika operationer. Därför är attributet `saved` i kompanjonsobjektet deklarerat med den mer generella typen.

```
1 class Point(val x: Int, val y: Int, save: Boolean = false):
2   import Point.*
3
4   if save then saved.prepend(this)
5
6   def this() = this(0, 0)
7
8   def distanceTo(that: Point) = distanceBetween(this, that)
9
```

```

10  override def toString = s"Point($x, $y)"
11
12  object Point:
13    import scala.collection.mutable.{ArrayBuffer, Buffer}
14
15    private val saved: Buffer[Point] = ArrayBuffer.empty
16
17    def distanceBetween(p1: Point, p2: Point) =
18      math.hypot(p1.x - p2.x, p1.y - p2.y)
19
20    def showSaved: Unit =
21      println(saved.mkString("Saved: ", ", ", "\n"))

```

a) Översätt klassen Point ovan från Scala till Java. Vi ska i nästa deluppgift kompilera både Scala-programmet ovan och ditt motsvarande Java-program i terminalen och testa i REPL att klasserna har motsvarande funktionalitet.

*Tips och riktlinjer:*

1. För att namnen inte ska krocka i våra kommande tester, kalla Javatypen för JPoint.
2. I stället för Scalas ArrayBuffer och Buffer, använd Javas ArrayList och List som båda ligger i paketet java.util.
3. Undersök dokumentationen för java.util.List för att hitta en motsvarighet till prepend för att lägga till i början av listan.
4. I stället för default-argumentet i Scalas primärkonstruktor, använd en extra Java-konstruktor.
5. Det finns inga singelobjekt och inga kompanjonsobjekt i Java; istället kan man använda statiska klassmedlemmar. Placera kompanjonsobjektets medlemmars motsvarigheter *inuti* Java-klassen och gör dem till **static**-medlemmar.
6. Kod i klasskroppen i Scalaklassen, så som if-satsen på rad 4, placeras i lämplig konstruktor i Javaklassen.
7. Utskrifter med print och println behöver i Java föregås av System.out.
8. Det finns inget nyckelord **override** i Java, men en s.k. annotering som ger samma kompilatorhjälp. Den skrivs med ett snabel-a och stor begynnelsebokstav, så här: **@Override** före metoddeklarationen.
9. I Java används konventionen att börja getter-metoder med ordet get, t.ex. getX().
10. Det finns ingen motsvarighet till mkString för List så du behöver själv gå igenom listan och hämta elementreferenser för utskrift med en **for**-loop. Notera att efter sista elementet ska radbrytning göras i utskriften och att inget komma ska skrivas ut efter sista elementet.
11. I Java behövs en ny **import**-deklaration om man vill importera ännu en typ från samma paket. Man kan även i Java använda asterisk \*, (motsvarande \_ i Scala), för att importera allt i ett paket, men då får man med alla möjliga namn och det vill man kanske inte.
12. Metoder i Java slutar med () om de saknar parametrar.
13. Alla satser i Java slutar med lättglömda semikolon. (Efter att man i skrivit mycket Javakod och växlar till Scalakod är det svårt att vänja sig av med att skriva semikolon...)

b) Starta REPL i samma bibliotek som du kompilerat kodfilerna. Testa så att klasserna Point och JPoint beter sig på samma vis enligt nedan. Skriv även testkod i REPL för att avläsa de attributvärden som har getters och undersök att allt funkar som det ska.

```

> scalac Point.scala
> javac JPoint.java
> scala
scala> val (p, jp) = (new Point, new JPoint)
scala> p.distanceTo(new Point(3, 4))
scala> Point.showSaved
scala> jp.distanceTo(new JPoint(3, 4))
scala> JPoint.showSaved
scala> for (i <- 1 to 10) { new Point(i, i, true) }
scala> Point.showSaved
scala> for (i <- 1 to 10) { new JPoint(i, i, true) }
scala> JPoint.showSaved

```

c) Översätt nedan Javaklass JPerson till en **case class** Person i Scala med motsvarande funktionalitet.

```

1  public class JPerson {
2      private final String name;
3      private final int age;
4
5      public JPerson(final String name, final int age){
6          this.name = name;
7          this.age = age;
8      }
9
10     public JPerson(final String name){
11         this(name, 0);
12     }
13
14     public String getName() {
15         return name;
16     }
17
18     public int getAge() {
19         return age;
20     }
21
22     public boolean canEqual(Object other) {
23         return (other instanceof JPerson);
24     }
25
26     @Override public boolean equals(Object other){
27         boolean result = false;
28         if (other instanceof JPerson) {
29             JPerson that = (JPerson) other;
30             result = that.canEqual(this) &&
31                 this.getName() == that.getName() &&
32                 this.getAge() == that.getAge();
33         }
34         return result;
35     }

```

```

36
37     @Override public int hashCode() {
38         return name.hashCode() * 41 + age;
39     }
40
41     @Override public String toString() {
42         return "JPerson(" + name + ", " + age + ")";
43     }
44 }

```



d) Undersök i REPL vilken funktionalitet i Scala-case-klassen `Person` som *inte* är implementerad i Java-klassen `JPerson` ovan. Skriv upp namnen på några av case-klassens extra metoder samt deras signatur genom att för en `Person`-instans, och för kompanjonsobjektet `Person`, trycka på TAB-tangenten. Prova några av de extra metoderna i REPL och förklara vad de gör.

```

1 scala> val p = Person("Björn", 49)
2 scala> p.          // tryck TAB en gång
3 scala> Person.    // tryck TAB en gång
4 scala> p.copy     // tryck TAB en gång
5 scala> p.copy()
6 scala> p.copy(age = p.age + 1)
7 scala> Person.unapply(p)

```

**Uppgift 3.** *Oföränderlig Java-klass.* Översätt nedan Scala-klass till Java-klassen `JPoint3D`. Alla attribut ska vara privata (varför?). Översätt defaultargumentet till en alternativ konstruktor. Kalla getters för t.ex. `getX()`. Kör `javac` och testa i REPL.

```
class Point3D(val x: Int, val y: Int, val z: Int = 0)
```

**Uppgift 4.** *Förändringsbar Java-klass.*

Översätt nedan Scala-klass till Java-klassen `JMutablePoint3D`. Alla attribut ska vara privata (varför?). Översätt defaultargumentet till en alternativ konstruktor. Kalla setters för t.ex. `setX`. Kör `javac` och testa i REPL.

```
class MutablePoint3D(var x: Int, var y: Int, var z: Int = 0)
```

**Uppgift 5.** *Jämföra strängar i Java.* I Java kan man **inte** jämföra strängar med operatorerna `<`, `<=`, `>`, och `>=`. Dessutom ger operatorerna `==` och `!=` inte innehålls(o)likhet utan referens(o)likhet. Istället får man använda metoderna `equals` och `compareTo`, vilka också fungerar i Scala eftersom strängar i Scala och Java är av samma typ, nämligen `java.lang.String`.

a) Vad ger följande uttryck för värde?

```

1 scala> "hej".getClass.getTypeName
2 scala> "hej".equals("hej")
3 scala> "hej".compareTo("hej")

```

b) Studera dokumentationen för metoden `compareTo` i `java.lang.String`<sup>5</sup> och skriv minst 3 olika uttryck i Scala REPL som testar hur metoden fungerar i olika fall.

<sup>5</sup>[docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareTo-java.lang.String-](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareTo-java.lang.String-)



- c) Studera dokumentationen `compareToIgnoreCase`<sup>6</sup> och skriv minst 3 olika stränguttryck i Scala REPL som testar hur metoden fungerar i olika fall.
- d) Vad skriver följande Java-program ut?

```
public class StringEqTest {
    public static void main(String[] args){
        boolean eqTest1 =
            (new String("hej")) == (new String("hej"));
        boolean eqTest2 =
            (new String("hej")).equals(new String("hej"));
        int eqTest3 =
            (new String("hej")).compareTo(new String("hej"));
        System.out.println(eqTest1);
        System.out.println(eqTest2);
        System.out.println(eqTest3);
    }
}
```

**Uppgift 6.** *Linjärsökning i Java.* Denna uppgift bygger vidare på uppgift 13 i kapitel 8. Du ska göra en variant på linjärsökning som innebär att leta upp första yatzy-raden i en matris där varje rad innehåller utfallet av 5 tärningskast.

- a) Du ska lägga till metoderna `isYatzy` och `findFirstYatzyRow` i klassen `ArrayMatrix` i uppgift 13 i kapitel 8 enligt nedan skiss. Vi börjar med metoden `isYatzy` i denna deluppgift (nästa deluppgift handlar om `findFirstYatzyRow`). OBS! Det finns en bugg i `isYatzy` – rätta buggen och testa så att den fungerar.

```
public static boolean isYatzy(int[] dice){ /* has one bug! */
    int col = 1;
    boolean allSimilar = true;
    while (col < dice.length && allSimilar) {
        allSimilar = dice[0] == dice[col];
    }
    return allSimilar;
}

/** Finds first yatzy row in m; returns -1 if not found */
public static int findFirstYatzyRow(int[][] m){
    int row = 0;
    int result = -1;
    while (???) {
        /* linear search */
    }
    return result;
}
```

- b) Implementera `findFirstYatzyRow`. Skapa först pseudo-kod för linjärsökningsalgoritmen innan du skriver implementationen i Java. Testa ditt program genom att lägga till

<sup>6</sup>[docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareToIgnoreCase-java.lang.String-](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareToIgnoreCase-java.lang.String-)



följande rader i huvudprogrammet. Metoden `fillRnd` ingår i uppgift 13 i kapitel 8.

```
int[][] yss = new int[2500][5];
fillRnd(yss, 6);
int i = findFirstYatzyRow(yss);
System.out.println("First Yatzy Index: " + i);
```

**Uppgift 7. Jämförelsestöd i Java.** Java har motsvarigheter till Scalas `Ordering` och `Ordered`, som heter `java.util.Comparator` och `java.lang.Comparable`. I själva verket så är Scalas `Ordering` en subtyp till Javas `Comparator`, medan Scalas `Ordered` är en subtyp till Javas `Comparable`.

- Javas `Comparator` och Scalas `Ordering` används för att skapa fristående ordningar som kan jämföra *två olika* objekt. I Scala kan dessa göras implicit tillgängliga. I Javas samlingsbibliotek skickas instanser av `Comparator` med som explicita argument.
- Javas `Comparable` och Scalas `Ordered` används som supertyp för klasser som vill kunna jämföra "sig själv" med andra objekt och har *en* naturlig ordningsdefinition.



- a) Sök upp dokumentationen för `java.util.Comparator`. Vilken abstrakt metod måste implementeras och vad gör den?
- b) I paketet `java.util.Arrays` finns en metod `sort` som tar en `Array[T]` och en `Comparable[T]`. Testa att använda dessa i REPL enligt nedan skiss. Starta om REPL så att ev. tidigare implicita ordningar för `Team` inte finns kvar.

```
1 scala> import java.util.Comparator
2 scala> val teamComparator = new Comparator[Team]{
3     def compare(o1: Team, o2: Team) = ???
4 }
5 scala> val xs =
6     Array(Team("fnatic", 1499), Team("nip", 1473), Team("lumi", 1601))
7 scala> java.util.Arrays.sort(xs.toArray, teamComparator)
8 scala> xs
```

- c) I Scala finns en behändig metod `Ordering.comparatorToOrdering` som skapar en implicit tillgänglig ordning om man har en `java.util.Comparator`. Testa detta enligt nedan i REPL, med deklarationerna från föregående deluppgift.

```
1 scala> implicit val teamOrd = Ordering.comparatorToOrdering(teamComparator)
2 scala> xs.sorted
```



- d) Sök upp dokumentationen för `java.lang.Comparable`. Vilken abstrakt metod måste implementeras och vad gör den?
- e) Gör så att klassen `Point` är `Comparable` och att punkter närmare origo sorteras före punkter som är längre ifrån origo enligt nedan skiss. I Scala är typer som är `Comparable` implicit även `Ordered`, varför sorteringen nedan funkar. Verifiera detta i REPL när du klurat ut hur implementera `compareTo`.

```
case class Point(x: Int, y: Int) extends Comparable[Point] {
    def distanceFromOrigin: Double = ???
    def compareTo(that: Point): Int = ???
}
```

```
1 scala> val xs = Seq(Point(10,10), Point(2,1), Point(5,3), Point(0,0))
2 scala> xs.sorted
```

**Uppgift 8.** `java.util.Arrays.binarySearch` I klassen `java.util.Arrays`<sup>7</sup> finns en statisk metod `binarySearch` som kan användas enligt nedan.

```
1 scala> val xs = Array(5,1,3,42,-1)
2 scala> java.util.Arrays.sort(xs)
3 scala> xs
4 scala> java.util.Arrays.binarySearch(xs, 42)
5 scala> java.util.Arrays.binarySearch(xs, 43)
```

Skriv ett valfritt Javaprogram som testar `java.util.Arrays.binarySearch`. Använd en array av typen `int[]` med några heltal som först sorteras med `java.util.Arrays.sort`. Skriv ut det som returneras från `java.util.Arrays.binarySearch` i olika fall genom att asöka efter tal som finns först, mitt i, sist och tal som saknas. *Tips:* Man kan deklarera en array, allokerar den och fylla den med värden så här i Java:

```
int[] xs = new int[]{5, 1, 3, 42, -1};
```

**Uppgift 9.** *Auto(un)boxing.* I JVM måste typparametern för generiska klasser vara av referenstyp. I Scala löser kompilatorn detta åt oss så att vi ändå kan ha t.ex. `Int` som argument till en typparameter i Scala, medan man i Java *inte* direkt kan ha den primitiva typen `int` som typparameter till t.ex. `ArrayList`.

I Java och i den underliggande plattformen JVM används s.k. wrapper-klasser för att lösa detta, t.ex. genom wrapper-klassen `Integer` som boxar den primitiva typen `int`. Java-kompilatorn har stöd för att automatiskt packa in värden av primitiv typ i sådana wrapper-klasser för att skapa referenstyper och kan även automatiskt packa upp dem.

a) Studera hur Scala-kompilatorn låter oss arbeta med en `Cell[Int]` även om det underliggande JVM:ens körtidstyp (eng. *runtime type*) är en wrapper-klass. Man kan se JVM-körtidstypen med metoderna `getClass` och `getTypeName` enligt nedan.

```
1 scala> class Cell[T](var value: T){
2     val typeName: String = value.getClass.getTypeName
3     override def toString = "Cell[" + typeName + "]" + value + ")"
4 }
5 scala> val c = new Cell[Int](42)
6 scala> c.value.getClass.getTypeName
```

b) Vad är körtidstypen för `c.value` ovan? Förklara hur det kan komma sig trots att vi deklarerade med typargumentet `Int`?

c) Studera dokumentationen för `java.lang.Integer`<sup>8</sup> och testa i REPL några av *klassmetoderna* (de som är **static** och därmed kan anropas med punktnotation direkt på klassens namn utan **new**) och några av *instansmetoderna* (de som inte är **static**).

```
1 scala> Integer. //tryck TAB
2 scala> Integer.
3 scala> Integer.toBinaryString(42)
4 scala> Integer.valueOf(42)
5 scala> val i = new Integer(42)
6 scala> i. // tryck TAB
7 scala> i.toString
8 scala> i.compareTo // tryck TAB 2 gånger
9 scala> i.compareTo(Integer.valueOf(42))
10 scala> i.compareTo(42) // varför fungerar detta?
```



d) Enligt dokumentationen<sup>9</sup> tar instansmetoden `compareTo` i klassen `Integer` en `Integer`

<sup>7</sup>[docs.oracle.com/javase/8/docs/api/java/util/Arrays.html](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html)

<sup>8</sup>[docs.oracle.com/javase/8/docs/api/java/lang/Integer.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html)

<sup>9</sup>[docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#compareTo-java.lang.Integer-](https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#compareTo-java.lang.Integer-)

som parameter. Hur kan det då komma sig att sista raden ovan fungerar med en `Int`?


e) Studera nedan Java-program och beskriv vad som kommer att skrivas ut *innan* du kompilerar och testkör.


```

1  import java.util.ArrayList;
2
3  public class Autoboxing {
4      public static void main(String[] args) {
5          ArrayList<Integer> xs = new ArrayList<Integer>();
6          for (int i = 0; i < 42; i++) {
7              xs.add(new Integer(i));
8          }
9          for (Integer x: xs) {
10             int y = x.intValue() * 10;
11             System.out.print(y + " ");
12         }
13         int pos = xs.size();
14         xs.add(pos, new Integer(0));
15         System.out.println("\n\n[0]: " + xs.get(0).intValue());
16         System.out.println "[" + pos + "]: " + xs.get(pos));
17         if (xs.get(0) == xs.get(pos)) {
18             System.out.println("EQUAL");
19         } else {
20             System.out.println("NOT EQUAL");
21         }
22     }
23 }

```

f) Ändra i programmet ovan så att autoboxing och autounboxing utnyttjas på alla ställen där så är möjligt. Utnyttja även att `toString`-metoden på `Integer` ger samma stränrepresentation som `int` vid utskrift. Fixa också så att du undviker *fallgropen* att i Java jämföra med referenslikhet i stället för att använda `equals`. Testa så att allt fungerar som det borde efter dina ändringar.

 g) Antag att du råkar skriva `xs.add(0, pos)` på rad 14 i ditt program från föregående uppgift. Förklara hur autoboxingen stjälper dig i en *fallgrop* då.

 h) Med ledning av de båda tidigare deluppgifterna: sammanfatta de två nämnda fallgropar med autoboxing i Java i två generella punkter, så att du har nytta av att memorera dem inför din framtida Javakodning.

**Uppgift 10.** *CollectionConverters*. Med `import scala.jdk.CollectionConverters._` får man i sina Scalaprogram tillgång till de smidiga metoderna `asJava` och `asScala` som översätter mellan motsvarande samlingar i resp språks standardbibliotek. Kör nedan i REPL och gör efterföljande deluppgifter.

```

1  scala> val sv = Vector(1,2,3)
2  scala> val ss = Set('a','b','c')
3  scala> val sm = Map("gurka" -> 42, "tomat" -> 0)
4  scala> val ja = new java.util.ArrayList[Int]
5  scala> ja.add(42)
6  scala> val js = new java.util.HashSet[Char]
7  scala> js.add('a')

```

```
8 scala> import scala.jdk.CollectionConverters._
```

- Till vilka typer konverteras Scalasamlingarna `Vector[Int]`, `Set[Char]` och `Map[String, Int]` om du anropar metoden `asJava` på dessa?
- Till vilka typer konverteras Javasamlingarna `ArrayList[Int]` och `HashSet[Char]` om du anropar metoden `asScala` på dessa? Blir det föränderliga eller oföränderliga motsvarigheter?
- Vad får resultatet för typ om du kör `toSet` på en samling av typen `mutable.Set`?
- Undersök hur du kan efter att du gjort `sm.asJava.asScala` anropa ytterligare en metod för att få tillbaka en oföränderlig `immutable.Map`.
- Läs mer i dokumentationen om `CollectionConverters`<sup>10</sup> och prova några fler konverteringar.

**Uppgift 11.** Hur fungerar en **switch**-sats i Java (och flera andra språk)? Det händer ofta att man vill testa om ett värde är ett av många olika alternativ. Då kan man använda en sekvens av många **if-else**, ett för varje alternativ. Men det finns ett annat sätt i Java och många andra språk: man kan använda **switch** som kollar flera alternativ i en och samma sats, se t.ex. [en.wikipedia.org/wiki/Switch\\_statement](https://en.wikipedia.org/wiki/Switch_statement).

- Skriv in nedan kod i en kodeditor. Spara med namnet `Switch.java` och kompilera filen med kommandot `javac Switch.java`. Kör den med `java Switch` och ange din favoritgrönsak som argument till programmet. Vad händer? Förklara hur **switch**-satsen fungerar.

```
1 public class Switch {
2     public static void main(String[] args) {
3         String favorite = "selleri";
4         if (args.length > 0) {
5             favorite = args[0];
6         }
7         System.out.println("Din favoritgrönsak: " + favorite);
8         char firstChar = Character.toLowerCase(favorite.charAt(0));
9         System.out.print("Jag tycker att ");
10        switch (firstChar) {
11            case 'g':
12                System.out.println("gurka är gott!");
13                break;
14            case 't':
15                System.out.println("tomat är gott!");
16                break;
17            case 'b':
18                System.out.println("broccoli är gott!");
19                break;
20            default:
21                System.out.println(favorite + " är mindre gott...");
22                break;
23        }
24    }
25 }
```

- Vad händer om du tar bort **break**-satsen på rad 16?

**Uppgift 12.** Fånga undantag i Java med en **try-catch**-sats. Det finns som vi såg i förra uppgiften inbyggt stöd i JVM för att hantera när program avbryts på oväntade sätt, t.ex. på grund av division med noll eller ej förväntade indata från användaren. Spara koden nedan<sup>11</sup> i en fil med namnet `TryCatch.java` och kompilera med `javac TryCatch.java` i terminalen.

<sup>10</sup>[docs.scala-lang.org/overviews/collections-2.13/conversions-between-java-and-scala-collections.html](https://docs.scala-lang.org/overviews/collections-2.13/conversions-between-java-and-scala-collections.html)

<sup>11</sup><https://github.com/lunduniversity/introprog/blob/master/compendium/examples/TryCatch.java>

```

1 // TryCatch.java
2
3 public class TryCatch {
4     public static void main(String[] args) {
5         int input;
6         int output;
7         if (args[0].equals("safe")) {
8             try {
9                 input = Integer.parseInt(args[1]);
10                System.out.println("Skyddad division!");
11                output = 42 / input;
12            } catch (Exception e) {
13                System.out.println("Undantag fångat: " + e);
14                System.out.println("Dividerar ändå med säker default!");
15                input = 1;
16                output = 42 / input;
17            }
18        } else {
19            input = Integer.parseInt(args[0]);
20            System.out.println("Oskyddad division!");
21            output = 42 / input;
22        }
23        System.out.println("42 / " + input + " == " + output);
24    }
25 }

```

a) Förklara vad som händer när du kör programmet med olika indata:

```

1 > java TryCatch 42
2 > java TryCatch 0
3 > java TryCatch safe 42
4 > java TryCatch safe 0
5 > java TryCatch

```

b) Vad händer om du "glömmer bort" raden 15 och därmed missar att initialisera input? Hur lyder felmeddelandet? Är det ett körtidsfel eller kompileringsfel?

**Uppgift 13. Matriser med array i Java.** Om man redan vid allokering vet hur många element en matris ska ha, använder man i Java gärna en array av arrayer. En heltalsmatris (en array av array av heltal) skrivs i Java med dubbla hakparentespar `int[][]` direkt efter typen. Vid allokering använder man nyckelordet `new` och antalet element i respektive dimension anges inom hakparenteserna; t.ex. så ger `new int[42][21]` en matris med 42 rader och 21 kolumner, vilket motsvarar att man i Scala skriver `Array.ofDim[Int](42,21)`<sup>12</sup>. Alla element får defaultvärdet för typen, här 0 för heltal.

a) Skriv nedan program i en editor och spara koden i filen `JavaArrayTest.java` och kompilera med `javac JavaArrayTest.java` och kör i terminalen med `java JavaArrayTest` och undersök utskriften. Förklara vad som händer. Notera några skillnader i hur matriser används i Scala och Java.

```

public class JavaArrayTest {

    public static void showMatrix(int[][] m){
        System.out.println("\n--- showMatrix ---");
        for (int row = 0; row < m.length; row++){

```

<sup>12</sup>Ett annat sätt att skriva detta i Scala där initialvärdet framgår explicit: `Array.fill(42,21)(0)`

```

        for (int col = 0; col < m[row].length; col++) {
            System.out.print "[" + row + " ]");
            System.out.print "[" + col + " ] = ";
            System.out.print(m[row][col] + "; ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    System.out.println("Hello JavaArrayTest!");
    int[][] xss = new int[10][5];
    showMatrix(xss);
}
}

```

b) Implementera nedan metod `fillRnd` inuti klassen `JavaArrayTest`. Skriv kod som fyller matrisen `m` med slumpstal mellan 1 och `n`.

```

public static void fillRnd(int[][] m, int n){
    /* ??? */
}

```

*Tips:* med detta uttryck skapas ett slumpstal mellan 1 och 42 i Java:

`(int) (Math.random() * 42 + 1);`

där typkonverteringen `(int)` ger samma effekt som ett anrop av metoden `toInt` i Scala; alltså att dubbelprecisionsflyttal omvandlas till heltal genom avkortning av alla decimaler.

Ändra huvudprogrammet så det anropar `fillRnd(xss, 6)`. Programmet ska ge en utskrift som liknar följande:

```

1 Hello JavaArrayTest!
2
3 --- showMatrix ---
4 [0][0] = 6; [0][1] = 2; [0][2] = 6; [0][3] = 3; [0][4] = 5;
5 [1][0] = 2; [1][1] = 4; [1][2] = 6; [1][3] = 1; [1][4] = 1;
6 [2][0] = 5; [2][1] = 4; [2][2] = 4; [2][3] = 1; [2][4] = 5;
7 [3][0] = 4; [3][1] = 6; [3][2] = 6; [3][3] = 1; [3][4] = 3;
8 [4][0] = 4; [4][1] = 6; [4][2] = 2; [4][3] = 3; [4][4] = 2;
9 [5][0] = 2; [5][1] = 4; [5][2] = 5; [5][3] = 5; [5][4] = 3;
10 [6][0] = 6; [6][1] = 5; [6][2] = 2; [6][3] = 4; [6][4] = 3;
11 [7][0] = 1; [7][1] = 6; [7][2] = 1; [7][3] = 6; [7][4] = 2;
12 [8][0] = 1; [8][1] = 1; [8][2] = 5; [8][3] = 3; [8][4] = 2;
13 [9][0] = 1; [9][1] = 1; [9][2] = 1; [9][3] = 5; [9][4] = 4;

```

**Uppgift 14.** Översätta från Java till Scala. Översätt nedan kod från Java till Scala. Skriv koden i en fil som heter `showInt.scala` och kalla Scala-objektet med `main`-metoden för `showInt`. Läs tipsen som följer efter koden innan du börjar.

```

1 import java.util.Scanner;
2
3 public class JShowInt {
4     private static Scanner scan = new Scanner(System.in);
5

```

```
6  public static void show(Object obj) {
7      System.out.println(obj);
8  }
9
10 public static void show(Object obj, String msg) {
11     System.out.println(msg + obj);
12 }
13
14 public static String repeatChar(char ch, int n) {
15     StringBuilder sb = new StringBuilder();
16     for (int i = 0; i < n; i++) {
17         sb.append(ch);
18     }
19     return sb.toString();
20 }
21
22 public static String readLine(String prompt) {
23     System.out.print(prompt);
24     return scan.nextLine();
25 }
26
27 public static void showInt(int i) {
28     int leading = Integer.numberOfLeadingZeros(i);
29     String binaryString =
30         repeatChar('0', leading) + Integer.toBinaryString(i);
31     show(i, "Heltal: ");
32     show((char) i, "Tecken: ");
33     show(binaryString, "Binärt: ");
34     show(Integer.toHexString(i), "Hex : ");
35     show(Integer.toOctalString(i), "Okalt: ");
36 }
37
38 public static void loop() {
39     boolean hasExploded = false;
40     while (!hasExploded) {
41         try {
42             String s = readLine("Heltal annars pang: ");
43             showInt(Integer.parseInt(s));
44         } catch (Throwable e){
45             show(e);
46             hasExploded = true;
47         }
48     }
49     show("PANG!");
50 }
51
52 public static void main(String[] args){
53     if (args.length == 0) {
```



```

54         loop();
55     } else {
56         for (String arg: args) {
57             showInt(Integer.parseInt(arg));
58             System.out.println();
59         }
60     }
61 }
62 }

```

*Tips:*

- En Javaklass med bara statiska medlemmar motsvaras av ett singletonobjekt i Scala, alltså en **object**-deklaration. Scala har därför inte nyckelordet **static**.
- Typen Object i Java motsvaras av Scalas Any.
- Du kan använda Scalas möjlighet med default-argument (som saknas i Java) för att bara definiera en enda show-metod med en tom sträng som default msg-argument.
- I Scala har objekt av typen Char en metod **def** \*(n: Int): String som skapar en sträng med tecknet repeterat n gånger. Men du kan ju välja att ändå implementera metoden repeatChar med StringBuilder som nedan om du vill träna på att översätta en **for**-loop från Java till Scala.
- I stället för Scanner.nextLine kan du använda scala.io.StdIn.readLine som tar en prompt som parameter, men du kan också använda Scanner i Scala om du vill träna på det.
- I Java *måste* man använda nyckelordet **return** om metoden inte är en **void**-metod, medan man i Scala faktiskt *får* använda **return** även om man brukar undvika det och i stället utnyttja att satser i Scala också är uttryck.

Kompilera din Scala-kod och kör i terminalen och testa så att allt funkar. Vill du även kompilera Java-koden så finns den i kursens repo i filen  
compendium/examples/scalajava/JShowInt.java

**Uppgift 15.** *Innehållslikhet och referenslikhet i Java.* Studera och prova denna fallgrop med innehållslikhet: [TestPitfall3.java](#)



**Uppgift 16.** *Implementera innehållslikhet i Java.* Studera fallgropar för hur man skriver en equals-metod i Java här: [www.artima.com/lejava/articles/equality.html](http://www.artima.com/lejava/articles/equality.html) och jämför med det fullständiga receptet för hur man skriver en välfungerande equals och hashCode i Scala här: [www.artima.com/pins1ed/object-equality.html](http://www.artima.com/pins1ed/object-equality.html)

- Vilka skillnader och likheter finns vid överskuggning av equals i Java respektive Scala, som ska ge en fungerande innehållstest för en hierarki med bastyper och subtyper?
- Vilka fallgropar är gemensamma för Java och Scala?

**Uppgift 17.** *Array och **for**-sats i Java.* Ladda ner programet nedan från kursens GitHub-repo: [compendium/examples/DiceReg.java](#)

- Kompilera med javac DiceReg.java och kör med java DiceReg 10000 42 och förklara vad som händer.

```

// DiceReg.java
import java.util.Random;

```



```

public class DiceReg {
    public static void main(String[] args) {
        int[] diceReg = new int[6];
        int n = 100;
        Random rnd = new Random();
        if (args.length > 0) {
            n = Integer.parseInt(args[0]);
        }
        System.out.print("Rolling the dice " + n + " times");
        if (args.length > 1) {
            int seed = Integer.parseInt(args[1]);
            rnd.setSeed(seed);
            System.out.print(" with seed " + seed);
        }
        System.out.println(".");
        for (int i = 0; i < n; i++) {
            int pips = rnd.nextInt(6);
            diceReg[pips]++;
        }
        for (int i = 1; i <= 6; i++) {
            System.out.println("Number of " + i + "'s: " +
                               diceReg[i-1]);
        }
    }
}

```

- b) Beskriv skillnaderna mellan Scala och Java, vad gäller syntaxen för array och **for**-sats. Beskriv några andra skillnader mellan språken som syns i programmet ovan.
- c) Ändra i programmet ovan så att loop-variabeln *i* skrivs ut i varje runda i varje **for**-sats. Kompilera om och kör.
- d) Skriv om programmet ovan genom att abstrahera huvudprogrammets delar till de statistiska metoderna `parseArguments`, `registerPips` och `printReg` enligt nedan skelett. Spara programmet i filen `DiceReg2.java` och kompilera med `javac DiceReg2.java` i terminalen.

```

// DiceReg2.java
import java.util.Random;

public class DiceReg2 {
    public static int[] diceReg = new int[6];
    private static Random rnd = new Random();

    public static int parseArguments(String[] args) {
        // ???
        return n;
    }

    public static void registerPips(int n){
        // ???
    }
}

```

```

    }

    public static void printReg() {
        // ???
    }

    public static void main(String[] args) {
        int n = parseArguments(args);
        registerPips(n);
        printReg();
    }
}

```

e) Starta Scala REPL i samma katalog som filen `DiceReg2.class` ligger i och kör nedan 7 rader i REPL och förklara vad som händer:

```

1 scala> DiceReg2.main(Array("1000","42"))
2 scala> DiceReg2.diceReg
3 scala> DiceReg2.registerPips(1000)
4 scala> DiceReg2.printReg
5 scala> DiceReg2.registerPips(1000)
6 scala> DiceReg2.printReg
7 scala> DiceReg2.rnd

```

**Uppgift 18.** *Läsa in sekvens av tal med Scanner i Java.* Läs i Java-delen av snabbreferensen om `java.util.Scanner`. Med `new Scanner(System.in)` skapas ett objekt som kan läsa in tal från teckensträngar som användaren skriver i terminalfönstret, så som visas i Java-programmet nedan:

```

// DiceScanBuggy.java
import java.util.Random;
import java.util.Scanner;

public class DiceScanBuggy {
    public static int[] diceReg = new int[6];
    public static Scanner scan = new Scanner(System.in);

    public static void registerPips(){
        System.out.println("Enter pips separated by blanks.");
        System.out.println("End with -1 and <Enter>.");
        boolean isPips = true;
        while (isPips && scan.hasNextInt()) {
            int pips = scan.nextInt();
            if (pips >= 1 && pips <=6 ) {
                diceReg[pips]++;
            } else {
                isPips = false;
            }
        }
    }
}

```

```
public static void printReg() {
    for (int i = 0; i < 6; i++) {
        System.out.println("Number of " + i + "'s: " +
            diceReg[i-1]);
    }
}

public static void main(String[] args) {
    registerPips();
    printReg();
}
}
```

Ladda ner programmet [compendium/examples/DiceScanBuggy.java](#) och kompilera och kör med indatasekvensen 1 2 3 4 -1 och notera hur registreringen sker.

- a) Sök upp och läs JDK8-dokumentationen av `java.util.Scanner`. Vad gör `hasNextInt()` och `nextInt()`?
- b) Programmet fungerar inte som det ska. Du behöver korrigera 3 saker för att programmet ska göra rätt. Rätta buggarna och spara det rättade programmet som `DiceScan.java`. Kompilera och testa det rättade programmet.

## J.3 Laboration: java

### Mål

- ☐ Förstå skillnaden mellan primitiva typer och objekt i Java.
- ☐ Kunna förklara hur autoboxing fungerar i Java.
- ☐ Kunna förklara vad statiska metoder och attribut i Java innebär.
- ☐ Kunna använda ArrayList och arrayer i Java.
- ☐ Kunna använda Java-klassen Scanner.
- ☐ Kunna skapa en for-sats i Java.
- ☐ Känna till hur man kan förenkla användandet av Java och Scala i samma program med hjälp av `scala.jdk.CollectionConverters`.

### Förberedelser

- ☐ Gör övningarna tidigare i detta Appendix.
- ☐ Studera given kod här:  
[github.com/lunduniversity/introprog/tree/master/workspace/javatext/](https://github.com/lunduniversity/introprog/tree/master/workspace/javatext/)

### J.3.1 Krav

Du ska skapa ett textspel för terminalen som är (lagom) intressant/roligt att spela och sparar poäng per spelomgång för olika spelare. Till din hjälp har du den färdiga filen `Main.java` (som går bra att förändra om det behövs) samt de två kodskeletten `Game.java` och `UserInterface.scala`. Ditt textspel ska köras i terminalen och uppfylla följande krav och riktlinjer:

1. När ditt program kör ska man ska kunna starta flera spelomgångar efter varandra utan att behöva avsluta programmet.
2. För varje spelomgång ska programmet komma ihåg spelarens namn<sup>13</sup> med tillhörande resultat.
3. Efter begäran ska programmet kunna visa en topplista med bästa poäng, både för alla spelare och för ett specifikt spelarnamn.
4. Koden för själva spelet ska vara skriven i Java, men Scala ska användas för att implementera funktionerna i singelobjektet `UserInterface`.
5. I Scala-koden ska du för träningens skull använda Java-klassen `java.util.Scanner` när du läser in data från terminalen.
6. Koden i singelobjektet `UserInterface` ska använda omvandlingsmetoden `asScala` efter **import** `scala.jdk.CollectionConverters._` för att omvandla argument av typen `java.util.ArrayList` till `scala.collection.mutable.Buffer`<sup>14</sup>.
7. Ditt spel ska i Java-kod använda minst en av datastrukturerna `ArrayList`, `HashSet`, `HashMap` ur paketet `java.util`, samt minst en array. (Den givna koden i `Main.java` räknas inte till detta krav.)
8. Du ska spela någon annans halvfärdiga spel och, efter att du studerat koden, ge återkoppling på kodens läsbarhet.

<sup>13</sup>eller spelarnas namn om det är ett spel för två eller flera personer

<sup>14</sup>Notera att `asJava` på `Buffer` ger en Java-samling av typen `List`.

9. Du ska låta någon annan spela ditt halvfärdiga spel och visa din kod och fråga om återkoppling på läsbarheten. Du ska anteckna den återkoppling du får.
10. Du ska inför redovisningen förbereda följande:
  - (a) en kort genomgång av spelets idé,
  - (b) en kort förklaring av kodens struktur och de olika Java-klassernas ansvar,
  - (c) en kort redogörelse för den återkoppling du fått på din kods läsbarhet och hur du arbetat med att förbättra läsbarheten under dina stegvisa utvidgningar av din kod,
  - (d) en lista med koncept som du tränat på när du skapat ditt textspel.

### J.3.2 Frivilliga extrauppgifter

1. Spara resultat i en fil efter varje spelomgång, och läs in resultat från filen antingen när programmet startas eller när användaren vill se poänglistan, så att det går att se spelresultat från tidigare körningar av programmet. Den kod du behöver lägga till för att åstadkomma detta kan vara skriven antingen i Java eller Scala. Tänk på att du kan behöva göra ändringar även i Main-klassen.
2. Mät speltiden för varje spelomgång och spara tiden tillsammans med poängresultatet för respektive spelare.

### J.3.3 Inspiration och tips

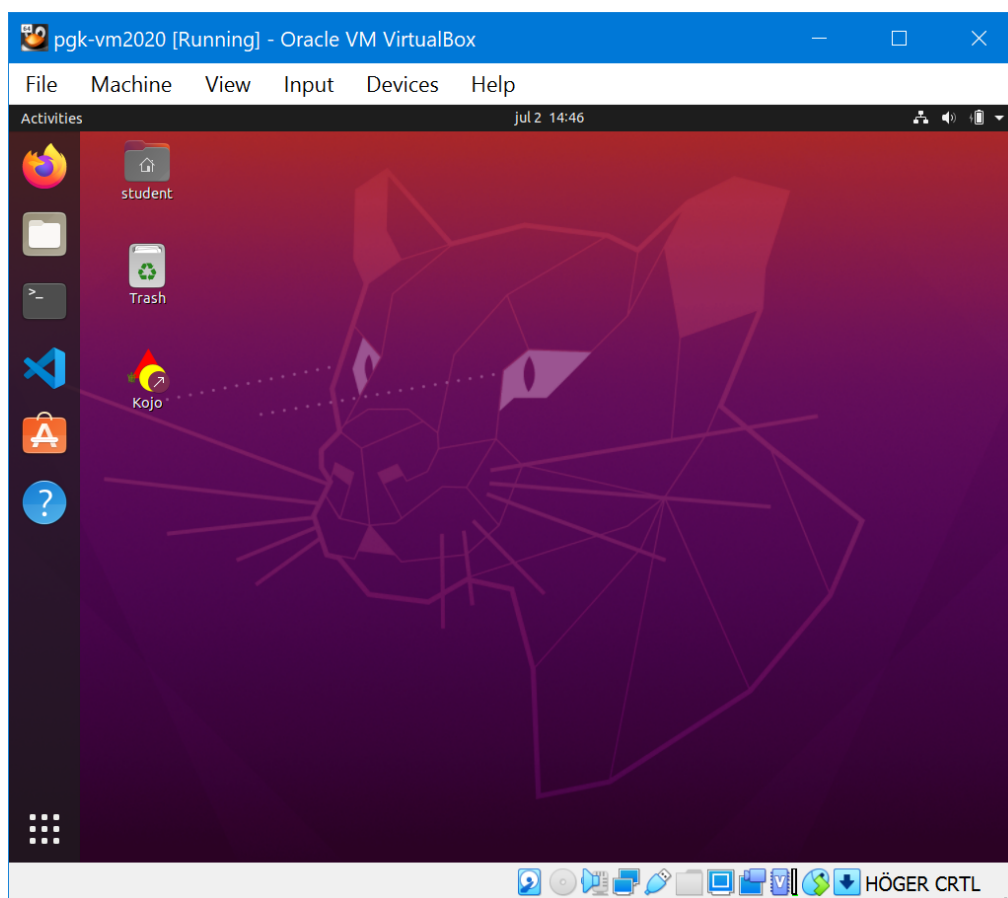
1. Utgå från Hangman i veckans övning eller,
2. Yatzy från tidigare övningar, eller
3. skapa ett kortspel inspirerat av shuffle-labben, eller
4. inspireras av listan med sällskapsspel på wikipedia:  
[sv.wikipedia.org/wiki/Kategori:Sällskapsspel](https://sv.wikipedia.org/wiki/Kategori:Sällskapsspel)
5. eller hitta på ett eget textspel.
6. Börja med en starkt förenklad variant som du sedan bygger vidare på.
7. Kompilera och testa efter varje ändring, så att du hela tiden har ett fungerande program.
8. Dela upp din spelkod i flera metoder, och även flera klasser om det är lämpligt.
9. Det finns mycket information på nätet om hur man skriver Java-kod och använder JDK, t.ex. på <https://stackoverflow.com/>
10. Träna på att använda JDK-dokumentationen här:  
<https://docs.oracle.com/javase/8/docs/api/>

## Appendix K

# Virtuell maskin

### K.1 Vad är en virtuell maskin?

Du kan köra alla kursens verktyg i en så kallad **virtuell maskin** (förk. vm, eng. *virtual machine*). Detta är ett enkelt och säkert sätt köra ett annat operativsystem i en ”sandlåda” som inte påverkar din dators ursprungliga operativsystem. Figur K.1 visar kursens virtuella maskin pgk-vm2020. Exekveringen av en vm sker på en **värddator** (eng. *host*). I figur K.1 körs kursens vm i en Windows-värd med virtualiseringsapplikationen *VirtualBox*<sup>1</sup>, som är öppen och gratis och finns för Linux-, Windows- och macOS-värdar.



**Figur K.1:** Den virtuella maskinen pgk-vm2020 med Ubuntu 22.04 under Windows 10 i VirtualBox 7.0.

---

<sup>1</sup>[en.wikipedia.org/wiki/VirtualBox](https://en.wikipedia.org/wiki/VirtualBox)

## K.2 Vad innehåller kursens vm?

Kursens virtuella maskin kör en minimal installation av Ubuntu Linux och har verktyg för programmering med Scala förinstallerade. Detta ingår i kursens vm:

- java och javac med OpenJDK 17
- scala och scalac version 3.2.2
- Kojo 2.9.24
- VS code version 1.78 med Scala (Metals) version v1.19
- sbt version 1.8.2

Du kan själv uppdatera dessa applikationer till dess senaste versioner, och även installera fler applikationer, när du väl startat den virtuella maskinen. Se vidare kursens hemsida under "Verktyg".

## K.3 Installera kursens vm

Det går lite långsammare att köra i en virtuell maskin jämfört med att köra direkt "på metallen", då det sker vissa översättningar och kontroller under virtualiseringsprocessen som annars inte behövs. Den virtuella maskinen behöver dessutom få en rejäl andel av din dators minne. Så för att köra en virtuell maskin utan att det ska bli segt behövs en ganska snabb processor, gärna över 2 GHz, och ganska mycket minne, gärna minst 8 GB.

Även om det går lite segt är en virtuell maskin ett utmärkt sätt att prova på Linux och Ubuntu. Eftersom man lätt kan spara undan en hel maskin är det ett bra sätt att experimentera med olika inställningar och installationer utan att din normala miljö påverkas. Du kan lätt kлона maskinen för att spara undan den i ett visst läge. Och kör du terminalfönster och en enkel editor brukar svag prestanda och lite minne inte vara ett stort problem. Om du tycker det går alltför segt kan du istället installera Linux direkt på din dator jämsides ditt andra operativsystem – fråga någon som vet om hur man gör detta.

Gör så här för att installera VirtualBox och köra kursens virtuella maskin:

1. Ladda ner VirtualBox 7.0 (eller senare version) för ditt operativsystem (t.ex. "Platform Packages for Windows Hosts") här och installera:  
<https://www.virtualbox.org/wiki/Downloads>
2. Ladda ner filen `pgk-vm2020.ova` här:  
<http://cs.lth.se/pgk/vm/>  
OBS! Då filen är mer än 5 GB kan nedladdningen ta *mycket* lång tid, kanske flera timmar beroende på din internetuppkoppling. Har du problem med nedladdningstider kan du prova att ladda ner filen till ett USB-minne på skolans datorer, så att överföringen sker lokalt i E-huset.
3. Öppna VirtualBox och välj *File* → *Import appliance...* och välj filen `pgk-vm2020.ova` och klicka **Next** och sedan **Import**. Själva importen kan ta lång tid, kanske flera tiotals minuter beroende på hur snabbt din dator läser från disk.
4. Starta maskinen **pgk-vm2020** med ett dubbelklick. Ha lite tålamod innan maskinen är igång. Du kan behöva justera skärmstorleken i värdmaskinsmenyn *View*. Du behöver lösenordet **pgkBytMig2020** för att logga in och för att installera nya program.

5. Det kan hända att du får ett felmeddelande som innehåller något som liknar "Intel VT-x" eller "Hyper-V", så som beskrivs här:  
[www.howtogeek.com/213795/how-to-enable-intel-vt-x-in-your-computers-bios-or-uefi-firmware](http://www.howtogeek.com/213795/how-to-enable-intel-vt-x-in-your-computers-bios-or-uefi-firmware)  
Då behöver du tillåta virtualiseringsfunktioner i BIOS på din dator. Om du inte vet hur du ska göra detta, be någon som vet om hjälp.

6. Börja med att öppna ett terminalfönster och uppdatera mjukvaran på din virtuella maskin med detta terminalkommando:

```
sudo apt update && sudo apt dist-upgrade
```

7. Byt lösenord genom att trycka på windowsknappen och skriva "users", klicka på *Users*-ikonen och trycka **Password**.
8. Skriv `scala` i ett terminalfönster och du är igång och kan börja göra övningarna i detta kompendium!
9. Om allt verkar fungera fint kan du nu prova att öka minnet och även sätta på 3D-accelereringen för snabbare grafikrendering så här:
- (a) Stäng maskinen genom att välja *Shut Down...* i systemmenyn.
  - (b) Markera maskinen **pgk-vm2020** och välj menyn *Machine* → *Settings...* (eller tryck Ctrl+S) och undersök inställningarna. Se speciellt under fliken **System** och **Motherboard** där det står hur mycket **Base memory** du tilldelar. Om din värddator har gott om minne (undersök exakt hur mycket) så kan du med fördel öka minnet till minst 4096 MB, speciellt om du tänker köra IntelliJ. I din virtuella maskin kan du undersöka hur stor andel av maskinens minne som är ockuperat genom att trycka på windowsknappen, skriva "syst", klicka på **System Monitor** och välja fliken **Resources**.
  - (c) Ändra inställningar i menyn *Settings...* → *Display* genom att i fliken **Acceleration** under **Screen** markera ☒ **Enable 3D acceleration**. Stara maskinen. Om det fungerar så blir animeringar avsevärt snyggare och smidigare. Om det inte fungerar, stäng av maskinen med *Power off* och avmarkera ☐ **Enable 3D acceleration** igen.



# **Del IV**

## **Lösningar**

## **Appendix L**

# **Lösningar till övningarna**

## L.1 Lösning expressions

### L.1.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** *Para ihop begrepp med beskrivning.*

litteral	1	↪	D	anger ett specifikt datavärde
sträng	2	↪	G	en sekvens av tecken
sats	3	↪	F	en kodrad som gör något; kan särskiljas med semikolon
uttryck	4	↪	H	kombinerar värden och funktioner till ett nytt värde
funktion	5	↪	K	vid anrop beräknas ett returvärde
procedur	6	↪	J	vid anrop sker (sido)effekt; returvärdet är tomt
exekveringsfel	7	↪	N	kan inträffa medan programmet kör
kompileringsfel	8	↪	M	kan inträffa innan exekveringen startat
abstrahera	9	↪	A	att införa nya begrepp som förenklar kodningen
kompilera	10	↪	C	att översätta kod till exekverbar form
typ	11	↪	I	beskriver vad data kan användas till
for-sats	12	↪	O	bra då antalet repetitioner är bestämt i förväg
while-sats	13	↪	P	bra då antalet repetitioner ej är bestämt i förväg
tilldelning	14	↪	L	för att ändra en variabels värde
flyttal	15	↪	E	decimaltal med begränsad noggrannhet
boolesk	16	↪	B	antingen sann eller falsk

**Lösn. uppg. 2.** *Utskrift i Scala REPL.*

a) Till exempel:

```
scala> println("hejsan svejsan")
```

b) Om högerparentes fattas får man fortsätta skriva på nästa rad. Detta indikeras med vertikalstreck i början av varje ny rad:

```
scala> println("hejsan svejsan"
| + "!"
| )
hejsan svejsan!
```

**Lösn. uppg. 3.** *Konkatenering av strängar.*

a)

```
scala> "gurk" + "burk"
res1: String = gurkburk
```

värde: "gurkburk", typ: String

b)

```
scala> res1 * 42
res2: String = gurkatomatgurkatomatgurkatomatgurkatomatgurkatomatgurkatomatgurkatomatgurka
```

**Lösn. uppg. 4.** När upptäcks felet?

- a) Typ: String, värde: "hejhejhej"
- b) Körtidsfel:

```
scala> "hej" * Int.MaxValue
java.lang.OutOfMemoryError: Java heap space
```

- c) Kompileringsfel: (indikeras av texten <console> ... error:)

```
scala> "hej" * true
<console>:12: error: type mismatch;
 found   : Boolean(true)
 required: Int
    "hej" * true
```

Ett typfel innebär att kompilatorn inte kan få typerna att överensstämman i t.ex. ett funktionsanrop. I Scala får vi reda på typfel redan vid kompilering medan i andra språk (t.ex. Javascript) upptäcks sådana fel under exekveringen, i värsta fall genom svårhittade buggar som kanske först märks långt senare.

**Lösn. uppg. 5.** Litteraler och typer.

- a)

1	1	↔	E	Int
1L	2	↔	G	Long
1.0	3	↔	J	Double
1D	4	↔	F	Double
1F	5	↔	H	Float
'1'	6	↔	I	Char
"1"	7	↔	A	String
true	8	↔	C	Boolean
false	9	↔	B	Boolean
()	10	↔	D	Unit

- b) Värdet går över gränsen för vad som får plats i ett 32 bitars heltal och "börjar om" på det minsta möjliga heltalet Int.MinValue eftersom det är så binär aritmetik med begränsat antal bitar fungerar i CPU:n.

```
1 scala> Int.MaxValue + 1
2 res3: Int = -2147483648
3
4 scala> Int.MinValue
5 res4: Int = -2147483648
```

- c) Båda är heltal men Long kan representera större tal än Int.
- d) Båda är flyttal men Double har dubbel precision och kan representera större tal med fler decimaler.

**Lösn. uppg. 6.** Matematiska funktioner. Använda dokumentation.

a) Beräkning av  $2^{64} - 1$  med `math.pow` enligt nedan ger ungefär  $1.8 \cdot 10^{19}$

```
1 scala> math.pow(2, 64) - 1
2 res0: Double = 1.8446744073709552E19
```

b) Ja, returtyp-annoteringen : `Double` kan utelämnas.

- Varför kan returtyp utelämnas?  
Eftersom kompilatorns typhärledning kan härleda returtypen.
- Varför kan man vilja utelämna den?  
Det blir kortare att skriva utan.
- Anledningar att ange returtyp:
  - Med explicit returtyp får du hjälp av kompilatorn att redan under kompileringen kontrollera att uttrycket till höger om likhetstecknet har den typ som förväntas.
  - Genom att du anger returtypen explicit får de som enbart läser metodhuvudet (och inte implementationen) tydligt se vad som returneras.

c) Ca 500 km.

```
1 scala> omkrets(12750 / 2) / 80
2 res0: Double = 500.6913291658733
```

### Lösn. uppg. 7. Variabler och tilldelning. Förändringsbar och oföränderlig variabel.

a)

Efter rad 1:    a: Int

Efter rad 2:    a: Int     b: Int

Efter rad 3:    a: Int     b: Int     c: Double

Efter rad 4:    a: Int     b: Int     c: Double

Efter rad 5:    a: Int     b: Int     c: Double

Efter rad 6:    a: Int     b: Int     c: Double

b) Oföränderliga variabler deklareraras med nyckelordet **val**. Det går inte att tilldela en oföränderlig variabel ett nytt värde; vid försök blir det kompileringsfel som lyder **error: reassignment to val**. Kompileringsfel känns igen med hjälp av texten **error:**, så som visas nedan:

```
scala> b = 0
<console>:12: error: reassignment to val
    b = 0
    ^
```

### Lösn. uppg. 8. *Slumptal med `math.random()`.*

a) Ur dokumentationen:

```
/** Returns a Double value with a positive sign,
 *  greater than or equal to 0.0 and less than 1.0.
 */
def random(): Double
```

Dokumentationskommentarer, som börjar med `/**` och slutar med `*/`, ger oss en beskrivning av hur funktionen fungerar. Efter dokumentationskommentaren kommer funktionshuvudet, som här berättar att funktionen heter `random` och alltid kommer att returnera en `Double`. (Verktöget `scaladoc` kan med hjälp av dokumentationskommentarerna automatiskt generera webbsajter med speciella dokumentationssidor och sökfunktioner.)

b)

```
1 scala> def roll: Int = (math.random() * 6 + 1).toInt
2
3 scala> roll
4 res0: Int = 4
5
6 scala> roll
7 res1: Int = 1
```

### Lösn. uppg. 9. *Repetition med `for`, `foreach` och `while`.*

a)

```
for i <- 1 to 100 do print(s"$roll, ")
```

b)

```
(1 to 100).foreach(i => print(s"$roll, "))
```

c)

```
var i = 1
while i <= 100 do { print(s"$roll, "); i = i + 1 }
```

```
var i = 1
while i <= 100 do
  print(s"$roll, ")
  i += 1
```

### Lösn. uppg. 10. *Alternativ med `if`-sats och `if`-uttryck.*

a)

```
for i <- 1 to 100 do
  if roll == 6 then print("GRATTIS! ") else print(":(")
```

eller

```
for (i <- 1 to 100) if (roll == 6) print("GRATTIS! ") else print(":(")
```

b)

```
var i = 1
var n = 0
while i <= 100 do
  if roll == 6 then n = n + 1
  i = i + 1
println("Antalet sexor: " + n)
```

### Lösn. uppg. 11. Sekvens, sats och block.

a) Satserna skapar denna utskrift:

```
san!hej
san!hej
san!hej
san!hej
```

b)

```
scala> def p = { print("hej"); print("san"); println("!") }
scala> p;p;p;p
```

c)

- Klammerparenteser används för att gruppera flera satser. Klammerparenteser behövs om man vill definiera en funktion som består av mer än en sats. Sedan scala 3 kan man istället använda indentering för att definiera en funktion med flera rader och satser.
- Semikolon särskiljer flera satser. Semikolon behövs om man vill skriva många satser på samma rad.

### Lösn. uppg. 12. Heltalsdivision.

4 / 42	1	↪	F	0: Int
42.0 / 2	2	↪	C	21.0: Double
42 / 4	3	↪	B	10: Int
42 % 4	4	↪	G	2: Int
4 % 42	5	↪	A	4: Int
40 % 4 == 0	6	↪	D	true : Boolean
42 % 4 == 0	7	↪	E	false: Boolean

**Lösn. uppg. 13.** *Booleska värden.*

- a) **true**
- b) **false**
- c) **true**
- d) **true**
- e) **false**
- f) **true**
- g) **true**
- h) **true**
- i) Undantag kastas: `java.lang.ArithmeticException: / by zero`
- j) **false**

**Lösn. uppg. 14.** *Booleska variabler.*

2: Ingenting skrivs ut.

4: akta dig!!!

**Lösn. uppg. 15.** *Turtle graphics med Kojo.*

a) Genom att börja din Kojo-program med `sudda` så startar du exekveringen i samma utgångsläge: en tom rityta (eng. *canvas*) där paddan pekar uppåt, pennan är nere och pennans färg är röd. Då blir det lättare att resonera om vad programmet gör från början till slut, jämfört med om exekveringen beror på resultatet av tidigare exekveringar.

b)

```
sudda

fram; vänster
fram; vänster
fram; vänster
fram; vänster
```

c)

```
sudda

fram; vänster
fram; höger

fram; vänster
fram; höger

fram; vänster
fram; höger

fram; vänster
```



## L.1.2 Extrauppgifter; träna mer

**Lösn. uppg. 16.** Typ och värde.

1.0 + 18	1	↪ H	19.0: Double
(41 + 1).toDouble	2	↪ K	42.0: Double
1.042e42 + 1	3	↪ A	1.042E42: Double
12E6.toLong	4	↪ I	12000000: Long
32.toChar.toString	5	↪ E	" ": String
'A'.toInt	6	↪ B	65: Int
0.toInt	7	↪ F	0: Int
'0'.toInt	8	↪ D	48: Int
'9'.toInt	9	↪ L	57: Int
'A' + '0'	10	↪ C	113: Int
('A' + '0').toChar	11	↪ J	'q': Char
"*!%#".charAt(0)	12	↪ G	'*': Char

**Lösn. uppg. 17.** Satser och uttryck.

a) Ett uttryck kan evalueras och resulterar då i ett användbart värde. En sats *gör* något (t.ex. skriver ut något), men resulterar inte i något användbart värde.

b) `println()`

c)

värdeSaknas innehåller Unit

Skriver ut Unit

Skriver ut " () "

Skriver ut " () "

Skriver först ut hej med det innersta anropet och sen ( ) med det yttre anropet

d) Unit

e) Unit

**Lösn. uppg. 18.** Procedur med parameter.

a)

```
var highscore = 0
```

b)

```
def updateHighscore(points: Int): Unit =
  if points > highscore then
    highscore = points
    println("REKORD!")
  else println("GE INTE UPP!")
```

c)

```
def updateHighscore(points: Int): String =  
  if points > highscore then  
    highscore = points  
    "REKORD!"  
  else "GE INTE UPP!"
```

**Lösn. uppg. 19.** *Flyttalsaritmetik.*

a)

```
1 scala> Double.MinPositiveValue  
2 res0: Double = 4.9E-324
```

b)

```
1 scala> Double.MaxValue + Double.MinPositiveValue == Double.MaxValue  
2 res2: Boolean = true
```

**Lösn. uppg. 20.** *if-sats.*

1. Utskrift: falskt
2. Utskrift: sant
3. Inget skrivs ut, funktionen deklarereras men körs ej.
4. Utskrift: 1:krona 2:klave 3:krona 4:krona 5:klave eller liknande beroende på vilka slumpstal `math.random()` ger.

**Lösn. uppg. 21.** *if-uttryck.* Notera typen Any på de sista två uttrycken.

```
scala> if grönsak == "tomat" then "gott" else "inte gott"  
res0: String = inte gott  
  
scala> if frukt == "banan" then "gott" else "inte gott"  
res1: String = gott  
  
scala> if true then grönsak else 42  
res2: Any = gurka  
  
scala> if false then grönsak else 42  
res3: Any = 42
```

**Lösn. uppg. 22.** *Modulo-operatorn % och Booleska värden.*

a)

```
1 scala> def isEven(n: Int): Boolean = n % 2 == 0  
2  
3 scala> isEven(42)  
4 res0: Boolean = true  
5  
6 scala> isEven(43)  
7 res1: Boolean = false
```

b)

```
1 scala> def isOdd(n: Int): Boolean = !isEven(n)
2
3 scala> isOdd(42)
4 res2: Boolean = false
5
6 scala> isOdd(43)
7 res3: Boolean = true
```

**Lösn. uppg. 23.** Skillnader mellan *var*, *val*, *def*.

a)

```
1 scala> var x = 30
2 x: Int = 30
3
4 scala> x + 1
5 res6: Int = 31
6
7 scala> x = x + 1
8 x: Int = 31
9
10 scala> x == x + 1
11 res7: Boolean = false
12
13 scala> val y = 20
14 y: Int = 20
15
16 scala> y = y + 1
17 <console>:12: error: reassignment to val
18     y = y + 1
19       ^
20
21 scala> var z = { println("hej z!"); math.random() }
22 hej z!
23 z: Double = 0.3381365875903367
24
25 scala> def w = { println("hej w!"); math.random() }
26 w: Double
27
28 scala> z
29 res8: Double = 0.3381365875903367
30
31 scala> z
32 res9: Double = 0.3381365875903367
33
34 scala> z = z + 1
35 z: Double = 1.3381365875903368
36
37 scala> w
38 hej w!
39 res10: Double = 0.06420209879434557
40
41 scala> w
42 hej w!
43 res11: Double = 0.5777951341051852
44
45 scala> w = w + 1
46 <console>:12: error: value w_ = is not a member of object
47     w = w + 1
```

b)

- **var** namn = uttryck används för att deklarera en förändringsbar variabel. Namnet kan med hjälp av en tilldelningssats referera till nya värden.
- **val** namn = uttryck används för att deklarera en oföränderlig variabel som efter initialisering inte kan förändras med tilldelningssatser. Vid försök ges kompileringsfel.
- **def** namn = uttryck används för att deklarera en funktion vars uttryck evalueras varje gång den anropas.

**Lösn. uppg. 24.** Skillnaden mellan **if** och **while**.

- Rad 3: Har du tur (50% chans) får du vinst en gång.
- Rad 4: Har du tur får du många vinster i rad. Sannolikheten för  $n$  vinster i rad är  $(\frac{1}{2})^n$ .

### L.1.3 Fördjupningsuppgifter; utmaningar

#### Lösn. uppg. 25. Logik och De Morgans Lagar.

- a) poäng > 1000
- b) poäng > 100
- c) poäng <= highscore
- d) poäng <= 0 || poäng >= highscore
- e) poäng >= 0 && poäng <= highscore
- f) klar
- g) !klar

#### Lösn. uppg. 26. Stränginterpolatorn s.

a)

```
Namnet 'Kim Finkodare' har 12 bokstäver.
```

b)

```
println(s"$f har ${f.size} bokstäver.")  
println(s"$e har ${e.size} bokstäver.")
```

#### Lösn. uppg. 27. Tilldelningsoperatorer.

Efter rad1:    a: Int

Efter rad2:    a: Int     b: Int

Efter rad3:    a: Int     b: Int

Efter rad4:    a: Int     b: Int

Efter rad5:    a: Int     b: Int

Efter rad6:    a: Int     b: Int

#### Lösn. uppg. 28. Stora tal.

a) `BigInt` kan användas i stället för `Int` vid mycket stora heltal. Det finns förstås även `Long` som har dubbelt omfång jämfört med `Int`, medan `BigInt` kan ha godtyckligt många siffror (ända tills minnet tar slut) och kan därmed representera ofantligt stora tal. `BigDecimal` kan användas i stället för `Double` vid mycket stora decimaltal.

b)

```
1 scala> BigInt(2).pow(64)
2 res0: scala.math.BigInt = 18446744073709551616
```

c) Beräkningar går mycket långsammare och de är lite krångligare att använda.

### Lösn. uppg. 29. Precedensregler

- a) 77: Int
- b) 13: Int
- c) -13: Int

### Lösn. uppg. 30. Dokumentation av paket i Java och Scala.

- a) Scala: Pi, Java: PI
- b) Man kan söka och filtrera fram alla förekomster av en viss teckenkombination.
- c) Räknar ut hypotenusan (Pythagoras sats) utan risk för avrundningsproblem i mellanberäkningar.

### Lösn. uppg. 31. Noggrannhet och undantag i aritmetiska uttryck.

- a) -2147483648 vilket motsvarar Int.MinValue.
- b) Ett undantag kastas: java.lang.ArithmeticException: / by zero
- c) 1.0000000000000001E8 (som förväntat)
- d) Avrundas till 1E9 (flyttalsaritmetik med noggrannhetsproblem: ett stort flyttal plus ett (alltför) litet flyttal kan ge samma tal. Det lilla talet "försvinner").
- e) 45.000000000000001 (flyttalsaritmetik med noggrannhetsproblem: enligt "normal" aritmetik ska det bli exakt 45.)
- f) Infinity (som även ges av Double.PositiveInfinity och som representerar den positiva oändligheten).
- g) 2147483647 vilket motsvarar Int.MaxValue.
- h) NaN vilket betyder "Not a Number".
- i) NaN vilket betyder "Not a Number".
- j) Ett undantag kastas: java.lang.Exception: PANG!!!

**Lösn. uppg. 32. Modulo-räkning med negativa tal.** I Scala har resultatet samma tecken som dividenden.

```
1 scala> 1 % 2
2 res0: Int = 1
3
4 scala> -1 % 2
5 res1: Int = -1
6
7 scala> -1 % -2
8 res2: Int = -1
9
10 scala> 1 % -2
11 res3: Int = 1
```

**Lösn. uppg. 33.** *Bokstavliga identifierare.*

- a) Variabeln får namnet 'bokstavlig val', bakåt-apostrofer (eng. *backticks*) gör att man kan namnge variabler till annars otillåtna namn, t.ex. med mellanrum eller nyckelord i sig.
- b) Backticks i Scala möjliggör alla möjliga tecken i namn. Exempel på användning: I java finns en metod som heter `java.lang.Thread.yield` men i Scala är `yield` ett nyckelord; för att komma runt det går det att i Scala skriva `java.lang.Thread.`yield``

**Lösn. uppg. 34.** *java.lang.Integer, hexadecimala litteraler, BigDecimal.*

a)

```
1 scala> import Integer.{toBinaryString => toBin, toHexString => toHex}
2
3 scala> for i <- Seq(33, 42, 64) do println(s"$i \t ${toBin(i)} \t ${toHex(i)}")
4 33    100001    21
5 42    101010    2a
6 64    1000000   40
```

- b) Det hexadecimala heltalet `10c` kan anges med litteralen `0x10c` i Scala, Java och många andra språk: <sup>1</sup>

```
1 scala> 0x10c
2 res0: Int = 268
```

c) <sup>2</sup>

```
1 scala> val c = 299792458
2 c: Int = 299792458
3
4 scala> BigDecimal(0x10).pow(c)
5 res68: scala.math.BigDecimal = 2.124892963227906613060986110887672E+360986089
```

**Lösn. uppg. 35.** *Strängformatering.*

```
val str = f"Jättegurkan är $g%1.3f meter lång"
```

(Om du tycker att `$g%1.3f` ser kryptiskt ut, så kan du trösta dig med att du nu får chansen att föra vidare ett anrikt arv från det urgamla språket C och den sägenomspunna funktionen `printf` till kommande generationer av invigda kodmagiker.)

**Lösn. uppg. 36.** *Multiplikationsvarning.*

- a) Den andra multiplikationen flödar över (eng. *overflow*) gränsen för största möjliga värdet av en `Int`. I den tredje multiplikationen kastas i stället ett undantag `java.lang.ArithmeticException`

```
scala> Math.multiplyExact(1, 2)
res70: Int = 2

scala> Int.MaxValue * 2
res71: Int = -2

scala> Math.multiplyExact(Int.MaxValue, 2)
java.lang.ArithmeticException: integer overflow
  at java.lang.Math.multiplyExact(Math.java:867)
  ... 42 elided
```

<sup>1</sup><https://en.wikipedia.org/wiki/0x10c>

<sup>2</sup><https://c418.bandcamp.com/album/0x10c>

b) Används då man vill vara helt säker på att overflow-buggar ”smäller” direkt i stället för att generera felaktiga resultat vars konsekvenser kanske manifesterar sig långt senare. Dock är `multiplyExact` aningen långsammare än vanlig multiplikation.

**Lösn. uppg. 37.** *Extra operatorer för exakt multiplikation.*

a)

```
1 scala> Int.MaxValue *! 1
2 res0: Int = 2147483647
3
4 scala> Int.MaxValue *! 2
5 java.lang.ArithmeticException: integer overflow
6   at java.lang.Math.multiplyExact(Math.java:867)
7   at IntExtra.$times$bang(<console>:16)
8   ... 32 elided
```

b)

```
extension (i: Int)
  def *!(j: Int) = Math.multiplyExact(i,j)
  def +!(j: Int) = Math.addExact(i,j)
  def -!(j: Int) = Math.subtractExact(i,j)
```

c) Det blir lätt väldigt kryptiskt med namn som består av flera specialtecken. Om du *verkligt* vill ha sådana operatorer är det *mycket* lämpligt att också erbjuda varianter i klartext:

```
extension (i: Int)
  def mulExact(j: Int) = Math.multiplyExact(i,j)
  def *!(j: Int) = i mulExact j

  def addExact(j: Int) = Math.addExact(i,j)
  def +!(j: Int) = i addExact j

  def subExact(j: Int) = Math.subtractExact(i,j)
  def -!(j: Int) = i subExact j
```



## L.2 Lösning programs

### L.2.1 Grunduppgifter

**Lösn. uppg. 1.** *Para ihop begrepp med beskrivning.*

kompilera	1	↪	I	maskinkod skapas ur en eller flera källkodsfiler
skript	2	↪	J	ensam kodfil, huvudprogram behövs ej
objekt	3	↪	G	samlar variabler och funktioner
@main	4	↪	L	där exekveringen av kompilerat program startar
programargument	5	↪	A	kan överföras via parametern args till main
datastruktur	6	↪	B	många olika element i en helhet; elementvis åtkomst
samling	7	↪	C	datastruktur med element av samma typ
sekvenssamling	8	↪	F	datastruktur med element i en viss ordning
Array	9	↪	K	en förändringsbar, indexerbar sekvenssamling
Vector	10	↪	E	en oföränderlig, indexerbar sekvenssamling
Range	11	↪	N	en samling som representerar ett intervall av heltal
yield	12	↪	O	används i for-uttryck för att skapa ny samling
map	13	↪	H	applicerar en funktion på varje element i en samling
algorithm	14	↪	M	stegvis beskrivning av en lösning på ett problem
implementation	15	↪	D	en specifik realisering av en algorithm

**Lösn. uppg. 2.** *Använda terminalen.*

a)

```
1 > mkdir hello
2 > cd hello
3 > pwd
```

b)

```
1 > cd ..
2 > ls
```

**Lösn. uppg. 3.** *Skapa och köra ett Scala-skript.*

a)

```
1 Summan av de 1000 första talen är: 500500
```

b) Kompileringsfelet blir: `' ) ' expected, but eof found`

c) Filen ska se ut så här:

```
val n = args(0).toInt
val summa = (1 to n).sum
println(s"Summan av de $n första talen är: $summa")
```

Utskriften blir så här:

1 Summan av de 5001 första talen är: 12507501

d) Körtidsfelet blir:

```
java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
```

Eftersom rrayen args är tom om programargument saknas så finns ej platsen med index 0.

### Lösn. uppg. 4. Scala-applikation med @main.

a) Kompilatorn har skapat 5 filer i underkataloger till .scala-build som heter:

```
'hello$package.class' 'hello$package$.class' 'hello$package.tasty'  
run.class run.tasty
```

b) Felmeddelandet får du om du tar bort den sista krullparentesen. eof i felmeddelandet står för end-of-file. Detta felmeddelande är vanligt vid oparade parenteser, men kompilatorn har ofta extra svårt att ge bra felmeddelande om en av parenteserna i ett parentespar saknas och det kan hända att den pekar ut felaktig rad för positionen där det som saknas borde stå.

c) Syntax Error: Expected a toplevel definition. Utan klammerparenteser så är det indenteringarna som bestämmer vilka delar av koden som hör samman. Om du tar bort indenteringen på den sista raden med utskrift-satsen så tolkar kompilatorn detta som att denna ligger *utanför* main-funktionen och du får ett felmeddelande eftersom det inte är tillåtet att ha ensamma satser på toppnivå. (Det går dock bra att ha ensamma satser i ett skript med .sc i slutet av namnet på kodfilen. )

d) Annoteringen @main berättar för kompilatorn att funktionen är ett huvudprogram kan utgöra en startpunkt för exekveringen.

Under huven skapar kompilatorn ett objekt med samma namn som ditt huvudprogram. I det objektet genererar kompilatorn i sin tur en metod med namnet main som tar en sträng-array som parameter och har returtypen Unit. Ett kompilerat program måste ha minst ett objekt med exakt en sådan main-metod eftersom exekveringsmiljön JVM förutsätter detta och anropar en sådan main-metoden med en sträng-array innehållande eventuella programargument när exekveringen startar.

Ett alternativ till @main är att definiera en s.k. *primitiv* main-metod i ett singelobjekt. (Detta är nödvändigt i gamla Scala 2, innan den enklare @main.annoteringen kom i Scala 3.)

```
object Hello:  
  def main(args: Array[String]): Unit =  
    val message = "Hello world!"  
    println(message)
```

### Lösn. uppg. 5. Skapa och använda samlingar.

<b>val</b> xs = Vector(2)	1	↪	I	ny referens till sekvens av längd 1
<b>val</b> ys = Array.fill(9)(0)	2	↪	C	ny referens till förändringsbar sekvens
Vector.fill(9>(' '))	3	↪	J	ny oföränderlig sekvens med blanktecken
xs(0)	4	↪	E	förkortad skrivning av apply(0)
xs.apply(0)	5	↪	F	indexering, ger första elementet
xs :+ 0	6	↪	A	ny samling med en nolla tillagd på slutet
0 +: xs	7	↪	H	ny samling med en nolla tillagd i början
ys.mkString	8	↪	K	ny sträng med alla element intill varandra
ys.mkString(",")	9	↪	G	ny sträng med komma mellan elementen
xs.map(_.toString)	10	↪	D	ny samling, elementen omgjorda till strängar
xs.map(_.toInt)	11	↪	B	ny samling, elementen omgjorda till heltal

### Lösn. uppg. 6. Jämför Array och Vector.

a)

Vector	1	↪	B	oföränderlig
Array	2	↪	A	förändringsbar

b)

Vector	1	↪	B	varianter med fler/andra element skapas snabbt ur befintlig
Array	2	↪	A	långsam vid ändring av storlek (kopiering av rubbet krävs)

c)

Vector	1	↪	A	xs == ys är <b>true</b> om alla element lika
Array	2	↪	B	olikt andra Scala-samlingar kollar == ej innehållslighet

### Lösn. uppg. 7. Räkna ut summa, min och max i args.

```
@main def sumMinMax(args: Int*): Unit =
  println(s"${args.sum} ${args.min} ${args.max}")
```

```
> scala-cli run sum-min-max.scala -- hej
Illegal command line: java.lang.NumberFormatException: For input string: "hej"
```

### Lösn. uppg. 8. Algoritm: SWAP.

a) Pseudokoden kan se ut såhär:

```
Deklarera heltalsvariabel temp.
```

```
Kopiera värdet från x till temp.
Kopiera värdet från y till x.
Kopiera värdet från temp till y.
```

b)

Du behöver deklarera en temporär variabel där du kan spara undan ett av värdena, så det inte skrivs över vid första tilldelningen.

```
val temp = x
x = y
y = temp
```

### Lösn. uppg. 9. Indexering och tilldelning i Array med SWAP.

```
@main def swapFirstLastArg(args: String*): Unit =
  val xs = args.toArray
  if xs.length > 1 then
    val temp = xs(0)
    xs(0) = xs(xs.length - 1)
    xs(xs.length - 1) = temp
    println(xs.mkString(" "))
```

### Lösn. uppg. 10. for-uttryck och map-uttryck.

for x <- xs yield x * 2	1	↪ A	Vector(2, 4, 6)
for i <- xs.indices yield i	2	↪ E	Vector(0, 1, 2)
xs.map(x => x + 1)	3	↪ D	Vector(2, 3, 4)
for i <- 0 to 1 yield xs(i)	4	↪ B	Vector(1, 2)
(1 to 3).map(i => i)	5	↪ C	Vector(1, 2, 3)
(1 until 3).map(i => xs(i))	6	↪ F	Vector(2, 3)

### Lösn. uppg. 11. Algoritm: SUMBUG

a) Bugg: Eftersom i inte inkrementeras, fastnar programmet i en oändlig loop. Fix: Lägg till en sats i slutet av while-blocket som ökar värdet på i med 1. Bugg: Eftersom man bara ökar summan med 1 varje gång, kommer resultatet att bli summan av n stycken 1or, inte de n första heltalen. Fix: Ändra så att summan ökar med i varje gång, istället för 1. För -1, blir resultatet 0. Förklaring: i börjar på 1 och är alltså aldrig mindre än n som ju är -1. while-blocket genomförs alltså noll gånger, och efter att sum får sitt ursprungsvärde förändras den aldrig.

b) Summan blir 39502716.

Såhär kan en implementation se ut:

```
@main def sumn(n: Int): Unit =
  var sum = 0
  var i = 1
  while i <= n do
```

```
    sum = sum + i  
    i = i + 1  
println(sum)
```

## L.2.2 Extrauppgifter; träna mer

### Lösn. uppg. 12. Algoritm: MAXBUG

a) Bugg: `i` inkrementeras aldrig. Programmet fastnar i en oändlig loop. Fix: Lägg till en sats som ökar `i` med 1, i slutet av `while`-blocket.

b) Så här kan implementationen se ut:

```
@main def maxn(args: String*): Unit =
  var max = Int.MinValue
  val n = args.length
  var i = 0
  while i < n do
    val x = args(i).toInt
    if x > max then
      max = x
    i += 1
  println(max)
```

c) Raden där `max` initieras ändras till `var max = args(0).toInt`

d) För att inte få `java.lang.IndexOutOfBoundsException`: 0 behövs en kontroll som säkerställer att inget görs om samlingen `args` är tom:

```
@main def maxn(args: String*): Unit =
  if args.size > 0 then
    var max = args(0).toInt
    val n = args.size
    var i = 0
    while i < n do
      val x = args(i).toInt
      if x > max then
        max = x
      i += 1
    println(max)
  else
    println("Empty")
```

### Lösn. uppg. 13. Algoritm MIN-INDEX.

a) En onödig jämförelse sker, men resultatet påverkas ej.

b)

```
def indexOfMin(xs: Array[Int]): Int =
  var minPos = 0
  var i = 1
  while i < xs.size do
    if xs(i) < xs(minPos) then
      minPos = i
    i += 1
  if xs.size > 0 then minPos else -1
```

**Lösn. uppg. 14.** *Datastrukturen Range.*

- a) värde: `Range(1,2,3,4,5,6,7,8,9)`  
typ: `scala.collection.immutable.Range`
- b) värde: `Range(1,2,3,4,5,6,7,8,9,10)`  
typ: `scala.collection.immutable.Range`
- c) värde: `Range(0,5,10,15,20,25,30,35,40,45)`  
typ: `scala.collection.immutable.Range`
- d) värde: 10, typ: `Int`
- e) värde: `Range(0,5,10,15,20,25,30,35,40,45,50)`  
typ: `scala.collection.immutable.Range`
- f) värde: 11, typ: `Int`
- g) värde: `Range(0,1,2,3,4,5,6,7,8,9)`  
typ: `scala.collection.immutable.Range`
- h) värde: `Range(0,1,2,3,4,5,6,7,8,9)`  
typ: `scala.collection.immutable.Range`
- i) värde: `Range(0,1,2,3,4,5,6,7,8,9)`  
typ: `scala.collection.immutable.Range`
- j) värde: `Range(0,1,2,3,4,5,6,7,8,9,10)`  
typ: `scala.collection.immutable.Range.Inclusive`
- k) värde: `Range(0,1,2,3,4,5,6,7,8,9,10)`  
typ: `scala.collection.immutable.Range.Inclusive`
- l) värde: `Range(0,5,10,15,20,25,30,35,40,45)`  
typ: `scala.collection.immutable.Range`
- m) värde: `Range(0,5,10,15,20,25,30,35,40,45,50)`  
typ: `scala.collection.immutable.Range`
- n) värde: 11, typ: `Int`
- o) värde: 500500, typ: `Int`

### L.2.3 Fördjupningsuppgifter; utmaningar

**Lösn. uppg. 15.** *Sten-Sax-Påse-spel.* En (lättbegriplig?) lösning som provar alla kombinationer:

```
def winner(user: Int, computer: Int): String =
  if choices(user) == "Sten" && choices(computer) == "Påse" then "Datorn"
  else if choices(user) == "Sten" && choices(computer) == "Sax" then "Du"
  else if choices(user) == "Påse" && choices(computer) == "Sten" then "Du"
  else if choices(user) == "Påse" && choices(computer) == "Sax" then "Datorn"
  else if choices(user) == "Sax" && choices(computer) == "Sten" then "Datorn"
  else if choices(user) == "Sax" && choices(computer) == "Påse" then "Du"
  else "Ingen"
```

En klurigare lösning (och svårbegripligare?) med hjälp av modulo-räkning:

```
def winner(user: Int, computer: Int): String =
  val result = (user - computer + 3) % 3
  if user == computer then "Ingen"
  else if result == 1 then "Du"
  else "Datorn"
```

Moduloräkningen kräver att elementen i choices är i *förlorar-över*-ordning, alltså Sten, Påse, Sax. Addition med 3 görs för att undvika negativa tal, som beter sig annorlunda i moduloräkning.

**Lösn. uppg. 16.** *Jämför exekveringstiden för storleksförändring mellan Array och Vector.*

a) Med en dator som har en i7-4790K CPU @ 4.00GHz blev det så här:

```
1 scala> def time(block: => Unit): Double =
2   |   val t = System.nanoTime
3   |   block
4   |   (System.nanoTime - t)/1e6 // ger millisekunder
5 def time(block: => Unit): Double
6
7 scala> val as = Array.fill(1e6.toInt)(0)
8 val as: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
9 large output truncated, print value to show all
10
11 scala> val vs = Vector.fill(1e6.toInt)(0)
12 val vs: Vector[Int] = Vector(0, 0, 0, 0, 0, ...
13 large output truncated, print value to show all
14
15 scala> val ast = (for i <- 1 to 10 yield time(as :+ 0)).sum / 10.0
16 val ast: Double = 1.8719819999999998
17
18 scala> val vst = (for i <- 1 to 10 yield time(vs :+ 0)).sum / 10.0
19 val vst: Double = 0.006485099999999999
20
21 scala> ast / vst
22 val res3: Double = 288.6589258453995
```

b) Vector är två tiopotenser snabbare i detta exempel. Anledningen är att varje storleksförändring av en Array kräver allokering och elementvis kopiering av en helt ny Array medan den oföränderliga Vector kan återanvända hela datastrukturen med redan allokerade element när nya element läggs till.



**Lösn. uppg. 17.** *Minnesåtgång för Range.*

- a) Variabeln intervall refererar till objekt som tar upp 12 bytes.
- b) Variabeln sekvens refererar till objekt som tar upp ca 4 miljarder bytes.

**Lösn. uppg. 18.** *Undersök den genererade byte-koden.*

- a) Så här ser funktionen plusxy ut:

```

1 public int plusxy(int, int);
2   descriptor: (II)I
3   flags: (0x0001) ACC_PUBLIC
4   Code:
5     stack=2, locals=3, args_size=3
6       0: iload_1
7       1: iload_2
8       2: iadd
9       3: ireturn
10  LineNumberTable:
11    line 2: 0
12  LocalVariableTable:
13    Start  Length  Slot  Name   Signature
14         0       4     0  this  Lplusxy$package$;
15         0       4     1    x    I
16         0       4     2    y    I

```

Det är instruktionen iadd som gör själva additionen.

- b) Det har tillkommit en parameter till i byte-koden. Instruktionen iadd görs nu två gånger. Instruktionen iadd adderar exakt två tal i taget.

```

1 public int plusxyz(int, int, int);
2   descriptor: (III)I
3   flags: (0x0001) ACC_PUBLIC
4   Code:
5     stack=2, locals=4, args_size=4
6       0: iload_1
7       1: iload_2
8       2: iadd
9       3: iload_3
10      4: iadd
11      5: ireturn
12  LineNumberTable:
13    line 2: 0
14  LocalVariableTable:
15    Start  Length  Slot  Name   Signature
16         0       6     0  this  Lplusxyz$package$;
17         0       6     1    x    I
18         0       6     2    y    I
19         0       6     3    z    I

```

- c) Prefixet i i instruktionsnamnet iadd står för ”integer” och anger att heltalsdivision avses.

## L.3 Lösning functions

**Lösn. uppg. 1.** Para ihop begrepp med beskrivning.

funktionshuvud	1	↪ D	har parameterlista och eventuellt en returtyp
funktionskropp	2	↪ G	koden som exekveras vid funktionsanrop
parameterlista	3	↪ I	beskriver namn och typ på parametrar
block	4	↪ M	kan ha lokala namn; sista raden ger värdet
namngivna argument	5	↪ N	gör att argument kan ges i valfri ordning
defaultargument	6	↪ A	gör att argument kan utelämnas
värdeanrop	7	↪ L	argumentet evalueras innan anrop
namnanrop	8	↪ E	fördröjd evaluering av argument
äkta funktion	9	↪ C	ger alltid samma resultat om samma argument
predikat	10	↪ J	en funktion som ger ett booleskt värde
slumptalsfrö	11	↪ F	ger återupprepningsbar sekvens av pseudoslumptal
anonym funktion	12	↪ B	funktion utan namn; kallas även lambda
rekursiv funktion	13	↪ K	en funktion som anropar sig själv
stack trace	14	↪ H	lista anropskedja vid körtidsfel

**Lösn. uppg. 2.** Definiera och anropa funktioner.

a)

```
def öka(x: Int = 1): Int = x + 1
```

b) 5

c)

```
def minska(x: Int = 1): Int = x - 1
```

d) 1

e)

- *Kort, förenklad förklaring:* Parametern i funktionshuvudet är ett lokalt namn på in-data som kan användas i funktionskroppen, medan argumentet är själva värdet på parametern som skickas med vid anrop.
- *Längre, mer exakt förklaring:* En **parameter** är en deklaration av en oföränderlig variabel i ett funktionshuvud vars namn finns tillgängligt lokalt i funktionskroppen. Vid anrop *binds* parameternamnet till ett specifikt argument. Ett **argument** är ett uttryck som appliceras på en funktion vid anrop. Normalt evalueras argumentet innan anropet sker, men om parametertypen föregås av => fördröjs evalueringen av argumentet och sker i stället *varje gång* parameternamnet förekommer i funktionskroppen.

**Lösn. uppg. 3.** Textspelet AliensOnEarth.

a) "penguin" är bästa alternativ med sannolikheten  $\frac{1}{2} + \frac{1}{2} \cdot \frac{1}{3} = \frac{2}{3}$

b)

options.indices	1	↪ B	heltalssekvens med alla index i en sekvens
"1X2".toLowerCase	2	↪ A	gör om en sträng till små bokstäver
Random.nextInt(n)	3	↪ C	slumptal i intervallet 0 until n
try { } catch { }	4	↪ E	fångar undantag för att förhindra krasch
""" ... """	5	↪ F	sträng som kan sträcka sig över flera kodrader
s.stripMargin	6	↪ D	tar bort marginal till och med vertikalstreck
e.printStackTrace	7	↪ G	skriver ut information om ett undantag

### Lösn. uppg. 4. Äkta funktioner.

- Funktionerna inc, addY och isPalindrome är äkta. Notera att y-variablen initialiseras till 0 och kan sedan inte ändras eftersom den är deklarerad med nyckelordet **val**.

### Lösn. uppg. 5. Applicera funktion på varje element i en samling. Funktion som argument.

for i <- 1 to 3 yield öka(i)	1	↪ E	Vector(2, 3, 4)
Vector(2, 3, 4).map(i => öka(i))	2	↪ A	xs
xs.map(öka)	3	↪ B	Vector(4, 5, 6)
xs.map(öka).map(öka)	4	↪ D	Vector(5, 6, 7)
xs.foreach(öka)	5	↪ C	()

### Lösn. uppg. 6. Funktion som äkta värde.

a)

fleraAnrop(1, hälsa)	1	↪ D	f2("Hej!")
fleraAnrop(3, hälsa)	2	↪ B	fleraAnrop(3, f1)
fleraAnrop(2, f1)	3	↪ A	f2("Hej!\nHej!")
fleraAnrop(1, f3)	4	↪ C	f3()

b) f1 och f3 är av typen () => Unit och f2 av typen String => Unit.

c) Nej. f1 och f2 är av två olika funktionstyper.

d) Ja, det går fint.

e) Nej. När funktionen inte har någon parameter behöver kompilatorn mer information för att vara säker på att det är ett funktionsvärde du vill ha.

f) Ja! Nu med typinformationen på plats är kompilatorn säker på vad du vill göra.

**Lösn. uppg. 7.** *Anonyma funktioner.*

(0 to 2).map(i => i + 1)	1	↪ B	(2 to 4).map(i => i - 1)
(1 to 3).map(_ + 1)	2	↪ D	Vector(2, 3, 4)
(2 to 4).map(math.pow(2, _))	3	↪ E	Vector(4.0, 8.0, 16.0)
(3 to 5).map(math.pow(_, 2))	4	↪ A	Vector(9.0, 16.0, 25.0)
(4 to 6).map(_.toDouble).map(_ / 2)	5	↪ C	Vector(2.0, 2.5, 3.0)

**Lösn. uppg. 8.** *Lär dig läsa en stack trace.* En stack trace innehåller följande information:

1. ett felmeddelande
2. namn på alla funktioner som anropats vid tiden för körtidsfelet, enligt alla aktiveringsposter som ligger på anropsstacken
3. aktuell namnrymnd för varje funktionen, alltså paket/singelobjekt
4. namnet på kodfilen för varje funktion
5. radnummer i varje funktion
6. den funktion som kommer först är den funktion där felet inträffade
7. eventuellt kan felet inträffa i standardbibliotekets funktioner och då är din egen funktion tidigare i anropskedjan

Exempel på en stack trace:

```
> cat fel.scala
@main def run =
  println("Hej Scala!" + Vector().head)
> scala-cli run fel.scala
Compiling project (Scala 3.3.0, JVM)
Compiled project (Scala 3.3.0, JVM)
Exception in thread "main" java.util.NoSuchElementException: empty.head
  at scala.collection.immutable.Vector.head(Vector.scala:279)
  at fel$package$.run(fel.scala:2)
  at run.main(fel.scala:1)
>
```

**L.3.1 Extrauppgifter; träna mer****Lösn. uppg. 9.** *Funktion med flera parametrar.*

- a) -100
- b) 15
- c) 185
- d) 256

**Lösn. uppg. 10.** *Medelvärde.*

```
def avg(x: Int, y: Int): Double = (x + y) / 2.0
```

**Lösn. uppg. 11.** *Funktionsanrop med namngivna argument.*

a)

```
1 Namn: Triangelsson, Stina
2 Namn: Oval, Viktor
```

b)

- Anroparen kan själv välja ordning.
- Koden blir lättare att begripa om parameternamnen är självbeskrivande.
- Hjälper till att förhindra buggar som beror på förväxlade parametrar.

**Lösn. uppg. 12.** *Bortkastade resultatvärden och returtypen Unit.*

- a) Procedurer returnerar tomta värdet och `println` är en procedur. När tomta värdet skrivs ut visas `()`.
- b) Procedurer returnerar tomta värdet. Om du anger returtyp `Unit` explicit, har du bättre chans att kompilatorn kan ge varning då uträkningar kommer att kastas bort. En varning avbryter inte exekveringen, utan är ett sätt för kompilatorn att ge dig tips om saker som kan behöva fixas till i din kod.
- c) I Scala är variabeldeklaration, precis som en tilldelningssats, och inte ett uttryck och saknar värde.
- d) Koden blir lättare att läsa och kompilatorn får bättre möjlighet att hjälpa till med varningar om resultatvärden riskerar att bli bortkastade.

### L.3.2 Fördjupningsuppgifter; utmaningar

#### Lösn. uppg. 13. Föränderlighet av parametrar.

a) Nej, i Scala är parametern oföränderlig och det blir kompileringsfel om man försöker tilldela den ett nytt värde i funktionskroppen.

b) c) Ja det går utmärkt i både Java och Python att ändra värdet på parametern i funktionskroppen med tilldelning, men koden riskerar att bli förvirrande.

<https://stackoverflow.com/questions/2970984>

#### Lösn. uppg. 14. Värdeanrop och namnanrop.

a) Vid varje anrop av snark sker en utskrift och en fördröjning innan 42 returneras.  $42 + 42 == 84$  vilket blir värdet av uttrycket.

```
1 scala> snark + snark
2 snark snark val res1: Int = 84
```

b) Uttrycket snark evalueras direkt vid anropet och parametern x binds till värdet 42 och i funktionskroppen beräknas  $42 + 42$ . Utskriften sker bara en gång.

```
1 callByValue(snark)
2 snark val res2: Int = 84
```

c) Evalueringen av uttrycket snark fördröjs tills varje förekomst av parametern x i funktionskroppen. Utskriften sker två gånger.

```
1 callByName(snark)
2 snark snark val res3: Int = 84
```

d) Evalueringen av uttrycket zzz fördröjs tills varje förekomst av parametern x i funktionskroppen. Utskriften sker en gång eftersom **val**-variabler tilldelas sitt värde en gång för alla vid den fördröjda initialiseringen.

```
1 callByName(zzz)
2 snark val res4: Int = 84
```

#### Lösn. uppg. 15. Skapa din egen kontrollstruktur med hjälp av namnanrop.

a) Blocket är ett uttryck som har värdet `()`: Unit. Evalueringen av blocket sker där namnet b förekommer i procedurkroppen, vilket är två gånger.

```
1 scala> görDettaTvåGånger { println("goddag") }
2 goddag
3 goddag
```

b)

```
def upprepa(n: Int)(block: => Unit): Unit =
  var i = 0
  while i < n do
    block
    i += 1
```

c)

```
upprepa(100):
  val tärningskast = (math.random() * 6 + 1).toInt
  print(s"\$tärningskast ")
```

d)

```
def repeat(n: Int)(p: Int => Unit): Unit =
  var i = 0
  while i < n do
    p(i)
    i += 1
  end while
end repeat
```

e)

```
repeat(100){ i =>
  print(s"$i: ")
  println(math.random())
}
```

Du kan använda färre klammerparenteser med hjälp av kolon:

```
repeat(100): i =>
  print(s"$i: ")
  println(math.random())
```

### Lösn. uppg. 16. Uppdelad parameterlista och stegade funktioner.

a)

```
1 scala> def add2(a: Int)(b: Int) = a + b
2 def add2(a: Int)(b: Int): Int
3
4 scala> add2(1)(1)
5 val res0: Int = 2
```

b)

- Rad 3:

```
doremi doremi doremi
```

- Rad 5:

```
lalalalalalala
```

### Lösn. uppg. 17. Rekursion.

a) countdown skriver ut x och gör ett rekursivt anrop med  $x - 1$  som argument, men bara om basvillkoret  $x > 0$  är uppfyllt. Resultatet blir en ändlig repetition. finalCountdown anropar sig själv rekursivt men saknar ett basvillkor som kan avbryta rekursionen, vilket

genererar en oändlig repetition. Vid -128 blir det *overflow* eftersom bitarna inte räcker till för större negativa tal och räkningen börjar om på 127. (Om minskar fördröjningen till `Thread.sleep(1)` blir det ganska snabbt *stack overflow*)

b) Eftersom vi hade  $1/x$  *efter* det rekursiva anropet i föregående deluppgift, så kom vi aldrig till denna (potentiellt ödesdigra) beräkning, utan lade bara aktiveringsposter på hög på stacken vid varje anrop. Om vi placerar  $1/x$  *före* det rekursiva anropet, så når vi detta uttryck direkt och det kastas ett undantag p.g.a. division med noll.

c) Den sista raden leder till många fler rekursiva anrop, så som basvillkoret och det rekursiva anropet är konstruerade. Lägg gärna in en `println`-sats före det rekursiva anropet och undersök i detalj vad som sker.

**Lösn. uppg. 18.** *Undersök svansrekursion genom att kasta undantag.* `countdown` är svansrekursiv eftersom det rekursiva anropet står *sist* och kan då optimeras till en **while**-loop av kompilatorn. Det går fint att köra ända till det exploderar, även med 10000 anrop, och i felmeddelandet finns det endast ett anrop till `countdown`.

`countdown2` är inte svansrekursiv eftersom den har ett uttryck efter det rekursiva anropet. I felutskriften syns alla rekursiva anrop till `countdown2` innan basvillkoret inträffade. Vid `countdown2(10000)` uppfylls inte basvillkoret innan det blir `StackOverflowError`.

**Lösn. uppg. 19.** *@tailrec-annotering.* Första gången `countNoTailrec(100000L)` anropas blir det `StackOverflowError`. Med annoteringen `@tailrec` får vi ett kompileringsfel eftersom kompilatorn inte kan optimera en icke svansrekursiv funktion. Om funktionen skrivs om kan kompilatorn optimera funktionen så att rekursionen byts ut mot en **while**-loop och vi kan köra så länge vi orkar utan att stacken flödar över. Och himla snabbt går det!!



## L.4 Lösning objects

### L.4.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** Para ihop begrepp med beskrivning.

modul	1	↔	C	kodenhet med abstraktioner som kan återanvändas
singelobjekt	2	↔	B	modul som kan ha tillstånd; finns i en enda upplaga
paket	3	↔	D	modul som skapar namnrymd; maskinkod får egen katalog
import	4	↔	F	gör namn tillgängligt lokalt utan att hela sökvägen behövs
export	5	↔	P	gör namn synligt utåt som medlem i detta objekt
lat initialisering	6	↔	G	allokering sker först när namnet refereras
medlem	7	↔	E	tillhör ett objekt; nås med punktnotation om synlig
attribut	8	↔	H	variabel som utgör (del av) ett objekts tillstånd
metod	9	↔	A	funktion som är medlem av ett objekt
privat	10	↔	K	modifierar synligheten av en objektmedlem
överlagring	11	↔	J	metoder med samma namn men olika parametertyper
namnskuggning	12	↔	L	lokalt namn döljer samma namn i omgivande block
namnrymd	13	↔	I	omgivning där är alla namn är unika
enhetlig access	14	↔	M	ändring mellan def och val påverkar ej användning
punktnotation	15	↔	O	används för att komma åt icke-privata delar
typalias	16	↔	N	alternativt namn på typ som ofta ökar läsbarheten

**Lösn. uppg. 2.** Nästlade singelobjekt, import, synlighet och punktnotation.

a)

**object** Underjorden:

```
var x = 0
```

```
var y = 1
```

**object** Mullvaden:

```
var x = Underjorden.x + 10
```

```
var y = Underjorden.y + 9
```

**object** Masken:

```
private var x = Mullvaden.x
```

```
var y = Mullvaden.y + 190
```

```
def ärMullvadsmat: Boolean = x == Mullvaden.x && y == Mullvaden.y
```

b)

```
1 scala> :load Underjorden.scala
2 scala> import Underjorden.*
3 scala> Masken.ärMullvadsmat
4 val res0: Boolean = false
5 scala> Masken.y = Mullvaden.y
```

```
6 scala> Masken.ärMullvadsmat
7 val res1: Boolean = true
```

c)

```
1 scala> import Mullvaden.*
2 scala> import Masken.*
3 scala> x = -1
4 scala> Mullvaden.x
5 val res2: Int = -1
6
7 scala> Masken.x
8 1 |Masken.x
9   |^^^^^^
10  |variable x cannot be accessed as a member of Underjorden.Masken.type from module class rs\line\
11
12 scala> Underjorden.x
13 val res3: Int = 0
```

*Förklaring:* När importen av Maskens alla synliga medlemmar sker kommer de som ej är privata att överskugga andra medlemmar med samma namn. Det är Mullvadens x-variabel som tilldelas -1 eftersom Maskens x är privat och ej syns utåt. Underjordens medlemmar blir överskuggade av Maskens y och Mullvadens x men man kan komma åt dem genom att använda punktnotation.

### Lösn. uppg. 3. Export.

- a) Likhet: Både **import** och **export** styr synlighet. Skillnad: **import** styr lokal synlighet *inuti* ett objekt medan **export** styr synlighet *utanför* ett objekt.
- b) Man kan med **export** på ett smidigt sätt plocka ihop medlemmar från andra objekt och göra dem synliga från mitt eget objekt.

```
object MittObjekt:
  export java.awt.Color.* // alla färger blir medlemmar i MittObjekt
  export math.{atan2, Pi} // atan2 och Pi blir medlemmar i MittObjekt
```

```
scala> object MittObjekt:
  |   export java.awt.Color.*
  |   export math.{atan2, Pi}
  |
scala> MittObjekt.RED
val res0: java.awt.Color = java.awt.Color[r=255,g=0,b=0]

scala> MittObjekt.atan2(3,3) / MittObjekt.Pi
val res1: Double = 0.25
```

### Lösn. uppg. 4. Tupler.

- a) djup har typen Double.
- b) hemlis har typen (String, (Int, Int, Double)).
- c)

```
object Underjorden3D:
  private val hemlis = ("uppgången till överjorden", (3, 4, 0.0))

object Mullvaden:
```

```

var pos = (5, 3, math.random() * 10 + 1)

def djup: Double = pos._3

object Masken:
  private var pos = (0, 0, 10.0)

  def ärMullvadsmat: Boolean = pos == Mullvaden.pos

  def ärRaktUnderUppgången: Boolean =
    pos._1 == hemlis._2._1 && pos._2 == hemlis._2._2

```

d) Noll-tupeln.

**Lösn. uppg. 5.** *Lat initialisering.*

a) "nu!" skrivs bara ut första gången z används.

```

1 scala> z
2 nu!
3 val res19: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
4
5 scala> z
6 val res20: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

```

b) Allokeringen av arrayen sker första gången z används (och inte vid deklarationen).

```

1 scala> lazy val z = { println("nu!"); Array.fill(1e9.toInt)(0) }
2 val z: Array[Int] = <lazy>
3
4 scala> z
5 nu!
6 java.lang.OutOfMemoryError: Java heap space

```

c) Utskriften av "nu!" sker först när singelobjektet z används för första gången. Vi borde lägga initialiseringen av b före a eller göra a till en **lazy val**.

d)

```

1 scala> import test.*
2 import test.*
3
4 scala> zzz.a // först när vi använder zzz skrivs "nu!"
5 nu! // detta skedde *inte* när vi importerade test
6 val res0: Int = 42
7
8 scala> buggig.a // a blir 0 eftersom b inte är initialiserad
9 val res1: Int = 0
10
11 scala> funkar.a // med lazy val undviker vi problemet
12 val res2: Int = 42
13
14
15 scala> zzz.a // andra gången är init redan gjort och ingen "nu!"
16 val res3: Int = 42

```

e) **lazy val** a = uttryck innebär att initialiseringsuttrycket evalueras *en* gång, men evalueringen skjuts på framtiden tills det eventuellt händer att namnet a används, medan

**def** b = uttryck innebär att funktionskroppens uttryck evalueras *varje gång* namnet b (eventuellt) används.

### Lösn. uppg. 6. *Extensionsmetoder.*

a)

```
scala> extension (i: Int) def inc = i + 1
```

b)

```
scala> extension (i: Int) def dec = i - 1
```

c)

```
extension (i: Int)
  def inc = math.incrementExact(i)
  def dec = math.decrementExact(i)
```

d) Med `math.incrementExact` och `math.decrementExact` ges exception om vi går över gränsen:

```
scala> math.incrementExact(Int.MaxValue)
java.lang.ArithmeticException: integer overflow
  at java.base/java.lang.Math.incrementExact(Math.java:1023)
  at scala.math.incrementExact(package.scala:418)
  ... 34 elided
```

### Lösn. uppg. 7. *Extensionsmetoder.*

a) Enligt dokumentationen har `PixelWindow`-klassen dessa parametrar:

- `width` : `Int` anger fönstrets bredd, defaultargument 800
- `height`: `Int` anger fönstrets höjd, defaultargument 640
- `title` : `String` anger fönstrets titel, defaultargument "PixelWindow"
- `background`: `Color` anger bakgrundsfärg, defaultargument `java.awt.Color.black`
- `foreground`: `Color` anger bakgrundsfärg, defaultargument `java.awt.Color.green`

Man kan skapa nya fönsterinstanser till exempel så här:

```
val w1 = new introprog.PixelWindow()
val w2 = new introprog.PixelWindow(100, 200, "Mitt fina nya fönster")
```

b) Du kan även ladda ner senaste `introprog` så här:

```
curl -o introprog_3-1.3.1.jar -sLO https://fileadmin.cs.lth.se/introprog.jar
```

c)

```
1 > scala repl --jar introprog_3-1.3.1.jar
2 scala> val w = new introprog.PixelWindow(400,300,"HEJ")
3 scala> w.line(100, 100, 200, 100)
4 scala> w.line(200, 100, 200, 200)
5 scala> w.line(200, 200, 100, 200)
6 scala> w.line(100, 200, 100, 100)
```

d)

```

package hello

object Main:
  val w = new introprog.PixelWindow(400, 300, "HEJ")

  var color = java.awt.Color.red

  def square(p: (Int, Int))(side: Int): Unit =
    if side > 0 then
      // side == 1 ger en kvadrat som är en enda pixel
      val d = side - 1

      w.line(p._1,      p._2,      p._1 + d, p._2,      color)
      w.line(p._1 + d, p._2,      p._1 + d, p._2 + d, color)
      w.line(p._1 + d, p._2 + d, p._1,      p._2 + d, color)
      w.line(p._1,      p._2 + d, p._1,      p._2,      color)

  def main(args: Array[String]): Unit =
    println("Rita kvadrat:")
    square(300,100)(50)

```

e)

```
> scala-cli run hello-window.scala --jar introprog_3-1.3.1.jar --main-class hello.Main
```

f)

```
> scala-cli run hello-window.scala --dep se.lth.cs::introprog:1.3.1 --main-class hello.Main
```

g)

```

//> using scala 3.3
//> using lib se.lth.cs::introprog:1.3.1

```

### Lösn. uppg. 8. Färg.

a)

```

object Color:
  import java.awt.{Color as JColor}

  val mole    = new JColor( 51,  51,  0)
  val soil    = new JColor(153, 102, 51)
  val tunnel  = new JColor(204, 153, 102)

```

b)

```

package hello

object Color:
  import java.awt.{Color as JColor}

```

```

val mole   = new JColor( 51, 51,   0)
val soil   = new JColor(153, 102, 51)
val tunnel = new JColor(204, 153, 102)

object Main:
  val w = new introprog.PixelWindow(width = 400, height = 300, title = "HEJ")

  type Pt = (Int, Int)

  var color = java.awt.Color.red

  def rak(p: Pt)(d: Int) = w.line(p._1, p._2, p._1 + d - 1, p._2, color)

  def fyll(p: Pt)(s: Int) = for i <- 0 until s do rak((p._1, p._2 + i))(s)

  def square(p: (Int, Int))(side: Int): Unit =
    if (side > 0) then
      val d = side - 1 // side == 1 ska ge en kvadrat som är en pixel stor
      w.line(p._1,      p._2,      p._1 + d, p._2,      color)
      w.line(p._1 + d, p._2,      p._1 + d, p._2 + d, color)
      w.line(p._1 + d, p._2 + d, p._1,      p._2 + d, color)
      w.line(p._1,      p._2 + d, p._1,      p._2,      color)

  def main(args: Array[String]): Unit =
    import Color.*
    color = soil
    fyll(100,100)(75)
    color = tunnel
    fyll(100,100)(50)
    color = mole
    fyll(150,150)(25)

```

c) Vid anropen av `rak` och `fyll` utnyttjas att man kan skippa tupelparenteserna om ett tupelargument är ensamt i sin parameterlista.

### Lösn. uppg. 9. Händelser.

a) Den oföränderliga heltalsvariabeln `KeyPressed` i `introprog.PixelWindow.Event` har värdet 1.

b) Kodraden nedan tar hand om knappnedtryckningsfallet:

```
case PixelWindow.Event.KeyPressed => println(s"lastKey == \${w.lastKey}")
```

c) När pil-upp-knappen på tangentbordet trycks ned får `w.lastKey` strängvärdet "Up". Följande skrivs ut av testprogrammet när pil-upp-tangenten trycks ned och släpps upp:

```

1 lastEventType: 1 => KeyPressed
2 lastKey == Up
3 lastEventType: 2 => KeyReleased
4 lastKey == Up

```

d) En loop som låter användaren rita linjer med musen:

```

var start = (0,0)
while w.lastEventType != PixelWindow.Event.WindowClosed do
  w.awaitEvent(10) // wait for next event for max 10 milliseconds
  w.lastEventType match {
    case PixelWindow.Event.MousePressed =>

```

```
    start = w.lastMousePos

    case PixelWindow.Event.MouseReleased =>
      w.line(start._1, start._2, w.lastMousePos._1, w.lastMousePos._2)

    case PixelWindow.Event.WindowClosed =>
      println("Goodbye!");
    case _ =>
  }
  PixelWindow.delay(100) // wait for 0.1 seconds
```

## L.4.2 Extrauppgifter; träna mer

**Lösn. uppg. 10.** *Funktioner är objekt med en apply-metod. Ja det går bra att skriva:*

```
1 scala> plus(42, 43)
```

Kompilatorn fyller i .apply åt dig.

**Lösn. uppg. 11.** *Skapa moduler med hjälp av singelobjekt.*

a)

```
scala> "päronisglass".split('i')
val res0: Array[String] = Array(päron, sglass)
```

b)

```
scala> Test()
--- FREKVENSANALYS AV:
Fem      myror är fler än fyra elefanter. Ät gurka.
# bokstäver: 36
# ord      : 9
# meningar : 2

--- FREKVENSANALYS AV:
Galaxer i mina braxer. Tomat är gott. Päronsplitt.
# bokstäver: 40
# ord      : 8
# meningar : 3

--- FREKVENSANALYS AV:
Fem      myror är fler än fyra elefanter. Ät gurka. Galaxer i mina braxer. Tomat
är gott. Päronsplitt.
# bokstäver: 76
# ord      : 17
# meningar : 5
```

c) Objektet statistics har ett förändringsbart tillstånd i variabeln history. Tillståndet ändras vid anrop av printFreq.

d)

```
object count:
  extension (s: String)
    def nbrOfLetters: Int = s.count(_.isLetter)
    def nbrOfWords: Int = split.words(s).size
    def nbrOfSentences: Int = split.sentences(s).size
```

**Lösn. uppg. 12.** *Tupler som parametrar.*

```
def distxy(x1: Int, y1: Int, x2: Int, y2: Int): Double =
  hypot(x1 - x2, y1 - y2)

def distpt(p1: (Int, Int), p2: (Int, Int)): Double =
  hypot(p1._1 - p2._1, p1._2 - p2._2)

def distp(p1: (Int, Int))(p2: (Int, Int)): Double =
```



```
hypot(p1._1 - p2._1, p1._2 - p2._2)
```

### Lösn. uppg. 13. Tupler som funktionsresultat.

```
def statistics(xs: Vector[Double]): (Int, Double, (Double, Double)) =
  (xs.size, xs.sum / xs.size, (xs.min, xs.max))
```

```
1 scala> statistics(Vector(0, 2.5, 5))
2 val res10: (Int, Double, (Double, Double)) = (3,2.5,(0.0,5.0))
```

### Lösn. uppg. 14. Skapa moduler med hjälp av paket.

a)

```
1 > code paket.scala
2 > scala-cli paket.scala
3 > find . -type d          # linuxkommando som listar alla subkataloger
4 ./scala-build/project_103be31561-3d0d386400/classes
5 ./scala-build/project_103be31561-3d0d386400/classes/main
6 ./scala-build/project_103be31561-3d0d386400/classes/main/gurka
7 ./scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat
8 ./scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan
9 ./scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p2
10 ./scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p2/p21
11 ./scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p1
12 ./scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p1/p12
13 ./scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p1/p11
```

b)

```
1 > scala-cli run paket.scala --main-class gurka.tomat.banan.Main
2 Hej paket p1.p11!
3 Hej paket p1.p12!
4 Hej paket p2.p21!
```

c) Ja, i Scala 3 får paket ha variabler och funktioner på toppnivå.

<https://stackoverflow.com/a/56566166>

## L.4.3 Fördjupningsuppgifter; utmaningar

### Lösn. uppg. 15. Hur klara sig utan **do** **while** i Scala 3?

a) Det blir kompileringsfel:

```
> scala-cli repl --scala 3
Welcome to Scala 3.1.3 (17.0.3, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> var i = 0
var i: Int = 0

scala> do i += 1 while (i < 10)
-- [E103] Syntax Error: -----
1 |do i += 1 while (i < 10)
  |^^
  |Illegal start of statement
```

b)

```
> scala-cli repl --scala 3
Welcome to Scala 3.1.3 (17.0.3, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> var i = 0
var i: Int = 0

scala> while
  |   i += 1
  |   i < 10
  | do ()

scala> i
val res0: Int = 10
```

**Lösn. uppg. 16.** *Postfixa operatorer för inkrementering och dekrementering.*

```
extension (i: Int)
  def ++ = i + 1
  def -- = i - 1
```

**Lösn. uppg. 17.** *Använda färdigt paket: Färgväljare.*

a) Den valda färgen returneras efter att användaren tryckt **OK**

```
1 scala> introprog.Dialog.selectColor()
2 val res1: java.awt.Color = java.awt.Color[r=0,g=204,b=0]
```

b) Default-färgen röd returneras efter att användaren tryckt **Cancel**

c) Färgväljaren återgår till default-färgen.

**Lösn. uppg. 18.** *Använda färdigt paket: användardialoger.*

a)

```
1 scala> introprog.Dialog.show("Game over!")
```

b) Funktionen input returnerar en sträng som blir tomma strängen "" om användaren klickar **Cancel**

```
1 scala> val name = introprog.Dialog.input("Vad heter du?")
2 name: String = Oddput Superkodare
```

c) Funktionen select returnerar en sträng med texten på knappen som användaren tryckte på.

```
1 scala> introprog.Dialog.select("Vad väljer du?",Vector("Sten","Sax","Påse"))
2 val res4: String = Påse
```

**Lösn. uppg. 19.** *Skapa din egen jar-fil.*

a)

jar -create -verbose -file <namn på skapad jar-fil> <namn på det som ska packas>

b)

```
package hello
```

```
object Main:
```

```
  def main(args: Array[String]): Unit = println("Hello package!")
```

```
scala-cli compile hello.scala --destination .
```

c)

```
1 > jar -c -v -f my.jar hello
2 > ls
3 > scala-cli repl --jar my.jar
4 scala> hello.Main.main(Array())
5 Hello package!
```

d)

```
1 > scala-cli run --jar my.jar --main-class hello.Main
```

**Lösn. uppg. 20.** *Hur stor är JDK8?* Med JDK8-plattformen kommer 4240 färdiga klasser, som är organiserade i 217 olika paket. Se Stackoverflow:

<http://stackoverflow.com/questions/3112882>

## L.5 Lösning classes

### L.5.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** Para ihop begrepp med beskrivning.

klass	1	↪	H	en mall för att skapa flera instanser av samma typ
instans	2	↪	I	upplaga av ett objekt med eget tillståndsminne
konstruktor	3	↪	M	skapar instans, allokerar plats för tillståndsminne
klassparameter	4	↪	K	binds till argument som ges vid konstruktion
referenslikhet	5	↪	B	instanser anses olika även om tillstånden är lika
innehållslikhet	6	↪	J	instanser anses lika om de har samma tillstånd
case-klass	7	↪	F	slipper skriva new; automatisk innehållslikhet
getter	8	↪	L	indirekt åtkomst av attributvärde
setter	9	↪	A	indirekt tilldelning av attributvärde
kompanjonsobjekt	10	↪	D	ser privata medlemmar i klass med samma namn
fabriksmetod	11	↪	E	hjälpfunktion för indirekt konstruktion
<b>null</b>	12	↪	G	ett värde som ej refererar till någon instans
<b>new</b>	13	↪	C	nyckelord vid direkt instansiering av klass

**Lösn. uppg. 2.** Klass och instans.

a)

Singelpunkt.x	1	↪	B	1
Punkt.x	2	↪	G	value is not a member of <b>object</b>
<b>val</b> p = <b>new</b> Singelpunkt	3	↪	C	Not found: type
<b>val</b> p1 = <b>new</b> Punkt	4	↪	D	p1: Punkt = Punkt@27a1a53c
<b>val</b> p2 = Punkt()	5	↪	F	p2: Punkt = Punkt@51ab04bd
{ p1.x = 1; p2.x }	6	↪	E	3
( <b>new</b> Punkt).y	7	↪	H	2
{ <b>val</b> p: Punkt = <b>null</b> ; p.x }	8	↪	A	java.lang.NullPointerException

b)

<i>fel</i>	<i>typ</i>	<i>förklaring</i>
value is not a member of <b>object</b>	kompileringsfel	det finns ingen instans med namnet Punkt. <i>Felmeddelandet syftar på att det i klassens autogenererade konstruktorombud saknas en variabel med namnet x.</i>
Not found: type	kompileringsfel	det finns ingen klass som heter Singelpunkt
NullPointerException	körtidsfel	det går inte att referera attribut i en instans som inte finns

### Lösn. uppg. 3. Klassparametrar.

a)

<b>val</b> p1 = Point(1, 2)	1	↪ C	p1: Point = Point@30ef773e
<b>val</b> p2 = Point()	2	↪ A	missing argument for parameter
<b>val</b> p2 = Point(3, 4)	3	↪ E	p2: Point = Point@218cf600
p2.x - p1.x	4	↪ B	2
Point(0, 1).y	5	↪ F	1
Point(0, 1, 2)	6	↪ D	too many arguments for constructor

b)

<i>fel</i>	<i>typ</i>	<i>förklaring</i>
missing argument for parameter	kompileringsfel	du måste ge argument vid konstruktion av klassen Point
too many arguments for constructor	kompileringsfel	antalet argument stämmer ej överens med antalet klassparametrar

### Lösn. uppg. 4. Oföränderlig klass med defaultargument.

a)

<b>val</b> p1 = Point3D()	1	↪ C	p1: Point3D = Point3D@2eb37eee
<b>val</b> p2 = Point3D(y = 1)	2	↪ F	p2: Point3D = Point3D@65a9e8d7
Point3D(z = 2).z	3	↪ E	value cannot be accessed
p2.y = 0	4	↪ B	Reassignment to <b>val</b>
p2.y == 0	5	↪ A	<b>false</b>
p1.x == Point3D().x	6	↪ D	<b>true</b>

b) Problemet är att så som klassen Point3D är deklarerad går det inte att avläsa z-koordinaten efter att en instans konstruerats. Det vore bättre om även z-attributet är **val**.

**Lösn. uppg. 5.** Case-klass, this, likhet, toString och kompanjonsobjekt.

a)

<b>val</b> p1 = Pt(1, 2)	1	↪ E	Pt(1,2)
<b>val</b> p2 = Pt(y = 3)	2	↪ C	Pt(0,3)
<b>val</b> p3 = MutablePt(5, 6)	3	↪ A	MPt(5,6)
<b>val</b> p4 = Mutable()	4	↪ D	Not found
p2.moved(dx = 1) == Pt(1, 3)	5	↪ F	<b>true</b>
p3.move(dy = 1) == MutablePt(5, 7)	6	↪ B	<b>false</b>

b) Kompilatorn härleder MutablePt eftersom det är typen på självreferensen this.

```
1 scala> :type new MutablePt().move()
2 MutablePt
```

c) Instansiering med universella apply-metoder (eng. *universal apply methods*) är godis som gör koden enklare att läsa och skriva. Detta är möjligt tack vare att det vid kompilering automatiskt skapas ett konstruktor-ombud (eng. *constructor proxy*) som instansierar objektet med nyckelordet **new**. Ett konstruktor-ombud är ett kompanjonsobjekt med tillhörande apply-metod.

Ett fall då **new** uttryckligen måste användas är vid implementering av egen apply-metod i ett kompanjonsobjekt. Om **new** inte används inuti apply-metoden, kommer samma metod att anropas rekursivt istället för att en ny instans skapas. Se följande exempel:

```
class Point3D(val x: Int, val y: Int, val z: Int)

object Point3D:
  var secretNumber = 42
  def apply(x: Int, y: Int, z: Int): Point3D =
    if secretNumber == 42 then
      Point3D(x, y, z) // Kodan kommer fastna i en evig loop.

    else new Point3D(x, y, z) // Funkar eftersom 'new' används.
```

d) En metod som avläser (delar av) ett objekts (privata) tillstånd utan att ändra det kallas för en *getter*.

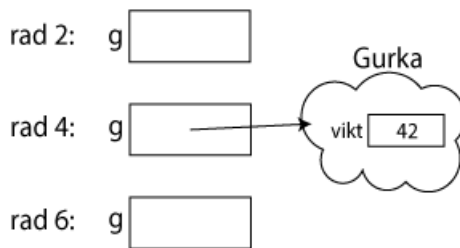
**Lösn. uppg. 6.** Implementera delar av klasserna *Pos*, *KeyControl*, *Mole* och *BlockWindow* som behövs under laborationen *blockbattle1*. Denna uppgift är laborationsförberedelse. Utvärdera dina lösningar genom egna tester i REPL.

a) Det går inte att anropa `Pos.moved(0,1)`. Anledningen till detta är att `moved` inte existerar i kompanjonsobjektet *Pos*, därav felmeddelandet "value moved is not a member of object Pos". För att anropa en metod definierad inuti en klass måste man göra anropet via en (referens till en) instans av klassen.

## L.5.2 Extrauppgifter; träna mer

**Lösn. uppg. 7.** Instansiering med *new* och värdet *null*.

a) Rad 3 och 7 ger båda felmeddelandet `java.lang.NullPointerException`, på grund av försök att referera medlemmar med hjälp av en *null*-referens, som alltså inte pekar på något objekt.



b)

**Lösn. uppg. 8.** Skapa en punktklass som kan hantera polära koordinater.

a)

```
package graphics

case class Point(x: Double, y: Double):
  val r: Double          = math.hypot(x, y)
  val theta: Double      = math.atan2(y, x)
  def negY: Point        = Point(x, -y)
  def +(p: Point): Point = Point(x + p.x, y + p.y)

object Point:
  def polar(r: Double, theta: Double): Point =
    Point(r * math.cos(theta), r * math.sin(theta))
```

b) **TODO!!!**

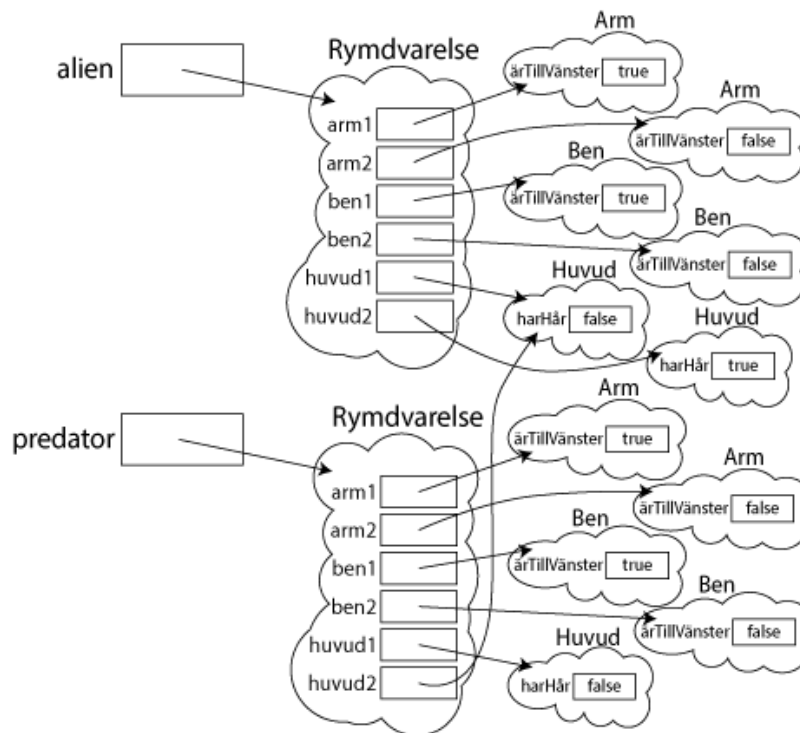
c) **TODO!!!**

d) **TODO!!!**

**Lösn. uppg. 9.** Klasser, instanser och skräp.

a) Vi skapar två rymdvarelser, *alien* och *predator*, med vardera två ben och två armar, samt vardera två huvuden (där det ena är skalligt och det andra har hår). Efter det är

varken alien eller predator skallig eftersom båda har ett huvud med hår. Sen låter man referensen till predators huvud med hår referera till aliens huvud utan hår. Nu är predator helt skallig och delar huvud med alien.



b) Eftersom det inte längre finns någon referens som pekar på det objektet kommer skräpsamlaren att ta hand om det och det kommer förr eller senare skrivas över av något annat när platsen i minnet behövs. Objekt som inte har någon referens till sig går inte att komma åt.

### Lösn. uppg. 10. Case-klass. Oföränderlig kvadrat.

a)

```
case class Square(val x: Int = 0, val y: Int = 0, val side: Int = 1):
  val area: Int = side * side

  def moved(dx: Int, dy: Int): Square = Square(x + dx, y + dy, side)

  def isEqualSizeAs(that: Square): Boolean = this.side == that.side

  def scale(factor: Double): Square =
    Square(x, y, (side * factor).round.toInt)

object Square:
  val unit: Square = Square()
```

b)

```
1 scala> val (s1, s2) = (Square(), Square(1, 10, 1))
2 val s1: Square = Square(0,0,1)
3 val s2: Square = Square(1,10,1)
4
```



```
5 scala> val s3 = s1 moved (1,-5)
6 val s3: Square = Square(1,-5,1)
7
8 scala> s1 isEqualSizeAs s3          // lika storlek
9 val res0: Boolean = true
10
11 scala> s2 isEqualSizeAs s1          // lika storlek
12 val res1: Boolean = true
13
14 scala> s1 isEqualSizeAs Square.unit // s1 har sidan 1
15 val res2: Boolean = true
16
17 scala> s2.scale(math.Pi) isEqualSizeAs s2 // olika storlek
18 val res3: Boolean = false
19
20 scala> s2.scale(math.Pi) == s2.scale(math.Pi) // lika innehåll
21 val res4: Boolean = true
22
23 scala> s2.scale(math.Pi) eq s2.scale(math.Pi) // olika objekt
24 val res5: Boolean = false
25
26 scala> Square.unit eq Square.unit    // samma objekt
27 val res6: Boolean = true
```

### L.5.3 Fördjupningsuppgifter; utmaningar

#### Lösn. uppg. 11. Innehållslighet mellan olika typer.

```

1 scala> 42 == "Fyrtiotvå"
2 1 |42 == "Fyrtiotvå"
3   |^^^^^^^^^^^^^^^^^^
4   |Values of types Int and String cannot be compared with == or !=
5
6 scala> Gurka(50) == Bil("Sedan")
7 val res0: Boolean = false

```

Det andra uttrycket är problematiskt eftersom det alltid kommer resultera i **false**, då klasserna Gurka och Bil är två ojämförbara typer som inte bör jämföras med avseende på innehållslighet. Detta försämrar typsäkerheten vilket ökar risken för svårupptäckta buggar där fel typer jämförs.

Likhetsjämförelser som sker mellan primitiva typer typkollas av kompilatorn och kan därför ge kompileringsfel om två olika typer, såsom Int och String, jämförs med varandra. Detta gäller dock i regel inte egendefinierade typer, vilket alltså innebär att en likhetsjämförelse mellan olika egendefinierade typer alltid resulterar i **false**.

Det är emellertid möjligt att få samma typkoll för egendefinierade typer som för primitiva typer genom att importera `scala.language.strictEquality`.

```

import scala.language.strictEquality
class Gurka(val vikt: Int)

class Bil(val typ: String)

```

```

1 scala> Gurka(50) == Bil("Sedan")
2 1 |Gurka(50) == Bil("Sedan")
3   |^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
4   |Values of types Gurka and Bil cannot be compared with == or !=

```

#### Lösn. uppg. 12. Attributrepresentation. Privat konstruktör. Fabriksmetod.

a) Det blir kompileringsfel eftersom konstruktorn är privat.

```

1 scala> class Point private (val x: Int, val y: Int)
2   | object Point:
3   |   def apply(x: Int = 0, y: Int = 0): Point = new Point(x, y)
4   |   val origo = apply()
5   |
6 // defined class Point
7 // defined object Point
8
9 scala> new Point(0, 0)
10 1 |new Point(0, 0)
11   |   ^^^^^
12   |constructor Point cannot be accessed as a member of Point from module class

```

b)

- Genom att ha en privat konstruktör och bara göra indirekt instansiering via fabriksmetod är lätt ändra attributrepresentation i framtiden utan att befintlig kod behöver ändras.

- Accessreglerna för kompanjonsobjekt är sådana att kompanjoner ser varandras privata delar.

c)

```
class Point private (private val p: (Int, Int)):
  def x: Int = p._1
  def y: Int = p._2

object Point:
  def apply(x: Int = 0, y: Int = 0): Point = new Point(x, y)
  val origo = apply()
```

**Lösn. uppg. 13.** Synlighet av klassparametrar och konstruktor, **private**[this].

a) Gurka5 är trasig. Eftersom vikten i Gurka5 är privat för instansen och inte klassen, kan en instans inte accessa en annan instans vikt.

```
1 11 | def kompisVikt = kompis.vikt
2   |           ^^^^^^^^^^^
3   |value vikt cannot be accessed as a member of (Gurka5.this.kompis : Gurka5) from class Gurka5.
```

b)

```
1 scala> new Gurka1(42).vikt
2 1 |new Gurka1(42).vikt
3   |^^^^^^^^^^^^^^^^^^^^
4   |value vikt cannot be accessed as a member of Gurka1 from module class
5
6 scala> new Gurka2(42).vikt
7 val res0: Int = 42
8
9 scala> new Gurka3(42).vikt
10 1 |new Gurka3(42).vikt
11   |^^^^^^^^^^^^^^^^^^^^
12   |value vikt cannot be accessed as a member of Gurka3 from module class
13
14 scala> val ingenGurka: Gurka4 = null
15 val ingenGurka: Gurka4 = null
16
17 scala> new Gurka4(42, ingenGurka).kompisVikt
18 java.lang.NullPointerException: Cannot invoke "rs$line$1$Gurka4.vikt()" bec...
19   at rs$line$1$Gurka4.kompisVikt(rs$line$1:8)
20   ... 38 elided
21
22 scala> new Gurka4(42, new Gurka4(84, null)).kompisVikt
23 val res2: Int = 84
24
25 scala> new Gurka6(42)
26 1 |new Gurka6(42)
27   |^^^^^^
28   |constructor Gurka6 cannot be accessed as a member of Gurka6 from module...
29
30 scala> new Gurka7(-42)
31 1 |new Gurka7(-42)
32   |^^^^^^
33   |constructor Gurka7 cannot be accessed as a member of Gurka7 from module...
34
35 scala> Gurka7(-42)
36 java.lang.IllegalArgumentException: requirement failed: negativ vikt: -42
```

```

37
38 scala> val g = Gurka7(42)
39 val g: Gurka7 = Gurka7@51fd1c7c
40
41 scala> g.vikt
42 val res4: Int = 42
43
44 scala> g.vikt = -1
45
46 scala> g.vikt
47 val res5: Int = -1

```

### Lösn. uppg. 14. Egendefinierad setter kombinerat med privat konstruktor.

a)

Rad 1:

```

1 java.lang.IllegalArgumentException: requirement failed: negativ vikt: -42

```

Gurka8.apply kräver att vikt  $\geq 0$  annars kastar require ett undantag.

Rad 5:

```

1 java.lang.IllegalArgumentException: requirement failed: negativ vikt: -1

```

Settern vikt\_ = kräver att vikt  $\geq 0$  annars kastar require ett undantag.

Rad 7:

```

1 java.lang.IllegalArgumentException: requirement failed: negativ vikt: -958

```

Eftersom  $42 - 1000$  är mindre än noll kastar require ett undantag.

b) Man kan sätta egna mer specifika krav på vad som får göras med värdena så man har större koll på att inget oväntat händer.

### Lösn. uppg. 15. Objekt med föränderligt tillstånd (eng. mutable state).

a)

```

class Frog private (initX: Int = 0, initY: Int = 0):
  private var _x: Int = initX
  private var _y: Int = initY
  private var _distanceJumped: Double = 0

  def x: Int = _x
  def y: Int = _y

  def jump(dx: Int, dy: Int): Unit =
    _x += dx
    _y += dy
    _distanceJumped += math.hypot(dx, dy)

  def randomJump: Unit =
    def rnd = util.Random.nextInt(10) + 1
    jump(rnd, rnd)

```

```

def distanceToStart: Double = math.hypot(x,y)
def distanceJumped: Double = _distanceJumped
def distanceTo(f: Frog): Double = math.hypot(x - f.x, y - f.y)

object Frog:
  def spawn(): Frog = Frog()

```

b) Exempel på testprogram:

```

object FrogTest:
  def test(): Unit =
    val f1 = Frog.spawn()
    assert(f1.x == 0 && f1.y == 0, "Test of spawn, reqt 1 & 4 failed.")

    f1.jump(4, 3)
    assert(f1.x == 4 && f1.y == 3, "Test of jump, reqt 1 & 4 failed.")

    f1.jump(4, 3)
    assert(f1.distanceJumped == 10, "Test of jump, reqt 2 failed.")

    f1.jump(-4, -3)
    assert(f1.distanceToStart == 5, "Test of jump, reqt 3 failed.")

    for x <- 1 to 10000 do
      val f2 = Frog.spawn()
      f2.randomJump
      assert(f2.x > 0 && f2.x <= 10 && f2.y > 0 && f2.y <= 10,
        "Test of randomJump, reqt 5 failed.")

    println("Test Ok!")

```

c) En metod som är en indirekt avläsning av attributvärden kallas getter.

d)

```

class Frog private (initX: Int = 0, initY: Int = 0):
  private var _x: Int = initX
  private var _y: Int = initY
  private var _distanceJumped: Double = 0

  def jump(dx: Int, dy: Int): Unit =
    _x += dx
    _y += dy
    _distanceJumped += math.hypot(dx, dy)

  def x: Int = _x
  def x_=(newX: Int): Unit = // Setter för x
    _distanceJumped += math.abs(x - newX)
    _x = newX

```

```

def y: Int = _y
def y_=(newY: Int): Unit = // Setter för y
    _distanceJumped += math.abs(y - newY)
    _y = newY

def randomJump: Unit =
    def rnd = util.Random.nextInt(10) + 1
    jump(rnd, rnd)

def distanceToStart: Double = math.hypot(x,y)
def distanceJumped: Double = _distanceJumped
def distanceTo(f: Frog): Double = math.hypot(x - f.x, y - f.y)

object Frog:
    def spawn(): Frog = Frog()

```

e)

```

object FrogSimulation:
    def isAnyCollision(frogs: Vector[Frog]): Boolean =
        var found = false
        frogs.indices.foreach(i => // generate all pairs (i,j)
            for j <- i + 1 until frogs.size do
                if !found then
                    found = frogs(i).distanceTo(frogs(j)) <= 0.5
        )
        found

    def jumpUntilCrash(n: Int = 100, initDist: Int = 8): (Int, Double) =
        val frogs = Vector.fill(n)(Frog.spawn())
        (0 until n).foreach(i => frogs(i).x = i * initDist)
        var count = 0
        while !isAnyCollision(frogs) do
            frogs(util.Random.nextInt(n)).randomJump
            count += 1
        (count, frogs.map(_.distanceJumped).sum)

    def run(nbrOfCrashTests: Int = 10) =
        for i <- 1 to nbrOfCrashTests do
            val (n, dist) = jumpUntilCrash()
            println(s"\nAntalet looprundor innan grodkrock: $n")
            println(s"Totalt avstånd hoppat av alla grodor: $dist")

```

**Lösn. uppg. 16.** Objekt med föränderligt tillstånd (eng. *mutable state*).

```

class Square private (val initX: Int, val initY: Int, val initSide: Int):
    private var nMoves = 0

```

```

private var sumCost = 0.0

private var _x = initX
private var _y = initY

private var _side = initSide

private def addCost(): Unit =
  sumCost += math.hypot(x - initX, y - initY) * side

def x: Int = _x
def y: Int = _y

def side = _side

def scale(factor: Double): Unit = _side = (_side * factor).round.toInt

def move(dx: Int, dy: Int): Unit =
  _x += dx; _y += dy
  nMoves += 1
  addCost()

def moveTo(x: Int, y: Int): Unit =
  _x = x; _y = y
  nMoves += 1
  addCost()

def cost: Double = sumCost

def pay: Double = {val temp = sumCost; sumCost = 0; temp}

override def toString: String =
  s"Square[($x, $y), side: $side, #moves: $nMoves times, cost: $sumCost]"

object Square:
  private var created = Vector[Square]()

  def apply(x: Int, y: Int, side: Int): Square =
    require(side >= 0, s"side must be positive: $side")
    val sq = (new Square(x, y, side))
    created := sq
    sq

  def apply(): Square = apply(0, 0, 1)

  def totalNumberOfMoves: Int = created.map(_.nMoves).sum

  def totalCost: Double = created.map(_.cost).sum

```

## L.6 Lösning patterns

### L.6.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** Matcha på konstanta värden.

a) Scalas **match**-uttryck jämför stegvis värdet med varje **case** för att sedan returnera ett värde tillhörande motsvarande **case**.

b)

```
1 scala.MatchError
```

Exekveringsfel, uppstår av en viss input under körningen.

**Lösn. uppg. 2.** Gard i case-grenar.

Garden som införts vid **case** 'g' slumpar fram ett tal mellan 0 och 1 och om talet inte är större än 0.5 så blir det ingen matchning med **case** 'g' och programmet testat vidare tills default-caset.

Gardens krav måste uppfyllas för att det ska matcha som vanligt.

**Lösn. uppg. 3.** Mönstermatcha på attributen i case-klasser.

G100true. Vid byte av plats: Gtrue100.

**match** testat om kompanjonsobjektet Gurka är av typen Gurka med två parametervärden. De angivna parametrarna tilldelas namn, vikt får namnet v och ärRutten namnet rutten och skrivs sedan ut. Byts namnen dessa ges skrivs de ut i den omvända ordningen.

**Lösn. uppg. 4.** Matcha på case-objekt och nyttan med **sealed**.

a)

```
1 Cannot extend sealed trait Färg in a different source file
```

Felmeddelandet fås av att REPL:en behandlar varje inmatning individuellt och tillåter därför inte att subtypen Spader ärver från (eng. *extends*) supertypen Färg eftersom denna var förseglad (eng. *sealed*). Mer om detta senare i kursen...

b) -

c) Förusatt att **import** Kortlek.\_ har skrivits...

```
def paraFärg(f: Färg): Färg = f match
  case Spader => Klöver
  case Hjärter => Ruter
  case Ruter => Hjärter
  case Klöver => Spader
```

d)

```
1 <console>:17: warning: match may not be exhaustive.
2 It would fail on the following input: Ruter
```

Varningen kommer redan vid kompilering.

e)

```
1 scala.MatchError: Ruter (of class Ruter)
2 at .paraFärg(<console>:17)
```



Detta är ett körtidsfel.

f) Om en klass är **sealed** innebär det att om ett element ska matchas och är en subtyp av denna klass så ger Scala varning redan vid kompilering om det finns en risk för ett `MatchError`, alltså om **match**-uttrycket inte är uttömmande och det finns fall som inte täcks av ett **case**.

En förseglad supertyp innebär att programmeraren redan vid kompileringstid får en varning om ett fall inte täcks och i sånt fall vilket av undertyperna, liksom annan hjälp av kompilatorn. Detta kräver dock att alla subtyperna delar samma fil som den förseglade klassen.

### Lösn. uppg. 5. *Mönstermatcha enumeration.a)*

```
def paraFärg(f: Färg): Färg = f match
  case Färg.Spader => Färg.Klöver
  case Färg.Hjärter => Färg.Ruter
  case Färg.Ruter => Färg.Hjärter
  case Färg.Klöver => Färg.Spader
```

Likt uppgift 4c så kan även här en **import**-sats skrivas för att nå medlemmarna i `Färg` utan punktnotation. Det är dock inte alltid fördelaktigt att importera medlemmar till den globala namnrymden, då det kan förekomma namnkrockar. Anta ett exempel där vi jobbar på ett program med grafiskt användargränssnitt där vi har en färg `Red` definierad. Anta också att vi nu till vårt program vill importera ytterligare en röd färg för kulörerna hjärter och ruter, denna också namngiven `Red`. I detta scenario hade det uppstått en namnkrock då `Red` redan är definierad så importeringen hade ej kunnat ske.

b) Vid mönstermatchning så fungerar **sealed trait** ihop med **case**-objekt i praktiken likadant som att använda sig av **enum**. Vi såg att i deluppgift 4d så varnade REPL redan vid kompilering att denna matchning inte var uttömmande (eng. *exhaustive*). Detta gäller även vid användning av **enum**.

### Lösn. uppg. 6. *Betydelsen av små och stora begynnelsebokstäver vid matchning.*

a) Både `str` och vadsomhelst matchar med inputen, oavsett vad denna är på grund av att de har en liten begynnelsebokstav.

`str` har dock en gard att strängen måste börja med `g` vilket gör så endast **val** `g = "gurka"` matchar med denna. **val** `x = "urka"` plockas dock upp av vadsomhelst som är utan gard.

b)

```
1 <console>:16: warning: patterns after a variable pattern cannot match (SLS 8.1
2 .1)
```

och

```
1 <console>:17: warning: unreachable code due to variable patter 'tomat' on line
2 16
```

Trots att en klass `tomat` existerar så tolkar Scalias **match** den som en **case**-gren som fångar allt på grund av en liten begynnelsebokstav. Detta gör så alla objekt som inte är av typen `Gurka` kommer ge utskriften `tomat` och att sista caset inte kan nås.

c)

```
case `tomat` => println("tomat")
```

**Lösn. uppg. 7.** Matcha på innehåll i en Vector.

```

1 jeh
2 jed
3 42

```

För varje element i xss görs en matching som resulterar i en sträng. Vad som händer i varje gren förklaras nedan.

1. Första match-grenen aktiveras aldrig eftersom xss ej innehåller någon tom vektor.
2. Andra grenen passar med Vector("hej") och variablen a binds till "hej".
3. Tredje grenen matchar Vector("på", "dej") där första värdet binds inte till någon variabel eftersom understreck finns på motsvarande plats, medan andra värdet binds till b.
4. Fjärde grenen matchar en sekvens med tre värden där mittenvärdet är "x". Den sista grenen aktiveras inte i detta exempel men hade matchat allt som inte fångas av tidigare grenar.

**Lösn. uppg. 8.** Använda Option och matcha på värden som kanske saknas.

a)

1. **var** kanske blir en Option som håller Int men är utan något värde, kallas då None.
2. Eftersom **var** kanske är utan värde är storleken av den 0.
3. **var** kanske tilldelas värdet 42 som förvaras i en Some som visar att värde finns.
4. Eftersom **var** kanske nu innehåller ett värde är storleken 1.
5. Eftersom **var** kanske innehåller ett värde är den inte tom.
6. Eftersom **var** kanske innehåller ett värde är den definierad.
7. **def** ökaOmFinns matchar en Option[Int] med dess olika fall.  
Finns ett värde, alltså opt: Option[Int] är en Some, så returneras en Some med ursprungliga värdet plus 1.  
Finns inget värde, alltså opt: Option[Int] är en None, så returneras en None.
8. -
9. -
10. -
11. **def** ökaOmFinns appliceras på kanske och returnerar en Some med värdet hos kanske plus 1, alltså 43.
12. **def** öka tar emot värdet av en Int och returnerar värdet av denna plus 1.
13. map applicerar **def** öka till det enda elementen i kanske, 42. Denna funktion returnerar en Some med värdet 43 som tilldelas merKanske.

b)

1. **val** meningen blir en `Some` med värdet 42.
2. **val** `ejMeningen` blir en `Option[Int]` utan något värde, en `None`.
3. `map(_ + 1)` appliceras på meningen och ökar det existerande värdet med 1 till 43.
4. `map(_ + 1)` appliceras på `ejMening` men eftersom inget värde existerar fortsätter denna vara `None`.
5. `map(_ + 1)` appliceras ännu en gång på `ejMening` men denna gång inkluderas metoden `orElse`. Om ett värde inte existerar hos en `Option`, alltså är av typen `None`, så utförs koden i `orElse`-metoden som i detta fall skriver ut *saknas* för värdet som saknas.
6. Samma anrop från föregående rad utförs denna gång på meningen och eftersom ett värde finns utförs endast första biten som ökar detta värde med 1.

Denna metod kan användas i stället för **match**-versionen i föregående exempel i och med dennas simplare form. En `Option` innehåller ju antingen ett värde eller inte så ett längre **match**-uttryck är inte nödvändigt.

c)

1. En vektor `xs` skapas med var femte tal från 42 till 82.
2. En tom `Int`-vektor `e` skapas.
3. `headOption` tar ut första värdet av vektorn `xs` och returnerar den sparad i en `Option`, `Some(42)`.
4. Första värdet i vektorn `xs` sparas i en `Option` och hämtas sedan av `get`-metoden, 42.
5. Som i föregående rad men denna gång används `getOrElse` som om den `Option` som returneras saknar ett värde, alltså är av typen `None`, returnerar 0 istället. Eftersom `xs` har minst ett värde så är den `Option` som returneras inte `None` och ger samma värde som i föregående, 42.
6. Som föregående rad fast istället för att returnera 0 om värde saknas så returneras en `Option[Int]` med 0 som värde.
7. `headOption` försöker ta ut första värdet av vektorn `e` men eftersom denna saknar värden returneras en `None`.

8.

```
1 java.util.NoSuchElementException: None.get
```

Liksom föregående rad returnerar `headOption` på den tomma vektorn `e` en `None`. När `get`-metoden försöker hämta ett värde från en `None` som saknar värde ger detta upphov till ett körtidsfel.

9. Liksom i föregående returneras `None` av `headOption` men eftersom `getOrElse`-metoden används på denna `None` returneras 0 istället.
10. Liksom föregående används `getOrElse`-metoden på den `None` som returneras. Denna gång returneras dock en `Option[Int]` som håller värdet 0.
11. En vektor innehållandes elementen `xs`-vektorn och 3 `e`-vektorer skapas.

12. `map` använder metoden `lastOption` på varje delvektor från vektorn på föregående rad. Detta sammanställer de sista elementen från varje delvektor i en ny vektor. Eftersom vektor `e` är tom returneras `None` som element från denna.
13. Samma sker som i föregående rad men `flatten`-metoden appliceras på slutgiltiga vektorn som rensar vektorn på `None` och lämnar endast faktiska värden.
14. `lift`-metoden hämtar det eventuella värdet på plats 0 i `xs` och returnerar den i en `Option` som blir `Some(42)`.
15. `lift`-metoden försöker hämta elementet på plats 1000 i `xs`, eftersom detta inte existerar returneras `None`.
16. Samma sker som i föregående fast applicerat på vektorn `e`. Sedan appliceras `getOrElse(0)` som, eftersom `lift`-metoden returnerar `None`, i sin tur returnerar 0.
17. `find`-metoden anropas på `xs`-vektorn. Den letar upp första talet över 50 och returnerar detta värde i en `Option[Int]`, alltså `Some(52)`.
18. `find`-metoden anropas på `xs`-vektorn. Den letar upp första värdet under 42 men eftersom inget värde existerar under 42 i `xs` returneras `None` istället.
19. `find`-metoden anropas på `e`-vektorn och skriver ut *HITTAT!* om ett element under 42 hittas. Eftersom `e`-vektorn är tom returneras `None` vilket `foreach` inte räknar som element och därav inte utförs på.

d) Användning av -1 som returvärde vid fel eller avsaknad på värde kan ge upphov till körtidsfel som är svåra att upptäcka. **null** kan i sin tur orsaka kraschar om det skulle bli fel under körningen. `Option` har inte samma problem som dessa, används ett `getOrElse`-uttryck eller dylikt så kraschar inte heller programmet.

Dessutom behöver inte en funktion som returnerar en `Option` samma dokumentation av returvärdena. Istället för att skriva kommentarer till koden på vilka värden som kan returneras och vad dessa betyder så syns det direkt i koden.

Slutgiltigen är `Option` mer typsäkert än **null**. När du returnerar en `Option` så specificeras typen av det värde som den kommer innehålla, om den innehåller något, vilket underlättar att förstå och begränsar vad den kan returnera.

### Lösn. uppg. 9. Kasta undantag.

a)

1. Ett `Exception` kastas med felmeddelandet *PANG!*.
2. Flera olika typer av `Exception` visas.
3. En typ av `Exception`, `IllegalArgumentException`, kastas med felmeddelandet *fel fel fel*.
4. Ett undantag med felmeddelandet *stormvind!* kastas och fångas av **catch**-uttrycket. Ett **match**-uttryck undersöker undantaget och skriver ut meddelandet, samt returnerar -1.

b) Exempelvis:

`OutOfMemoryError`, om programmet får slut på minne.

`IndexOutOfBoundsException`, om en vektorposition som är större än vad som finns hos vektorn försöker nås.

`NullPointerException`, om en metod eller dylikt försöker användas hos ett objekt som inte finns och därav är en nullreferens.

c) om både try-grenen och catch-grenen har samma typ, här `Int`, så härleder kompilatorn samma typ för hela uttrycket. Skulle **catch**-grenen returnera ett värde av en helt annan typ istället, t.ex. `String`, så blir den mest precisa typen som kompilatorn kan härleda för hela uttrycket `Matchable`, som är en direkt subtyp till den mest generella typen `Any`.

### Lösn. uppg. 10. Fånga undantag med `scala.util.Try`.

- a)
  1. **def** `pang` skapas som kastar ett `Exception` med felmeddelandet *PANG!*.
  2. Scalas verktyg `Try`, `Success` och `Failure` importeras.
  3. **def** `pang` anropas i `Try` som fångar undantaget och kapslar in den i en `Failure`.
  4. Metoden `recover` matchar undantaget i `Failure` från föregående rad med ett **case** och gör om föredetta `Failure` till `Success` vid matchning, liknande **catch**.
  5. Strängen *tyst* körs i föregående test men eftersom inget undantag kastas blir den inkapslad i en `Success` och `recover` behöver inte göra något. Den tar endast hand om undantag.
  6. **def** `kanskePang` skapas som har lika stor chans att returnera strängen *tyst* såsom anropa **def** `pang`.
  7. **def** `kanskeOk` skapas som testar **def** `kanskePang` med `Try`.
  8. En vektor `xs` fylls med resultaten, `Success` och `Failure`, från 100 körningar av `kanskeOk`.
  9. Elementet på plats 13 i vektor `xs` matchas med något av 2 **case**. Om det är en `Success` skrivs :) ut, om en `Failure` skrivs :( plus felmeddelandet ut.
  10. -
  11. -
  12. Metoden `isSuccess` testar om elementet på plats 13 i `xs` är en `Success` och returnerar **true** om så är fallet.
  13. Metoden `isFailure` testar om elementet på plats 13 i `xs` är en `Failure` och returnerar **true** om så är fallet.
  14. Metoden `count` räknar med hjälp av `isFailure` hur många av elementen i `xs` som är `Failure` och returnerar detta tal.
  15. Metoden `find` letar upp med hjälp av `isFailure` ett element i `xs` som är `Failure` och returnerar denna i en `Option`.
  16. `badOpt` tilldelas den första `Failure` som hittas i `xs`.

17. `goodOpt` tilldelas den första `Success` som hittas i `xs`.
  18. Resultatet `badOpt` skrivs ut, `Option[scala.util.Try[String]] = Some(Failure(java.lang.Exception: PANG!))`
  19. Metoden `get` hämtar från `badOpt` den `Failure` som förvaras i en `Option`.
  20. Metoden `get` anropas ännu en gång på resultatet från föregående rad, alltså en `Failure`, som hämtar undantaget från denna och som då i sin tur kastas.
  21. Metoden `getOrElse` anropas på den `Failure` som finns i `badOpt`. Eftersom detta är en `Exception` utförs `orElse`-biten istället för att undantaget försöker hämtas. Då returneras strängen *bomben desarmerad!*.
  22. Metoden `getOrElse` anropas på den `Success` som finns i `goodOpt`. Eftersom detta är en `Success` med en normal sträng sparad i sig returneras denna sträng, *tyst*.
  23. Metoden från föregående används denna gång på alla element i `xs` där resultatet skrivs ut för varje.
  24. Metoden `toOption` appliceras på alla `Success` och `Failure` i `xs`. De med ett `exception`, alltså `Failure`, blir en `None` medan de med värden i `Success` ger en `Some` med strängen *tyst* i sig.
  25. Metoden `flatten` appliceras på vektorn fylld med `Option` från föregående rad för att ta bort alla `None`-element.
  26. Metoden `size` används på slutgiltiga listan från föregående rad för att räkna ut hur många `Some` som resultatet innehåller. Den har alltså beräknat antalet element i `xs` som var av typen `Success` med hjälp av `Option`-typen.
- b) `pang` har returtypen `Nothing`, en specialtyp inom `Scala` som inte är kopplad till `Any`, och som inte går att returnera.
- c) Typen `Nothing` är en subtyp av varenda typ i `Scalas` hierarki. Detta innebär att den även är en subtyp av `String` vilket implicerar att `String` inkluderar både strängar och `Nothing` och därav blir returtypen.

## L.6.2 Fördjupningsuppgifter; utmaningar

**Lösn. uppg. 11.** Använda *matchning* eller *dynamisk bindning*?

a)

```
package vegopoly

trait Grönsak:
  def vikt: Int
  def ärRutten: Boolean
  def ärÄtbar: Boolean

case class Gurka(vikt: Int, ärRutten: Boolean) extends Grönsak:
  val ärÄtbar: Boolean = (!ärRutten && vikt > 100)
```

```

case class Tomat(vikt: Int, ärRutten: Boolean) extends Grönsak:
  val ärÄtbar: Boolean = (!ärRutten && vikt > 50)

object Main:
  def slumpvikt: Int = (math.random()*500 + 100).toInt

  def slumprutten: Boolean = math.random() > 0.8

  def slumpgurka: Gurka = Gurka(slumpvikt, slumprutten)

  def slumptomat: Tomat = Tomat(slumpvikt, slumprutten)

  def slumpgrönsak: Grönsak =
    if math.random() > 0.2 then slumpgurka else slumptomat

  def main(args: Array[String]): Unit =
    val skörd = Vector.fill(args(0).toInt)(slumpgrönsak)
    val ätvärda = skörd.filter(_.ärÄtbar)
    println("Antal skördade grönsaker: " + skörd.size)
    println("Antal ätvärda grönsaker: " + ätvärda.size)

```

b) Följande **case class** läggs till:

```

case class Broccoli(vikt: Int, ärRutten: Boolean) extends Grönsak:
  val ärÄtbar: Boolean = (!ärRutten && vikt > 80)

```

Därefter läggs följande till i **object** Main innan **def** slumpgrönsak:

```

def slumpbroccoli: Broccoli = Broccoli(slumpvikt, slumprutten)

```

Slutligen ändras **def** slumpgrönsak till följande:

```

def slumpgrönsak: Grönsak =      // välj t.ex. denna fördelning:
  val rnd = math.random()
  if rnd > 0.5 then slumpgurka      // 50% sannolikhet för gurka
  else if rnd > 0.2 then slumptomat // 30% sannolikhet för tomat
  else slumpbroccoli              // 20% sannolikhet för broccoli

```

c) Fördelarna med **match**-versionen, och mönstermatchning i sig, är att det är väldigt lätt att göra ändringar på hur matchningen sker. Detta innebär att det skulle vara väldigt lätt att ändra definitionen för ätbarheten. Skulle dock dessa inte ändras ofta utan snarare grönsaksutbudet så kan det polyformistiska alternativet vara att föredra. Detta eftersom det skulle implementeras och ändras lättare än mönstermatchningen vid byte av grönsaker.

### Lösn. uppg. 12. Metoden equals.

a)

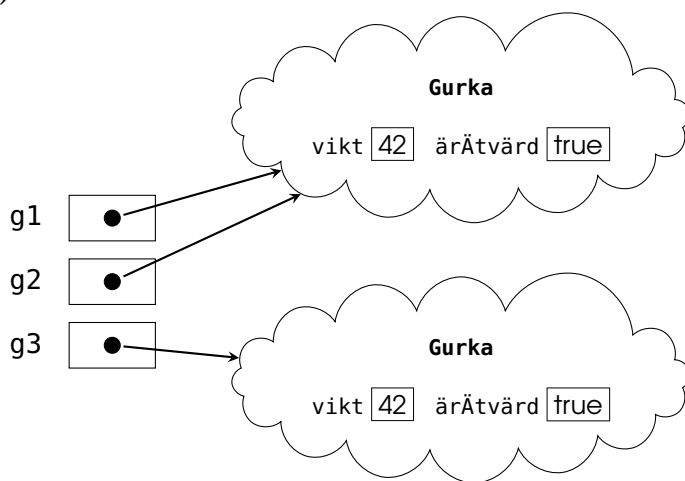
1. En klass Gurka skapas med parametrarna vikt av typen Int och ärÄtbar av typen Boolean.



2. g1 tilldelas en instans av Gurka-klassen med vikt = 42 och ärÄtbar = **true**.
3. g2 tilldelas samma Gurka-objekt som g1.
4. g3 tilldelas en ny instans av Gurka-klassen med motsvarande parametrar som g1.
5. `==(equals)`-metoden jämför g1 med g2 och returnerar **true**.
6. `==(equals)`-metoden jämför g1 med g3 och returnerar **false**.
7. **def** `equals(x\$: Any): Boolean`

Som kan ses ovan är elementet som jämförs i `equals` av typen `Any`. Eftersom programmet inte känner till klassen så används `Any.equals` vid jämförelsen. Till skillnad från de primitiva datatyperna som vid jämförelse med `equals` jämför innehållslighet, så jämförs referenslikheten hos klasser om inget annat är specificerat. g1 och g2 refererar till samma objekt medan g3 pekar på ett eget sådant vilket innebär att g1 och g3 inte har referenslikhet.

b)



c) -

d) I de första 3 raderna sker samma som i deluppgift a. När nu dessa jämförelser görs mellan Gurka-objekten så överskuggas `Any.equals` av den `equals` som är specificerad för just Gurka. Eftersom båda objekten g1 jämförs med också är av typen Gurka så matchar den med **case** `that: Gurka`. Denna i sin tur jämför vikterna hos de båda gurkorna och returnerar en Boolean huruvida de är lika eller inte, vilket de i båda fallen är.

e) I deluppgift a gav g1 `== g3` **false** trots innehållslighet. Efter skuggningen ger dock detta uttryck **true** vilket påvisar jämförelse av innehållslighet.

### Lösn. uppg. 13. *Polynom.*

a) **TODO!!!**

b) **TODO!!!**

### Lösn. uppg. 14. *Option som en samling.* **TODO!!!**

### Lösn. uppg. 15. *Fånga undantag med catch i Java och Scala.* **TODO!!!**

### Lösn. uppg. 16. *Polynom, fortsättning: reducering.*

### Lösn. uppg. 17. *Typsäker innehållstest med metoden ==.*



**Lösn. uppg. 18.** Överskugga *equals* med innehållslighet även för icke-finala klasser.

**Lösn. uppg. 19.** Överskugga *equals* vid arv.

**Lösn. uppg. 20.** Speciella matchningar. **TODO!!!**

**Lösn. uppg. 21.** Extraktorer. **TODO!!!**

**Lösn. uppg. 22.** Polynom, fortsättning: polynomdivision. **TODO!!!**

## L.7 Lösning sequences

### L.7.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** Para ihop begrepp med beskrivning.

element	1	↔	G	objekt i en datastruktur
samling	2	↔	B	datastruktur med element av samma typ
samlingsbibliotek	3	↔	J	många färdiga samlingar med olika egenskaper
sekvens(samling)	4	↔	L	noll el. flera element av samma typ i viss ordning
sekvensalgorithm	5	↔	I	lösning på problem som drar nytta av sekvenssamling
ordning	6	↔	A	definierar hur element av en viss typ ska ordnas
sortering	7	↔	C	algorithm som ordnar element i en viss ordning
sökning	8	↔	D	algorithm som letar upp element enligt sökkriterium
linjärsökning	9	↔	F	sökalgorithm som letar i sekvens tills element hittas
registrering	10	↔	H	algorithm som räknar element med vissa egenskaper
tidskomplexitet	11	↔	E	hur exekveringstiden växer med problemstorleken
minneskomplexitet	12	↔	K	hur minnesåtgången växer med problemstorleken

**Lösn. uppg. 2.** Olika sekvenssamlingar.

Vector	1	↔	B	oföränderlig, ger snabbt godtyckligt ändrad samling
List	2	↔	C	oföränderlig, ger snabbt ny samling ändrad i början
Array	3	↔	D	primitiv, förändringsbar, snabb indexering, fix storlek
ArrayBuffer	4	↔	A	förändringsbar, snabb indexering, kan ändra storlek
ListBuffer	5	↔	E	förändringsbar, snabb att ändra i början

**Lösn. uppg. 3.** Använda sekvenssamlingar.

a)

<code>x += xs</code>	1	↪ G	<code>Vector(0, 1, 2, 3)</code>
<code>xs += x</code>	2	↪ L	<code>error: value += is not a member of Int</code>
<code>xs :+ x</code>	3	↪ J	<code>Vector(1, 2, 3, 0)</code>
<code>xs ++ xs</code>	4	↪ M	<code>Vector(1, 2, 3, 1, 2, 3)</code>
<code>xs.indices</code>	5	↪ E	<code>(0 until 3)</code>
<code>xs apply 0</code>	6	↪ C	<code>1</code>
<code>xs(3)</code>	7	↪ I	<code>java.lang.IndexOutOfBoundsException</code>
<code>xs.length</code>	8	↪ O	<code>3</code>
<code>xs.take(4)</code>	9	↪ F	<code>Vector(1, 2, 3)</code>
<code>xs.drop(2)</code>	10	↪ K	<code>Vector(3)</code>
<code>xs.updated(0, 2)</code>	11	↪ B	<code>Vector(2, 2, 3)</code>
<code>xs.tail.head</code>	12	↪ N	<code>2</code>
<code>xs.head.tail</code>	13	↪ D	<code>error: value tail is not a member of Int</code>
<code>xs.isEmpty</code>	14	↪ H	<b>false</b>
<code>xs.nonEmpty</code>	15	↪ A	<b>true</b>

b)

<i>fel</i>	<i>typ</i>	<i>förklaring</i>
<code>value += is not a member of Int</code>	kompileringsfel	Operatorer som slutar med kolon är högerassociativa. Metoden <code>xs += x</code> motsvarar med punktnotation <code>x.+=(xs)</code> och det finns ingen metod med namnet <code>+=</code> på heltal.
<code>IndexOutOfBoundsException</code>	löp-tidsfel	Det finns bara 3 element och index räknas från 0 i sekvens-samlingar.
<code>value tail is not a member of Int</code>	kompileringsfel	Metoden <code>head</code> ger första elementet och heltal saknar sekvenssamlingsmetoden <code>tail</code> .

**Lösn. uppg. 4.** Kopiering av sekvenser.

a)

<code>xs(0)</code>	<code>rs\$line5\$Mutant@66d766b9</code> nya instanser får nya hexkoder
<code>ys(0).int</code>	<code>0</code> eftersom <code>ys</code> innehåller samma instans som <code>xs</code>
<code>zs(0).int</code>	<code>5</code> eftersom <code>!(xs(0) eq zs(0))</code>
<code>xs(0) eq ys(0)</code>	<b>true</b> eftersom samma instans
<code>xs(0) eq zs(0)</code>	<b>false</b> eftersom olika instanser
<code>(ys.toBuffer :+ new Mutant).apply(0).int</code>	<code>0</code> eftersom den ej djupkopierade kopian av typen <code>ArrayBuffer</code> refererar samma instans på första platsen som både <code>ys</code> och <code>xs</code> och <code>x(0).int</code> blev noll i en tilldelning på rad 5 i REPL-körningen

Observera alltså att kopiering med `toArray`, `toVector`, `toBuffer`, etc. *inte är djup*, d.v.s. det är bara instansreferenserna som kopieras och inte själva instanserna.

b)

```
def deepCopy(xs: Array[Mutant]): Array[Mutant] =
  val result = Array.ofDim[Mutant](xs.length) //fylld med null-referenser
  var i = 0
  while i < xs.length do
    result(i) = new Mutant(xs(i).int) //kopier med samma innehåll på samma plats
    i += 1
  result
```

Det går också bra att skapa resultatarrayen med `new Array[Mutant](xs.length)`. Du kan också använda `size` i stället för `length`.

c)

```
1 scala> class Mutant(var int: Int = 0)
2 // defined class Mutant
3
4 scala> def deepCopy(xs: Array[Mutant]): Array[Mutant] =
5   |   val result = Array.ofDim[Mutant](xs.length)
6   |   var i = 0
7   |   while i < xs.length do
8   |     result(i) = new Mutant(xs(i).int)
9   |     i += 1
10  |   result
11
12 scala> val xs = Array.fill(3)(new Mutant)
13 xs: Array[Mutant] = Array(rs$line$2$Mutant@46a123e4, rs$line$2$Mutant@44bc2449,
14 rs$line$2$Mutant@3c28e5b6)
15
16 scala> val ys = deepCopy(xs)
17 ys: Array[Mutant] = Array(rs$line$2$Mutant@14b8a751, rs$line$2$Mutant@7345f97d,
18 rs$line$2$Mutant@554566a8)
19
20 scala> xs(0).int = 5
21
22 scala> ys(0).int
23 val res0: Int = 0
```

d) Nej, eftersom elementen inte kan förändras kan man utan problem dela referenser mellan samlingar. Det finns inte någon möjlighet att det kan ske förändringar som påverkar flera samlingar samtidigt. Dock gör man vanligen (ofta tidsödande) djupkopieringar av

samlingar med förändringsbara element för att kunna vara säkra på att den ursprungliga samlingen inte förändras.

**Lösn. uppg. 5.** Uppdatering av sekvenser.

a)

{ buf(0) = -1; buf(0) }	1	↪ D	-1
{ xs(0) = -1; xs(0) }	2	↪ A	error: value update is not a member
buf.update(1, 5)	3	↪ F	(): Unit
xs.updated(0, 5)	4	↪ B	Vector(5, 2, 3, 4)
{ buf += 5; buf }	5	↪ C	ArrayBuffer(-1, 5, 3, 4, 5)
{ xs += 5; xs }	6	↪ G	error: value += is not a member
xs.patch(1, Vector(-1, 5), 3)	7	↪ E	Vector(1, -1, 5)
xs	8	↪ H	Vector(1, 2, 3, 4)

b)

```
def insert(xs: Array[Int], elem: Int, pos: Int): Array[Int] =
  xs.patch(from = pos, other = Array(elem), replaced = 0)
```

c) Pseudokoden nedan är skriven så att den kompilerar fast den är ofärdig.

```
def insert(xs: Array[Int], elem: Int, pos: Int): Array[Int] =
  val result = ??? /* ny array med plats för ett element mer än i xs */
  var i = 0
  while(???){/* kopiera elementen före plats pos och öka i */}
  if i < result.length then /* lägg elem i result på plats i */
  while(???){/* kopiera över resten */}
  result
```

d)

```
def insert(xs: Array[Int], elem: Int, pos: Int): Array[Int] =
  val result = new Array[Int](xs.length + 1)
  var i = 0
  while i < pos && i < xs.length do { result(i) = xs(i); i += 1 }
  if i < result.length then { result(i) = elem; i += 1 }
  while i < result.length do { result(i) = xs(i - 1); i += 1 }
  result
```

```
1 scala> insert(Array(1, 2), 0, pos = -1)
2 val res2: Array[Int] = Array(0, 1, 2)
3
4 scala> insert(Array(1, 2), 0, pos = 0)
5 val res3: Array[Int] = Array(0, 1, 2)
6
7 scala> insert(Array(1, 2), 0, pos = 1)
8 val res4: Array[Int] = Array(1, 0, 2)
9
```

```

10 scala> insert(Array(1, 2), 0, pos = 2)
11 val res5: Array[Int] = Array(1, 2, 0)
12
13 scala> insert(Array(1, 2), 0, pos = 42)
14 val res7: Array[Int] = Array(1, 2, 0)

```

### Lösn. uppg. 6. Jämföra strängar i Scala.

a)

```

1 true
2 true
3 true
4 true
5 true
6 false

```

b) *s1* kommer först.

### Lösn. uppg. 7. Linjärsökning enligt olika sökkriterier.

a)

<code>xs.indexOf(0)</code>	1	↪	F	5
<code>xs.indexOf(6)</code>	2	↪	B	-1
<code>xs.indexWhere(_ &lt; 2)</code>	3	↪	H	4
<code>xs.indexWhere(_ != 5)</code>	4	↪	I	1
<code>xs.find(_ == 1)</code>	5	↪	D	Some(1)
<code>xs.find(_ == 6)</code>	6	↪	J	None
<code>xs.contains(0)</code>	7	↪	C	<b>true</b>
<code>xs.filter(_ == 1)</code>	8	↪	A	Vector(1, 1)
<code>xs.filterNot(_ &gt; 1)</code>	9	↪	E	Vector(1, 0, 1)
<code>xs.zipWithIndex.filter(_._1 == 1).map(_._2)</code>	10	↪	G	Vector(4, 6)

b) Med en boolesk variabel found:

```

def indexOf(xs: Vector[String], p: String => Boolean): Int =
  var found = false
  var i = 0
  while i < xs.length && !found do
    found = p(xs(i))
    i += 1
  if found then i - 1 else -1

```

Eller utan found:

```

def indexOf(xs: Vector[String], p: String => Boolean): Int =
  var i = 0
  while i < xs.length && !p(xs(i)) do i += 1
  if i == xs.length then -1 else i

```

Eller så kanske man vill börja bakifrån; lösningen nedan är nog enklare att fatta (?) och definitivt mer koncis, men uppfyller *inte* kravet att returnera index för *första* förekomsten som det står i uppgiften. Men om sammanhanget tillåter att vi returnerar *något* index för vilket predikatet gäller, eller om man faktiskt har kravet att leta bakifrån, så funkar detta:

```
def indexOf(xs: Vector[String], p: String => Boolean): Int =
  var i = xs.length - 1
  while i >= 0 && !p(xs(i)) do i -= 1
  i
```

Eller så kan man göra på flera andra sätt. När du ska implementera algoritmer, både på programmeringstentan och i yrkeslivet som systemutvecklare, finns det ofta många olika sätt att lösa uppgiften på som har olika egenskaper, fördelar och nackdelar. Det viktiga är att lösningen fungerar så gott det går enligt kraven, att koden är begriplig för människor och att implementationen inte är så ineffektiv att användarna tröttnar i sin väntan på resultatet...

### Lösn. uppg. 8. Labbförberedelse: Implementera heltalsregistrering i Array.

a)

```
def registreraTärningskast(xs: Seq[Int]): Vector[Int] =
  val result = Array.fill(6)(0)
  xs.foreach{ x =>
    require(x >= 1 && x <= 6, "tärningskast ska vara mellan 1 & 6")
    result(x - 1) += 1
  }
  result.toVector
```

b)

```
1 scala> registreraTärningskast(kasta(1000))
2 val res0: Vector[Int] = Vector(171, 163, 166, 152, 184, 164)
3
4 scala> registreraTärningskast(kasta(1000))
5 val res1: Vector[Int] = Vector(163, 161, 158, 174, 161, 183)
```

### Lösn. uppg. 9. Inbyggda metoder för sortering.

'a' < 'A'	1	↔ E	false
"AÄÖö" < "AÅÖö"	2	↔ H	true
xs.sorted.head	3	↔ C	-1
xs.sorted.reverse.head	4	↔ G	3
ys.sorted.head	5	↔ I	"ak"
zs.indexOf('a')	6	↔ B	1
ps.sorted.head.förnamn.take(2)	7	↔ D	error: ...
ps.sortBy(_.förnamn).apply(1).förnamn.take(2)	8	↔ A	"ka"
xs.sortWith((x1,x2) => x1 > x2).indexOf(3)	9	↔ F	0

Det blir fel i uttrycket ovan som försöker sortera en sekvens med instanser av Person direkt med metoden sorted:

```

1 scala> ps.sorted
2 No implicit Ordering defined for Person.

```

Det blir fel eftersom kompilatorn inte hittar någon ordningsdefinition för dina egna klasser. Senare i kursen ska vi se hur vi kan skapa egna ordningar om man vill få sorted att fungera på sekvenser med instanser av egna klasser, men ofta räcker det fint med sortBy och sortByWith.

### Lösn. uppg. 10. Inbyggd metod för blandning.

a) Random.shuffle returnerar en ny blandad sekvenssamling av samma typ. Ordningen i den ursprungliga samlingen påverkas inte.

b) Exempel på användning av random.shuffle:

```

1 scala> import scala.util.Random
2
3 scala> val xs = Vector("Sten", "Sax", "Påse")
4 val xs: Vector[String] = Vector(Sten, Sax, Påse)
5
6 scala> (1 to 10).foreach(_ => println(Random.shuffle(xs).mkString(" ")))
7 Sax Påse Sten
8 Sten Påse Sax
9 Sten Sax Påse
10 Sten Sax Påse
11 Sten Påse Sax
12 Sten Påse Sax
13 Sax Sten Påse
14 Sten Påse Sax
15 Sax Påse Sten
16 Sax Påse Sten
17
18 scala> (1 to 5).map(_ => Random.shuffle(1 to 6))
19 val res1: IndexedSeq[IndexedSeq[Int]] =
20   Vector(Vector(5, 2, 1, 4, 3, 6), Vector(6, 5, 4, 2, 1, 3),
21     Vector(3, 1, 4, 6, 5, 2), Vector(3, 2, 6, 5, 1, 4),
22     Vector(5, 3, 4, 6, 1, 2))
23
24 scala> (1 to 1000).map(_ => Random.shuffle(1 to 6).head).count(_ == 6)
25 val res2: Int = 168

```

### Lösn. uppg. 11. Repeterade parametrar.

a)

```

1 scala> def stringSizes(xs: String*): Vector[Int] = xs.map(_.size).toVector
2 def stringSizes(xs: String*): Vector[Int]
3
4 scala> stringSizes("hej")
5 val res0: Vector[Int] = Vector(3)
6
7 scala> stringSizes("hej", "på", "dej", "")
8 val res1: Vector[Int] = Vector(3, 2, 3, 0)
9
10 scala> stringSizes()
11 val res2: Vector[Int] = Vector()

```

Anrop med tom argumentlista ger en tom heltalssekvens.

b)



```
1 scala> val xs = Vector("hej","på","dej", "")
2 val xs: Vector[String] = Vector(hej, på, dej, "")
3
4 scala> stringSizes(xs: _*)
5 val res0: Vector[Int] = Vector(3, 2, 3, 0)
6
7 scala> stringSizes(Vector(): _*)
8 val res1: Vector[Int] = Vector()
```

Ja, det funkar fint med tom sekvens.

## L.7.2 Extrauppgifter; träna mer

**Lösn. uppg. 12.** *Registrering av booleska värden. Singla slant.*

a)

```
def registerCoinFlips(xs: Seq[Boolean]): (Int, Int) =  
  val result = Array.fill(2)(0)  
  xs.foreach(x => if (x) result(0) += 1 else result(1) += 1)  
  (result(0), result(1))
```

b)

**Lösn. uppg. 13.** *Kopiering och tillägg på slutet.*

```
def copyAppend(xs: Array[Int], x: Int): Array[Int] =  
  val ys = new Array[Int](xs.length + 1)  
  var i = 0  
  while i < xs.length do  
    ys(i) = xs(i)  
    i += 1  
  ys(xs.length) = x  
  ys
```

De två buggarna i algoritmen finns (1) i villkoret som ska vara strikt mindre än och (2) inne i loopen där uppräknings av loppvariabeln saknas.

**Lösn. uppg. 14.** *Kopiera och reversera sekvens.*

a)

```
def seqReverseCopy(xs: Array[Int]): Array[Int] =  
  val n = xs.length  
  val ys = new Array[Int](n)  
  var i = 0  
  while i < n do  
    ys(n - i - 1) = xs(i)  
    i += 1  
  ys
```

b)

```
def seqReverseCopy(xs: Array[Int]): Array[Int] =  
  val n = xs.length  
  val ys = new Array[Int](n)  
  for i <- (n - 1) to 0 by -1 do  
    ys(n - i - 1) = xs(i)  
  ys
```

**Lösn. uppg. 15.** *Kopiera alla utom ett.*

**Indata :** En sekvens  $xs$  av typen `Array[Int]` och  $pos$

**Utdata:** En ny sekvens av typen `Array[Int]` som är en kopia av  $xs$  fast med elementet på plats  $pos$  borttaget

```

1  $n \leftarrow$  antalet element  $xs$ 
2  $ys \leftarrow$  en ny Array[Int] med plats för  $n - 1$  element
3 for  $i \leftarrow 0$  to  $pos - 1$  do
4    $ys(i) \leftarrow xs(i)$ 
5 end
6  $ys(pos) \leftarrow x$ 
7 for  $i \leftarrow pos + 1$  to  $n - 1$  do
8    $ys(i - 1) \leftarrow xs(i)$ 
9 end
10  $ys$ 

```

```

def removeCopy(xs: Array[Int], pos: Int): Array[Int] =
  val n = xs.size
  val ys = Array.fill(n - 1)(0)
  for i <- 0 until pos do
    ys(i) = xs(i)
  for i <- (pos + 1) until n do
    ys(i - 1) = xs(i)
  ys

```

### Lösn. uppg. 16. Borttagning på plats i array.

**Indata :** En sekvens  $xs$  av typen `Array[Int]`, en position  $pos$  och ett utfyllnadsvärde  $pad$

**Utdata:** En uppdaterad sekvens av  $xs$  där elementet på plats  $pos$  tagits bort och efterföljande element flyttas ett steg mot lägre index med ett sista elementet som tilldelats värdet av  $pad$

```

1  $n \leftarrow$  antalet element  $xs$ 
2 for  $i \leftarrow pos + 1$  to  $n - 1$  do
3    $xs(i - 1) \leftarrow xs(i)$ 
4 end
5  $xs(n - 1) \leftarrow pad$ 

```

```

def remove(xs: Array[Int], pos: Int, pad: Int = 0): Unit =
  val n = xs.size
  for i <- (pos + 1) until n do
    xs(i - 1) = xs(i)
  xs(n - 1) = pad

```

### Lösn. uppg. 17. Kopiering och insättning.

a)

```

def insertCopy(xs: Array[Int], x: Int, pos: Int): Array[Int] =
  val n = xs.size
  val ys = Array.ofDim[Int](n + 1)
  for i <- 0 until pos do

```

```

ys(i) = xs(i)
ys(pos) = x
for i <- pos until n do
  ys(i + 1) = xs(i)
ys

```

b) pos måste vara 0.

c)

```
1 java.lang.ArrayIndexOutOfBoundsException: -1
```

d) Elementet  $x$  läggs till på slutet av arrayen, alltså kommer den returnerande arrayen vara större än den som skickades in.

e)

```
1 java.lang.ArrayIndexOutOfBoundsException: 5
```

Man får `ArrayIndexOutOfBoundsException` då indexeringen är utanför storleken hos arrayen.

### Lösn. uppg. 18. *Insättning på plats i array.*

**Indata :** En sekvens  $xs$  av typen `Array[Int]` och heltalen  $x$  och  $pos$

**Utdata:**  $xs$  uppdaterat på plats, där elementet  $x$  har satts in på platsen  $pos$  och efterföljande element flyttas ett steg där sista elementet försvinner

```

1  $n \leftarrow$  antalet element i  $xs$ 
2  $ys \leftarrow$  en klon av  $xs$ 
3  $xs(pos) \leftarrow x$ 
4 for  $i \leftarrow pos + 1$  to  $n - 1$  do
5    $xs(i) \leftarrow ys(i - 1)$ 
6 end

```

```

def insertDropLast(xs: Array[Int], x: Int, pos: Int): Unit =
  val n = xs.size
  val ys = xs.clone
  xs(pos) = x
  for i <- pos + 1 until n do
    xs(i) = ys(i - 1)

```

### Lösn. uppg. 19. *Fler inbyggda metoder för linjärsökning.*

a)

- `lastIndexOf` är bra om man vill leta bakifrån i stället för framifrån; utan denna hade man annars då behövt använda `xs.reverse.indexOf(e)`
- `indexOfSlice(ys)` letar efter index där en hel sekvens  $ys$  börjar, till skillnad från `indexOf(e)` som bara letar efter ett enstaka element.
- `segmentLength(p, i)` ger längden på den längsta sammanhängande sekvens där alla element uppfyller predikatet  $p$  och sökningen efter en sådan sekvens börjar på plats  $i$
- `xs.maxBy(f)` kör först funktionen  $f$  på alla element  $i$  i  $xs$  och letar sedan upp det största värdet; motsvarande `minBy(f)` ger minimum av  $f(e)$  över alla element  $e$  i  $xs$

b) –

### L.7.3 Fördjupningsuppgifter; utmaningar

**Lösn. uppg. 20.** *Fixa svensk sorteringsordning av ÄÅÖ.*

**Lösn. uppg. 21.** *Fibonacci-sekvens med ListBuffer.*

a)

```
def fib(max: Long): List[Long] =
  val xs = scala.collection.mutable.ListBuffer.empty[Long]
  xs.prependAll(Vector(1, 1))
  while xs.head < max do xs.prepend(xs.take(2).sum)
  xs.reverse.drop(1).toList
```

b)

```
1 scala> fib(Int.MaxValue).size
2 val res0: Int = 46
```

c)

```
def fibBig(max: BigInt): List[BiInt] =
  val xs = scala.collection.mutable.ListBuffer.empty[BiInt]
  xs.prependAll(Vector(BiInt(1), BiInt(1)))
  while xs.head < max do xs.prepend(xs.take(2).sum)
  xs.reverse.drop(1).toList
```

```
1 scala> fibBig(Long.MaxValue).size
2 val res0: Int = 92
3
4 scala> fibBig(BiInt(Long.MaxValue).pow(64)).size
5 val res1: Int = 5809
6
7 scala> fibBig(BiInt(Long.MaxValue).pow(128)).last
8 val res2: BiInt = 466572805528355449194553611102863153950720005186045547177525242118545194247268198
9
10 scala> fibBig(BiInt(Long.MaxValue).pow(128)).last.toString.size
11 val res3: Int = 2428
12
13 scala> fibBig(BiInt(Long.MaxValue).pow(256)).last.toString.size
14 val res4: Int = 4856
15
16 scala> fibBig(BiInt(Long.MaxValue).pow(1024)).last.toString.size
17 java.lang.OutOfMemoryError: Java heap space
```

**Lösn. uppg. 22.** *Omvända sekvens på plats.*

```
def reverseChars(xs: Array[Char]): Unit =
  val n = xs.length
  for i <- 0 to (n/2 - 1) do
    val temp = xs(i)
    xs(i) = xs(n - i - 1)
    xs(n - i - 1) = temp
```

**Lösn. uppg. 23.** *Palindrompredikat.*

a) Omvändning med reverse kan kräva genomgång av hela strängen en gång samt minnesutrymme för kopian. Innehållstestet kräver ytterligare en genomgång. (Detta är i och för sig inget stort problem eftersom världens längsta palindrom inte är längre än 19 bokstäver och är ett obskyrt finskt ord som inte ofta yttras i dagligt tal. Vilket?)

b)

```
def isPalindrome(s: String): Boolean =
  val n = s.length
  var foundDiff = false
  var i = 0
  while i < n/2 && !foundDiff do
    foundDiff = s(i) != s(n - i - 1)
    i += 1
  !foundDiff
```

**Lösn. uppg. 24.** *Fler användbara sekvenssamlingsmetoder.*

```
1 scala> val xs = Vector.tabulate(10)(i => math.pow(2, i).toInt)
2 xs: Vector[Int] = Vector(1, 2, 4, 8, 16, 32, 64, 128, 256, 512)
3
4 scala> xs.forall(_ < 1024)
5 val res0: Boolean = true
6
7 scala> xs.exists(_ == 3)
8 val res1: Boolean = false
9
10 scala> xs.count(_ > 64)
11 val res2: Int = 3
12
13 scala> xs.zipWithIndex.take(5)
14 val res3: Vector[(Int, Int)] = Vector((1,0), (2,1), (4,2), (8,3), (16,4))
```

**Lösn. uppg. 25.** *Arrays don't behave, but ArraySeqs do!*

a) `xs` erbjuder innehållslighet och har typen `Seq[Int]` med den underliggande typen `ArraySeq[Int]`. Det går inte att göra tilldelning av element i en `ArraySeq` eftersom metoden `update` saknas, och den är oföränderlig. Den uppdateras därför inte när den ursprungliga arrayen uppdateras.

```
1 scala> val as1 = Array(1,2,3)
2 val as1: Array[Int] = Array(1, 2, 3)
3
4 scala> val as2 = Array(1,2,3)
5 val as2: Array[Int] = Array(1, 2, 3)
6
7
8 scala> val (xs1, xs2) = (as1.toSeq, as2.toSeq)
9 val xs1: Seq[Int] = ArraySeq(1, 2, 3)
10 val xs2: Seq[Int] = ArraySeq(1, 2, 3)
11
12 scala> as1 == as2
13 val res0: Boolean = false
14
15 scala> xs1 == xs2
16 val res1: Boolean = true
```

```
17
18 scala> as1(0) = 42
19
20 scala> xs1
21 val res2: Seq[Int] = ArraySeq(1, 2, 3)
22
23 scala> xs1(0) = 42
24 value update is not a member of Seq[Int]
```

b) Vid repeterade parametrar får man en ArraySeq.

```
1 scala> def f(xs: Int*) = xs
2 def f(xs: Int*): Seq[Int]
3
4 scala> println(f(1,2,3))
5 ArraySeq(1, 2, 3)
```

c) Det går inte att ha en generisk array som funktionsresultat utan att bifoga kontextgränsen ClassTag i typparametern för att kompilatorn ska kunna generera kod för den typkonvertering som krävs under runtime av JVM. Se exempel här:

<http://docs.scala-lang.org/overviews/collections/arrays.html>

**Lösn. uppg. 26.** Sekvenssamlingen List är nästan dubbelt så snabb vid bearbetning i början men ungefär 1000 gånger långsammare vid bearbetning i slutet av en sekvens med 100000 element.

Olika körningar går olika snabbt på JVM bl.a. p.g.a optimeringar som sker när JVM-en "värms upp" och den så kallade Just-In-Time-kompileringen gör sitt mäktiga jobb. Det går ibland plötsligt väsentligt långsammare när skräpsamlaren tvingas göra tidsödande storstädning av minnet.

**Lösn. uppg. 27.** –

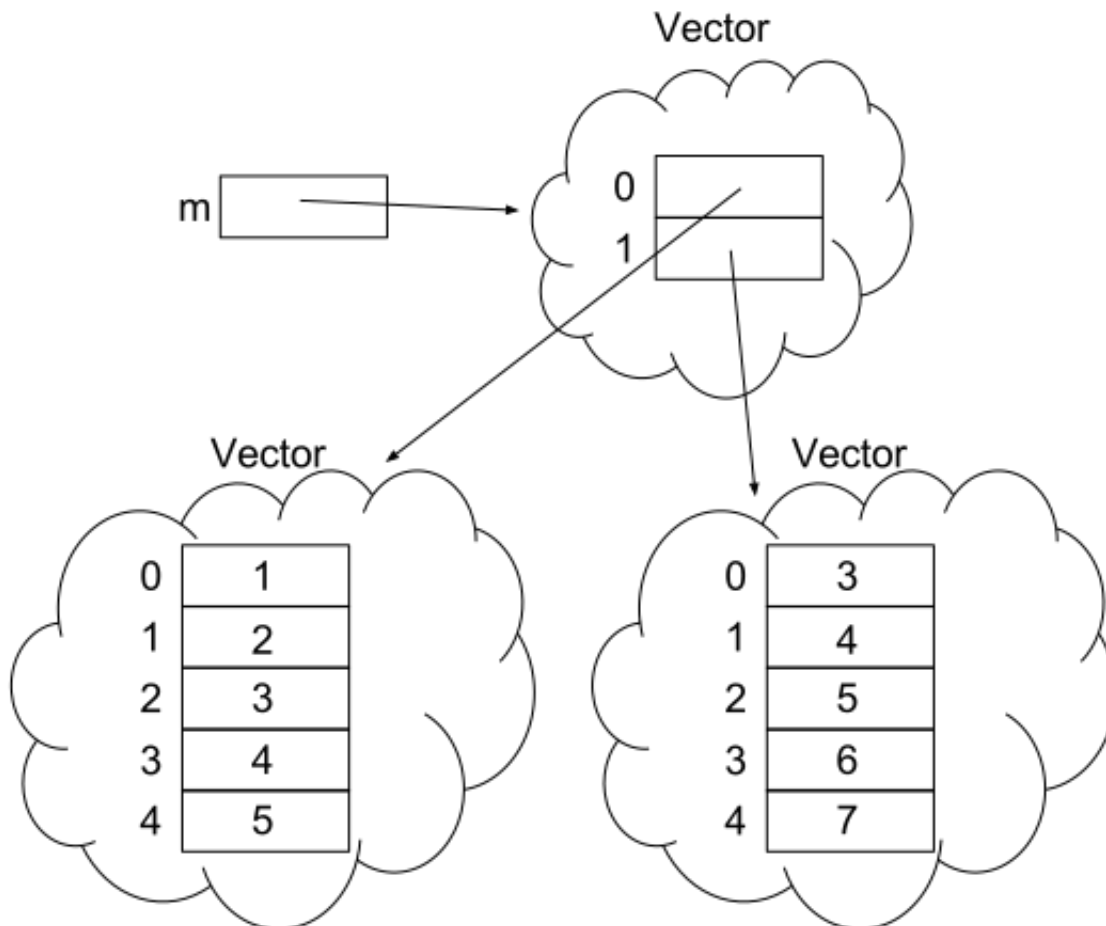
## L.8 Lösning matrices

### L.8.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** Para ihop begrepp med beskrivning.

matris	1	↪ A	indexerbar datastruktur i två dimensioner
radvektor	2	↪ F	matris av dimension $1 \times m$ med $m$ horisontella värden
kolumnvektor	3	↪ G	matris av dimension $m \times 1$ med $m$ vertikala värden
kolonn	4	↪ C	annat ord för kolumn
generisk	5	↪ B	har abstrakt typparameter, typen är generell
typargument	6	↪ D	konkret typ, binds till typparameter vid kompilering
typhärledning	7	↪ E	kompilatorn beräknar typ ur sammanhanget

**Lösn. uppg. 2.** Skapa matriser med hjälp av nästlade samlingar.



a)  
 Typ: `Vector[Vector[Int]]`  
 Värde: `Vector(Vector(1, 2, 3, 4, 5), Vector(3, 4, 5, 6, 7))`  
 Dimensioner:  $2 \times 5$   
 Inom matematiken sker indexering enligt konvention med 1 som lägsta index. I scala är lägsta index 0, man använder s.k. 0-indexering.<sup>3</sup>

<sup>3</sup>Detta är inte fallet i alla programmeringsspråk, vilket du kan läsa mer om på [https://en.wikipedia.org/wiki/Array\\_data\\_type#Index\\_origin](https://en.wikipedia.org/wiki/Array_data_type#Index_origin)



b)

```

1 scala> val m = Vector((1 to 5).toVector, (3 to 7).toVector)
2 m: Vector[Vector[Int]] = Vector(Vector(1, 2, 3, 4, 5), Vector(3, 4, 5, 6, 7))
3
4 scala> m.apply(0).apply(1)
5 res4: Int = 2
6
7 scala> m(1)
8 res5: Vector[Int] = Vector(3, 4, 5, 6, 7)
9
10 scala> m(1)(4)
11 res6: Int = 7

```

c)

m2: Vector[Vector[Int]]

m3: Vector[Vector[AnyVal]]

m4: Vector[Vector[Any]]

m5: Vector[Vector[Int]]

d) m5,  $42 \times 2$ **Lösn. uppg. 3.** Skapa och iterera över matriser.

a)

```
def throwDie: Int = (math.random() * 6).toInt + 1
```

Eller:

```
def throwDie: Int = scala.util.Random.nextInt(6) + 1
```

b) Matrisdimension i matematisk notation:  $1000 \times 5$ , vilket motsvarar en matris med 1000 rader och 5 kolumner.

c)

```

ds1: IndexedSeq[IndexedSeq[Int]]
ds2: IndexedSeq[IndexedSeq[Int]]
ds3: IndexedSeq[Vector[Int]]
ds4: IndexedSeq[Vector[Int]]
ds5: Vector[Vector[Int]]
ds6: Vector[Vector[Int]]

```

IndexedSeq och Vector ovan finns i paketet `scala.collection.immutable`

d)

```
def roll(n: Int) = Vector.fill(n)(throwDie).sorted
```

e)

```
def isYatzy(xs: Vector[Int]): Boolean = xs.forall(_ == xs(0))
```

f)

```
def diceMatrix(m: Int, n: Int): Vector[Vector[Int]] =
  Vector.fill(m)(roll(n))
```

g)

```
def diceMatrixToString(xss: Vector[Vector[Int]]): String =
  xss.map(_.mkString(" ")).mkString("\n")
```

h)

```
def filterYatzy(xss: Vector[Vector[Int]]): Vector[Vector[Int]] =
  xss.filter(isYatzy)
```

i)

```
def yatzyPips(xss: Vector[Vector[Int]]): Vector[Int] =
  filterYatzy(xss).map(_.head)
```

**Lösn. uppg. 4.** *En oföränderlig, generisk matris-klass till veckans laboration [life](#).*

- a) Typen på m blir Matrix.
- b) Typen på e blir String.
- c) Man behöver ändra på 3 ställen från String till Int.
- d) Generisk matris Matrix[T] för element av godtycklig typ T:

```
case class Matrix[T](data: Vector[Vector[T]]):
  def apply(row: Int, col: Int): T = data(row)(col)

object Matrix:
  def fill[T](dim: (Int, Int))(value: T): Matrix[T] =
    Matrix[T](Vector.fill(dim._1, dim._2)(value))
```

- e) Tack vare kompilatorns typinferens så får bm typen Matrix[Boolean].
- f) Typen på be blir Boolean.
- g) h) i) j) k) är alla implementerade i koden nedan:

```
case class Matrix[T](data: Vector[Vector[T]]):
  require(data.forall(row => row.length == data(0).length))

  val dim: (Int, Int) = (data.length, data(0).length)

  def apply(row: Int, col: Int): T = data(row)(col)

  def updated(row: Int, col: Int)(value: T): Matrix[T] =
    Matrix(data.updated(row, data(row).updated(col, value)))

  def foreachIndex(f: (Int, Int) => Unit): Unit =
    for r <- data.indices; c <- data(r).indices do f(r, c)

  override def toString =
    s""""Matrix of dim $dim:\n${ data.map(_.mkString(" ")).mkString("\n") }""""

object Matrix:
  def fill[T](dim: (Int, Int))(value: T): Matrix[T] =
    Matrix[T](Vector.fill(dim._1, dim._2)(value))
```

## L.8.2 Extrauppgifter; träna mer

### Lösn. uppg. 5. Imperativa matrisalgoritmer.

a)

```
def isYatzy(xs: Vector[Int]): Boolean =
  var foundDiff = false
  var i = 0
  while (i < xs.size && !foundDiff) do
    foundDiff = xs(i) != xs(0)
    i += 1
  end while
  !foundDiff
```

b) Funktionen går igenom varje matrisrad, där den i sin tur går igenom varje element på raden och lägger till i StringBuilder-objektet. Om det inte är det sista elementet på raden läggs även ett blanktecken till, annars läggs ett nyradstecken till. Undantaget är sista raden, där inget nyradstecken läggs till. Slutligen konverteras StringBuilder-objektet till en String som returneras.

Är xss tom blir xss.indices en tom Range och den yttre **for**-loopen hoppas över och en tom sträng returneras. Är alla rader tomma hoppas i stället de inre **for**-looparna över, med samma resultat.

*Fördel:* StringBuilder är snabbare vid tillägg på slutet vid stora strängar (men här kommer det inte märkas eftersom strängen är så liten).

*Nackdel:* StringBuilder-koden uppfattas av många som svårare att läsa.

c)

```
def filterYatzy(xss: Vector[Vector[Int]]): Vector[Vector[Int]] =
  var result: Vector[Vector[Int]] = Vector()
  for i <- xss.indices if isYatzy(xss(i)) do result = result :+ xss(i)
  result
```

d) Varje looprunda ger en vektor xss(i) om filtervillkoret är uppfyllt och resultatet av **for**-uttrycket blir en vektor med vektorer som är yatzyslag.

### Lösn. uppg. 6. Strängtabell med kolumnrubriker.

a)

```
case class Table(
  data: Vector[Vector[String]],
  headings: Vector[String],
  sep: Char
):

  val dim: (Int, Int) = (data.size, headings.size)

  def apply(r: Int, c: Int): String = data(r)(c)

  def row(r: Int): Vector[String] = data(r)

  def col(c: Int): Vector[String] = data.map(r => r(c))
```

```

lazy val indexOfHeading: Map[String, Int] = headings.zipWithIndex.toMap

def col(h: String): Vector[String] = col(indexOfHeading(h))

def values(h: String): Vector[String] = col(h).distinct.sorted

override def toString: String =
  val s = sep.toString
  headings.mkString(s) + "\n" + data.map(_.mkString(s)).mkString("\\n")

object Table:
  def fromFile(fileName: String, sep: Char = ';'): Table =
    val lines = scala.io.Source.fromFile(fileName).getLines.toVector
    val matrix = lines.map(_.split(sep).toVector)
    new Table(matrix.tail, matrix.head, sep)

```

b)

```

@main
def run(fileName: String, separator: String): Unit =
  require(separator.length == 1, "separator ska vara exakt ett tecken")
  val t = Table.fromFile(fileName, separator.head)
  val counts: Vector[Vector[String]] =
    (0 until t.dim._2)
      .map(i => t.values(t.headings(i))
        .map(x => s"$x: ${t.col(i).count(_ == x)}"))
      .toVector
  for (i <- 0 until t.dim._2) do
    println(s"\nColumn: ${i + 1}, ${t.headings(i)}:")
    for (j <- 0 until counts(i).length) do
      println(counts(i)(j))

```

**Lösn. uppg. 7.** Skapa ett yatzy-spel för användning i terminalen.

—

### L.8.3 Fördjupningsuppgifter; utmaningar

#### Lösn. uppg. 8. Generiska funktioner.

- a)
1. –
  2. Strängrepresentationen av 42 spegelvänds
  3. "hej" spegelvänds - toString av en sträng ger en likadan sträng
  4. –
  5. Gurk-objektets strängrepresentation spegelvänds
  6. Funktionens typparameter matchar inte parameterns typ: 42 är ingen sträng
  7. Implicit typkonvertering till Double sker för att stämma överens med typparametern, vilket ger en strängrepresentation med decimal

- b)
1. En funktion definieras så att den tar emot två andra funktioner som argument, sätter ihop dem, och matar in ett tredje argument till den sammansatta funktionen.
  2. En funktion som inkrementerar ett heltal med 1 definieras.
  3. En funktion som halverar ett flyttal definieras.
  4. 42 matas in i inc() och resultatet (43) matas vidare till half(). Inuti half() sker implicit typkonvertering till Double då talet divideras med ett flyttal (2.0) och resultatet blir  $43.0 / 2.0$ , alltså 21.5.
  5. Resultatet från half() är av typ Double, medan inc() tar emot ett argument av typ Int. Då flyttal generellt inte kan konverteras till heltal utan informationsförlust sker ingen implicit konvertering, istället sker ett kompileringsfel.

c)

```
def inc(x: Double): Double = x + 1.0
```

Nu ges kompileringsfel på rad 4 istället, vilket kan lösas med följande ändring:

```
def half(x: Double): Double = x / 2.0
```

#### Lösn. uppg. 9. Generiska klasser.

a) –

b)

```
class Cell[T](var value: T):  
  override def toString = "Cell(" + value + ")"  
  def concat[U](that: Cell[U]): Cell[String] =  
    Cell(s"$value${that.value}")
```

c) Endast celler med samma typparameter kan nu konkateneras. Eftersom `concat()` returnerar ett objekt av typ `Cell[String]` kan ett ojämnt antal celler med någon annan typparameter än `String` alltså inte längre konkateneras. Är antalet jämnt går det att konkatenera dem parvis och sedan konkatenera de returnerade `Cell[String]`-objekten, men det är något omständigt.

**Lösn. uppg. 10.** *Implementera fler generiska metoder i `Matrix[T]`.*

```
case class Matrix[T](data: Vector[Vector[T]]):
  require(data.forall(row => row.size == data(0).size))

  val dim: (Int, Int) = (data.length, data(0).length)

  def apply(row: Int, col: Int): T = data(row)(col)

  def updated(row: Int, col: Int)(value: T): Matrix[T] =
    Matrix(data.updated(row, data(row).updated(col, value)))

  def foreach(f: T => Unit): Unit = data.foreach(_.foreach(f))

  def foreachIndex(f: (Int, Int) => Unit): Unit =
    for r <- data.indices; c <- data(r).indices do f(r, c)

  def map[U](f: T => U): Matrix[U] = Matrix(data.map(_.map(f)))

  def mapIndex[U](f: (Int, Int) => U): Matrix[U] =
    var result = Matrix.fill(dim)(f(0,0))
    for
      r <- data.indices
      c <- data(r).indices
    do
      result = result.updated(r, c)(f(r, c))
    end for
    result

  override def toString =
    s"""Matrix of dim $dim:\n${ data.map(_.mkString(" ")).mkString("\n") }"""

object Matrix:
  def fill[T](dim: (Int, Int))(value: T): Matrix[T] =
    Matrix[T](Vector.fill(dim._1, dim._2)(value))
```

## L.9 Lösning lookup

### L.9.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** *Para ihop begrepp med beskrivning.*

mängd	1	↔	H	oordnad samling med unika element
nyckel-värde-tabell	2	↔	F	oordnad samling av mappningar med unika nycklar
mappning	3	↔	G	nyckel -> värde
nyckel	4	↔	C	en unik identifierare
persistens	5	↔	D	egenskapen att finnas kvar efter programmets avslut
serialisera	6	↔	E	koda objekt till avkodningsbar sekvens av symboler
de-serialisera	7	↔	B	avkoda symbolsekvens och återskapa objekt i minnet
linjärsöka	8	↔	A	leta i sekvens tills sökkriteriet är uppfyllt

**Lösn. uppg. 2.** *Vad är en mängd?* En mängd är en samling som snabbt kan ge svaret på frågan om ett visst element ingår i samlingen eller ej. Elementen i en mängd är unika. Tilläg av redan existerande element ignoreras. En mängd är inte en sekvens, eftersom traversering med t.ex. map eller foreach inte (nödvändigtvis) sker i den ordning som elementen gavs när mängden konstruerades eller uppdaterades.

**Lösn. uppg. 3.** *Använda mängder.*

Set(1, 2) ++ Set(1, 2)	1	↔	I	Set(1, 2)
(1 to 3).toSet	2	↔	G	Set(1) + 2 + 3
Vector.fill(3)(1).toSet	3	↔	F	Set(1, 2) - 2
Set(1, 2, 3) diff Set(1, 2)	4	↔	B	Set(3)
(1 to 7).toSet.apply(8)	5	↔	H	<b>false</b>
Set(1, 2, 3).sorted	6	↔	D	error: ...
Set(2,4) subsetOf (1 to 7).toSet	7	↔	E	<b>true</b>
Set(1, -1, 2, -2).map(_ .abs).sum	8	↔	A	3
Set(1, 1, 1, 1, 1, 5).sum	9	↔	C	6

**Lösn. uppg. 4.** *Räkna unika ord med hjälp av en mängd.*

a)

```
1 scala> val hej = "hej hej hemskt mycket hej"
2 scala> val n = hej.split(' ').toSet.size
3 n: Int = 3
```

b) Metoden `distinct` returnerar en sekvens med unika element och bibehållen ursprunglig ordning.

**Lösn. uppg. 5.** Skapa 2-tupler med metoden -> som kan uttalas "mappas till".

a) Ja, fabriksmetoden returnerar ett helt vanligt par:

```
scala> val härBorJag = "Skåne" -> "Lund"
val härBorJag: (String, String) = (Skåne,Lund)

scala> härBorJag._1
val res0: String = Skåne

scala> härBorJag._2
val res1: String = Lund
```

b)

```
val huvudstad = Vector(
  "Sverige" -> "Stockholm",
  "Danmark" -> "Köpenhamn",
  "Grönland" -> "Nuuk",
  "Skåne" -> "Lund"
)
```

c)

```
1 scala> huvudstad(3)._2
2 val res2: String = Lund
```

**Lösn. uppg. 6.** Linjärsöka efter nyckel i sekvens av mappningar.

a)

```
def lookupIndex(xs: Vector[(String, String)])(key: String): Int =
  xs.indexWhere(_._1 == key)
```

b)

```
1 scala> val i = lookupIndex(huvudstad)("Skåne")
2 val i: Int = 3
3
4 scala> huvudstad(i)._2
5 val res2: String = Lund
```

Eller med funktioner som återanvändbara dellösningar:

```
1 scala> val indexOf = lookupIndex(huvudstad) _
2
3 scala> def capital(key: String) = huvudstad(indexOf(key))._2
4
5 scala> capital("Skåne")
6 val res3: String = Lund
7
8 scala> capital("Sverige")
9 val res4: String = Stockholm
```

**Lösn. uppg. 7.** Nyckel-värde-tabell.

```
1 scala> telnr("Fröken Ur")
2 val res0: Long = 464690510
3
```



```

4 scala> :type telnr
5 Map[String,Long]
6
7 scala> :type telnr.toVector
8 Vector[(String, Long)]

```

### Lösn. uppg. 8. Använda nyckel-värdetabell.

a) Nej nyckel-värde-paren lagras i någon speciell ordning som bestäms av en intern, smart lagringsprincip enligt en s.k. hashfunktion<sup>4</sup>, för att åstadkomma snabba uppslagningar av värden från nycklar och vilket normalt inte sammanfaller med ordningen i den sekvens som de skapades ur.

b)

<code>xs(2) + xs(4)</code>	1	↪ A	8
<code>ys(0)</code>	2	↪ C	(10, 11)
<code>xs(0)</code>	3	↪ I	NoSuchElementException
<code>(xs + (0 -&gt; 1)).apply(0)</code>	4	↪ D	1
<code>xs.keySet.apply(2)</code>	5	↪ G	true
<code>xs.isDefinedAt 0</code>	6	↪ H	false
<code>xs.getOrElse(0, 7)</code>	7	↪ B	7
<code>xs.maxBy(_._2)</code>	8	↪ E	(16, 17)
<code>xs.map(p =&gt; p._1 -&gt; -p._2)(8)</code>	9	↪ F	-9

### Lösn. uppg. 9. Registrering i förändringsbar nyckel-värde-tabell.

```

class FreqMapBuilder:
  private val register = scala.collection.mutable.Map.empty[String,Int]
  def toMap: Map[String, Int] = register.toMap
  def add(s: String): Unit =
    register += (s -> (register.getOrElse(s, 0) + 1))

object FreqMapBuilder:
  def apply(xs: String*): FreqMapBuilder =
    val result = new FreqMapBuilder
    xs.foreach(result.add)
    result

```

### Lösn. uppg. 10. Metoden sliding.

a)

```

1 scala> val xs = Vector("fem", "gurkor", "är", "fler", "än", "fyra", "tomater")
2 val xs: Vector[String] =
3   Vector(fem, gurkor, är, fler, än, fyra, tomater)
4
5 scala> xs.sliding(2).toVector

```

<sup>4</sup><https://sv.wikipedia.org/wiki/Hashfunktion>

```

6 val res9: Vector[Vector[String]] =
7   Vector(Vector(fem, gurkor), Vector(gurkor, är), Vector(är, fler), Vector(fler, än), Vector(än, fyr
8
9 scala> xs.sliding(3).toVector
10 val res10: Vector[Vector[String]] =
11   Vector(Vector(fem, gurkor, är), Vector(gurkor, är, fler), Vector(är, fler, än), Vector(fler, än, f
12
13 scala> xs.sliding(10).toVector
14 val res11: Vector[Vector[String]] =
15   Vector(Vector(fem, gurkor, är, fler, än, fyra, tomater))

```

`xs.sliding(n).toVector` skapar en sekvens som innehåller sekvenser av längden `n` som bildas genom att ta varje element och dess `n - 1` efterföljande element.

b)

```

1 scala> xs.sliding(2).map(ys => ys(0) -> ys(1)).toMap
2 val res0: Map[String,String] =
3   Map(är -> fler,
4       än -> fyra,
5       fyra -> tomater,
6       gurkor -> är,
7       fem -> gurkor,
8       fler -> än
9   )

```

Man kan använda tabellen till att slå upp vilket som är efterföljande ord. Det fungerar eftersom alla ord är unika. Om det funnits flera likadana ord med olika efterföljande ord så hade vi behövt skapa en tabell med nycklar som mappar till en samling som registrerar efterföljande ord. Detta ska vi göra på veckans laboration.

**Lösn. uppg. 11.** *Läsa text från fil och webbserverar.*

a)

```

val populationOf = data.tail.map(v => v(0) -> v(1).toInt).toMap
val sizeOf       = data.tail.map(v => v(0) -> v(2).toInt).toMap
val capitalOf    = data.tail.map(v => v(0) -> v(3)).toMap

```

```

1 scala> capitalOf("Sverige")
2 res2: String = Stockholm
3
4 scala> populationOf("Sverige")
5 res3: Int = 9223766
6
7 scala> sizeOf("Sverige")
8 res4: Int = 449964

```

```

1 scala> val filename = "europa.txt"
2 scala> val xs = io.Source.fromFile(filename, "UTF-8").getLines.toVector
3 scala> val data = xs.map(_.split(';').toVector)
4 scala> data.map(_.map(_.take(15).padTo(15,' ')).mkString(" ")).foreach(println)

```

## L.9.2 Extrauppgifter; träna mer

**Lösn. uppg. 12.** –

### L.9.3 Fördjupningsuppgifter; utmaningar

#### Lösn. uppg. 13. Registrering med groupBy.

a) Metoden groupBy skapar en nyckel-värde-tabell där värdena i tabellen är en sekvens med elementen grupperade på ett speciellt sett. Mer precist:

Resultatet av `xs.groupBy(f: K => V)` för en sekvens `xs` av typen `Vector[K]` blir en tabell av typen `Map[V, Vector[K]]` där varje element `e` i `xs` är grupperade i samma tabellvärde om de lika är enligt `f(e)`. Varje grupp får tabellnyckeln `f(e)`.

*Listigt trick:* Om man låter funktionen `f` vara enhetsfunktionen som avbildar varje element på sig själv, alltså `x => x`, så grupperas värdena i samma sekvens om de är lika.

```
1 scala> val xs = Vector(1, 1, 2, 2, 4, 4, 4).groupBy(x => x > 2)
2 val xs: Map[Boolean, Vector[Int]] =
3   Map(false -> Vector(1, 1, 2, 2), true -> Vector(4, 4, 4))
4
5 scala> val ys = Vector(1, 1, 2, 2, 4, 4, 4).groupBy(x => x)
6 val ys: Map[Int, Vector[Int]] =
7   Map(2 -> Vector(2, 2), 4 -> Vector(4, 4, 4), 1 -> Vector(1, 1))
```

b)

```
def freq(xs: Vector[Int]): Map[Int, Int] =
  xs.groupBy(x => x).map(p => p._1 -> p._2.size)
```

Förklaring: metoden groupBy skapar en tabell med par `k, v` där `v` är en sekvens med så många `k` som antalet gånger `k` förekommer i `xs`. Genom att omvandla alla värden `p._2` till storleken `p._2.size` får vi en frekvenstabell.

```
1 scala> freq(kasta(1000))
2 val res0: Map[Int, Int] =
3   Map(5 -> 163, 1 -> 174, 6 -> 161, 2 -> 169, 3 -> 167, 4 -> 166)
4
5 scala> freq(kasta(1000)).toVector.sortBy(_._1).foreach(println)
6 (1,183)
7 (2,167)
8 (3,169)
9 (4,179)
10 (5,154)
11 (6,148)
```

#### Lösn. uppg. 14. –

## L.10 Lösning inheritance

### L.10.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** Para ihop begrepp med beskrivning.

bastyp	1	↔	N	den mest generella typen i en arvshierarki
supertyp	2	↔	O	en typ som är mer generell
subtyp	3	↔	D	en typ som är mer specifik
körtidstyp	4	↔	J	kan vara mer specifik än den statiska typen
dynamisk bindning	5	↔	L	körtidstypen avgör vilken metod som körs
polymorfism	6	↔	B	kan ha många former, t.ex. en av flera subtyper
trait	7	↔	I	är abstrakt, kan mixas in, kan ha parametrar
inmixning	8	↔	H	tillföra egenskaper med <b>with</b> och en trait
överskuggad medlem	9	↔	M	medlem i subtyp ersätter medlem i supertyp
anonym klass	10	↔	C	klass utan namn, utvidgad med extra implementation
skyddad medlem	11	↔	K	är endast synlig i subtyper
abstrakt medlem	12	↔	F	saknar implementation
abstrakt klass	13	↔	E	kan ha parametrar, kan ej instansieras, kan ej mixas in
förseglad typ	14	↔	P	subtypning utanför denna kodfil är förhindrad
referenstyp	15	↔	A	har supertypen AnyRef, allokeras i heapen via referens
värdetyp	16	↔	G	har supertypen AnyVal, lagras direkt på stacken

**Lösn. uppg. 2.** Gemensam bastyp.

a) `Vector[Object]`. Typen `Object` i JVM är motsvarar typen `AnyRef` som är bastyp för alla referenstyper.

b) Felmeddelande:

```
scala> grönsaker.map(_.vikt).sum
-- Error:
1 |grönsaker.map(_.vikt).sum
  |                ^^^^^^
  |                value vikt is not a member of Object - did you mean wait?
-- Error:
1 |grönsaker.map(_.vikt).sum
  |                ^
  |ambiguous implicit arguments: both object DoubleIsFractional in object Numeric and object ShortIsFractional in object Numeric
```

Det första felmeddelandet beror på att vektorns element är av typen `Object` och medlemmen `vikt` är inte definierat för denna typ. Det andra felmeddelandet är ett följdfejl som beror på att en sekvens med element av typen `Object` inte kan summeras eftersom kompilatorn inte kan härleda att elementtypen är numerisk.

c) Attributet `vikt` initialiseras vid konstruktion av `Gurka` resp. `Tomat`. Värdet ges av resp. klassparameter.

d) `Vector[Grönsak]`.

e) Ja. Eftersom den statiska typen för elementen i sekvensen är Grönsak (den dynamiska typen kan vara godtycklig subtyp av Grönsak) och alla instanser av denna typ garanterat har attributet vikt som är av typen Int så kan kompilatorn vid *kompileringstid* dra slutsatsen att summeringen är giltig och därmed kan kompilatorn kompilera koden till körbar maskinkod.

f)

```
scala> new Grönsak
-- Error:
1 |new Grönsak
  | ^^^^^^^
  | Grönsak is a trait; it cannot be instantiated
```

g)

```
scala> val anonymGrönsak = new Grönsak { val vikt = 42 }
val anonymGrönsak: Grönsak = anon$1@1edde8b6
scala> anonymGrönsak.toString
val res0: String = anon$1@1edde8b6
```

Typen är Grönsak och blir här en s.k. *anonym klass*, eftersom vi inte har använt en namngiven klass med **extends**, utan bara "hängt på" en klasskropp inom klammerparenteser direkt vid konstruktion. När du skapar anonyma klasser måste du använda nyckelordet **new**.

Kompilatorn hittar på ett unikt klassnamn, här anon\$1, för att hålla reda på den anonyma klassen under kompilering till maskinkod. Strängrepresentationen innehåller ett hexadecimalt heltal som är unikt för instansen, här 1edde8b6.

h)

```
scala> new Grönsak { }
-- Error:
1 |new Grönsak { }
  | ^
  | object creation impossible, since val vikt: Int in trait Grönsak is not defined
```

### Lösn. uppg. 3. Polymorfism vid arv, s.k. subtypspolymorfism.

a)

```
def skapaDjur(): Djur =
  if math.random() > 0.5 then Ko() else Gris()
```

b)

```
class Häst extends Djur:
  def väsnas = println("Gnääääägg")

def skapaDjur(): Djur =
  math.random() match
    case r if r < 0.33 => Ko()
    case r if r < 0.67 => Gris()
    case _             => Häst()
```

### Lösn. uppg. 4. Olika typer av heltalspar till laborationen snake0.

a)

```
trait Pair[T]:
  def x: T
  def y: T
  def tuple: (T, T) = (x, y)
```

b)

```
case class Dim(x: Int, y: Int) extends Pair[Int]
object Dim:
  def apply(dim: (Int, Int)): Dim = Dim(dim._1, dim._2)
```

c)

```
case class Pos private (x: Int, y: Int, dim: Dim) extends Pair[Int]:
  def +(p: Pair[Int]): Pos = Pos(x + p.x, y + p.y, dim)
  def -(p: Pair[Int]): Pos = Pos(x - p.x, y - p.y, dim)

object Pos:
  def apply(x: Int, y: Int, dim: Dim): Pos =
    import java.lang.Math.floorMod as mod
    new Pos(mod(x, dim.x), mod(y, dim.y), dim) //OBS: new nödvändig här!

  def random(dim: Dim): Pos =
    import scala.util.Random.nextInt as rni
    Pos(rni(dim.x), rni(dim.y), dim)
```

d) Om du glömmer skriva **new** explicit i kompanjonsobjektets `apply`-metod så blir det ett rekursivt anrop som resulterar i en oändlig loop vid körtid. Med **new** så är det garanterat den privata primärkonstruktorn för `Pos` som anropas.

I `Dim.apply` så skiljer sig parametertyperna åt mellan fabriksmetoden och primärkonstruktorn och kompilatorn väljer då primärkonstruktorn eftersom den passar med de givna två separata heltalen och inte med en 2-tupel.

e)

```
enum Dir(val x: Int, val y: Int) extends Pair[Int]:
  case North extends Dir( 0, -1)
  case South extends Dir( 0,  1)
  case East  extends Dir( 1,  0)
  case West  extends Dir(-1,  0)
export Dir.* // gör så att North etc blir synliga i paketet snake
```

### Lösn. uppg. 5. Supertyp med parameter.

a)

```
val person = new Person("Person1")
val akademiker = new Akademiker("Person2", "LTH")
val student = new Student("Person3", "LTH", "D")
val forskare = new Forskare("Person4", "LTH", "Doktorand")
```

b)

```
val vec = Vector(person, akademiker, student, forskare)
for(i <- vec){ print(i.toString + i.namn) }
```

c) Felmeddelande vid instansiering av **abstract class** Akademiker:

Akademiker is abstract; it cannot be instantiated

Det går *inte* lika bra med en **trait** i det speciella fallet Akademiker, eftersom en trait inte får skicka vidare parametrar till en supertyp. Felmeddelande:

trait Akademiker may not call constructor of trait Person

```
trait Person(val namn: String)

abstract class Akademiker(
  namn: String,
  val universitet: String) extends Person(namn)

class Student(
  namn: String,
  universitet: String,
  program: String) extends Akademiker(namn, universitet)

class Forskare(
  namn: String,
  universitet: String,
  titel: String) extends Akademiker(namn, universitet)
```

d)

```
scala>
| trait Person(val namn: String)
|
| abstract class Akademiker(
|   namn: String,
|   val universitet: String) extends Person(namn)
|
| case class Student(
|   namn: String,
|   universitet: String,
|   program: String) extends Akademiker(namn, universitet)
|
| case class Forskare(
|   namn: String,
|   universitet: String,
|   titel: String) extends Akademiker(namn, universitet)
-- Error:
8 |   namn: String,
  |   ^
  | error overriding value namn in trait Person of type String;
  |   value namn of type String needs `override` modifier
-- Error:
9 |   universitet: String,
  |   ^
  | error overriding value universitet in class Akademiker of type String;
  |   value universitet of type String needs `override` modifier
-- Error:
13 |   namn: String,
   |   ^
   | error overriding value namn in trait Person of type String;
   |   value namn of type String needs `override` modifier
-- Error:
14 |   universitet: String,
   |   ^
```

```
| error overriding value universitet in class Akademiker of type String;
| value universitet of type String needs `override` modifier
```

```
trait Person(val namn: String)

abstract class Akademiker(
  namn: String,
  val universitet: String) extends Person(namn)

case class Student(
  override val namn: String,
  override val universitet: String,
  program: String) extends Akademiker(namn, universitet)

case class Forskare(
  override val namn: String,
  override val universitet: String,
  titel: String) extends Akademiker(namn, universitet)
```

```
scala> val ps = Vector(Student("Kim", "Lund", "D"), Forskare("Herz", "Lund", "Dr"))
val ps: Vector[Akademiker] = Vector(Student(Kim,Lund,D), Forskare(Herz,Lund,Dr))
scala> ps :+ new Person("Abstrakt") {}
val res0: Vector[Person] =
  Vector(Student(Kim,Lund,D), Forskare(Herz,Lund,Dr), anon1@1941bbf3)
```

e)

```
trait Person:
  val namn: String
  val nbr: Long

trait Akademiker extends Person:
  val universitet: String

case class Student(
  namn: String,
  nbr: Long,
  universitet: String,
  program: String) extends Akademiker

case class Forskare(
  namn: String,
  nbr: Long,
  universitet: String,
  titel: String) extends Akademiker

case class IckeAkademiker(
  namn: String,
  nbr: Long) extends Person
```



## L.10.2 Extrauppgifter; träna mer

### Lösn. uppg. 6. Bastypen Shape och subtyperna Rectangle och Circle.

a)

```
val c1 = Circle(Point(1, 1), 42)
val r1 = Rectangle(Point(3, 3), 20, 30)
c1.move(2, 3)
r1.move(3, 2)
```

b)

```
case class Point(x: Double, y: Double):
  def move(dx: Double, dy: Double): Point = Point(x + dx, y + dy)
  def moveTo(x: Double, y: Double): Point = Point(x, y)

trait Shape:
  def pos: Point
  def move(dx: Double, dy: Double): Shape
  def moveTo(x: Double, y: Double): Shape

case class Rectangle(pos: Point, width: Double, height: Double) extends Shape:
  def move(dx: Double, dy: Double): Shape = copy(pos = pos.move(dx, dy))
  def moveTo(x: Double, y: Double): Shape = copy(pos.moveTo(x, y))

case class Circle(pos: Point, radius: Double) extends Shape:
  def move(dx: Double, dy: Double): Shape = copy(pos = pos.move(dx, dy))
  def moveTo(x: Double, y: Double): Shape = copy(pos.moveTo(x, y))
```

c) **def** distanceTo(that: Point): Double = math.hypot(that.x - x, that.y - y)

d) **def** distanceTo(that: Shape): Double = pos.distanceTo(that.pos).

e)

```
case class Point(x: Double, y: Double):
  def move(dx: Double, dy: Double): Point = Point(x + dx, y + dy)
  def moveTo(x: Double, y: Double): Point = Point(x, y)
  infix def distanceTo(that: Point): Double = math.hypot(that.x - x, that.y - y)

trait Shape:
  def pos: Point
  def move(dx: Double, dy: Double): Shape
  def moveTo(x: Double, y: Double): Shape
  infix def distanceTo(that: Shape): Double = pos.distanceTo(that.pos)

case class Rectangle(pos: Point, width: Double, height: Double) extends Shape:
  def move(dx: Double, dy: Double): Shape = copy(pos = pos.move(dx, dy))
  def moveTo(x: Double, y: Double): Shape = copy(pos.moveTo(x, y))

case class Circle(pos: Point, radius: Double) extends Shape:
  def move(dx: Double, dy: Double): Shape = copy(pos = pos.move(dx, dy))
  def moveTo(x: Double, y: Double): Shape = copy(pos.moveTo(x, y))
```

## L.10.3 Fördjupningsuppgifter; utmaningar

### Lösn. uppg. 7. Inmixning.

a) Det finns många olika sätt, några exempellösningar:

```
val antalFlygånkor: Int =
  fyle.count(f => f.isInstanceOf[Ånka] && f.ärFlygkunnig)
```

```
val antalFlygånkor: Int =
  fyle.filter(f => f.isInstanceOf[Ånka] && f.ärFlygkunnig).size
```

```
val antalFlygånkor: Int =
  fyle.collect{case f: Ånka if f.ärFlygkunnig}.size
```

```
val antalFlygånkor: Int = fyle.map(_ match
  case f: Ånka if f.ärFlygkunnig => 1
  case _ => 0
).sum
```

b)

```
val antalKrax: Int = fyle.filter(f => !f.ärSimkunnig).size * 2
val antalKvack: Int = fyle.filter(f => f.ärSimkunnig).size * 4
```

### Lösn. uppg. 8. Finala klasser.

- a) Sätt **final** framför **class** i klasserna.
- b) error: illegal inheritance from final class Kråga.

### Lösn. uppg. 9. Accessregler vid arv och nyckelordet **protected**.

a)

```
1 2 | def avslöja = minHemlis
2   |           ^^^^^^^^^
3   |           Not found: minHemlis
```

b)

```
1 scala> class Sub extends Super:
2     def kryptisk = vårHemlis * math.Pi
3 scala> (new Sub).vårHemlis
4 -- Error:
5 1 |(new Sub).vårHemlis
6   |^^^^^^^^^^^^^^^^^^
7   |value vårHemlis in trait Super cannot be accessed as a member of Sub.
8   | Access to protected value vårHemlis not permitted because enclosing object
9   | is not a subclass of trait Super where target is defined
```

c) Ja.

### Lösn. uppg. 10. Användning av **protected**.

a) I Fyle:

```
protected var räknaLäte: Int = 0
def väsnas: Unit = { print(läte * 2); räknaLäte += 2 }
```

I Ånka: **override def** väsnas = { print(läte \* 4); räknaLäte += 4 }

b) **def** antalLäten: Int = räknaLäte

c) Om en klass som representerar en fågel som skulle ge ifrån sig fler/färre läten än en vanlig Fyle, behöver väsnas ändras. Denna metod behöver tillgång till räknaLäte, vilken inte får vara **private**.

d) Räknar-variabeln ska inte kunna påverkas i någon annan del av programmet.

### Lösn. uppg. 11. Inmixning av egenskaper.

```
trait Fyle:
  val läte: String
  def väsnas: Unit = { print(läte * 2); räknaLäte += 2 }
  protected var räknaLäte: Int = 0
  val ärSimkunnig: Boolean
  val ärFlygkunnig: Boolean
  val ärStor : Boolean
  def antalLäten: Int = räknaLäte

trait KanSimma { val ärSimkunnig = true }
trait KanInteSimma { val ärSimkunnig = false }
trait KanFlyga { val ärFlygkunnig = true }
trait KanKanskeFlyga { val ärFlygkunnig = math.random() < 0.8 }
trait KanKanskeSimma { val ärSimkunnig = math.random() < 0.2 }
trait ÄrStor { val ärStor = true }
trait ÄrLiten { val ärStor = false }

final class Kråga extends Fyle, KanFlyga, KanInteSimma, ÄrStor:
  val läte = "krax"

final class Ånka extends Fyle, KanSimma, KanKanskeFlyga, ÄrStor:
  val läte = "kvack"
  override def väsnas = { print(läte * 4); räknaLäte += 4 }

final class Pjodd extends Fyle, KanFlyga, KanKanskeSimma, ÄrLiten:
  val läte = "kvitter"
  override def väsnas = { print(läte * 8); räknaLäte += 8 }
```

### I REPL:

```
1 val fyle = Vector.fill(42)(
2   if math.random() < 0.33 then Kråga()
3   else if math.random() < 0.5 then Ånka()
4   else Pjodd()
5 )
6 fyle.filter(f => f.isInstanceOf[Kråga]).size * 2
7 fyle.filter(f => f.isInstanceOf[Ånka]).size * 4
8 fyle.filter(f => f.isInstanceOf[Pjodd]).size * 8
```

## L.11 Lösning context

### L.11.1 Grunduppgifter; förberedelse inför laboration

#### Lösn. uppg. 1. Kontextparameter.

a)

```
scala> given Int = 0
val res0: Int = 83

scala> f(using 41)
val res1: Int = 42

scala> g(41)(using 42)
val res2: Int = 83
```

Om man glömmer **using** vid explicit kontextargument blir det kompileringsfel. Kompilatorn blir "förvirrad" och tror att du försöker ge ett "vanligt" argument till en (i detta fallet) icke-existerande "vanlig" parameterlista.

```
scala> f(41)
-- [E050] Type Error: -----
1 |f(41)
  |^
  |method f does not take more parameters
  |
  | longer explanation available when compiling with `-explain`
1 error found

scala> :setting -explain

scala> f(41)
-- [E050] Type Error: -----
1 |f(41)
  |^
  |method f does not take more parameters
  |-----
  | Explanation (enabled by `-explain`)
  |-----
  | You have specified more parameter lists than defined in the
  | method definition(s).
  |-----

scala> g(41)(42)
-- [E050] Type Error: -----
1 |g(41)(42)
  |^^^^
  |method g does not take more parameters
  |-----
  | Explanation (enabled by `-explain`)
  |-----
  | You have specified more parameter lists than defined in the
  | method definition(s).
  |-----
```

Det är inte vanligt att ange **using**-parametrar explicit; det vanligaste är att låta kompilatorn framkalla ett givet värde.

b) Det givna värdet 0 binds till motsvarande kontextparameter, som ska vara deklarerad i en egen parameterlista som börjar med **using**.

```
scala> f
```

```
val res3: Int = 1

scala> g(42)
val res4: Int = 42
```

c) Nej, det blir kompileringsfel om man försöker blanda vanliga parametrar och kontextparametrar i en och samma parameterlista:

```
scala> def h(i: Int, using j: Int) = i + j
-- [E040] Syntax Error: -----
1 |def h(i: Int, using j: Int) = i + j
  |               ^
  |               ':' expected, but identifier found
```

Det är ett medvetet val att kräva separata parameterlistor, så att det inte ska uppstå förvirring om huruvida en vanlig parameter eller kontextparameter avses.

**Lösn. uppg. 2.** *Flera olika givna värden i lokal kontext.*

- a) 2
- b) 43
- c) När kompilatorn försöker framkalla ett givet värde att automatiskt använda som argument till **using**-parametern `dx`, så letar den i den kontext som är närmast anropet först. Om det finns ett givet värdet i kompanjonsobjektet för parametertypen så tar kompilatorn detta i sista hand, om inget annat givet värde hittas närmare anropet.

**Lösn. uppg. 3.** *Lösning på konfigurationsproblemet med hjälp av givna värden.*

Nedan visas test av de tre olika lösningarna som givits i uppg. a)

Efter varje test diskuteras tillhörande för- och nackdelar, som efterfrågas i uppg. b)

```
def testGlobalVar(useDefault: Boolean = true) =
  import GlobalVar.*
  if useDefault then println(greetMsg) else
    GreetConfig.config = GreetConfig("Godmorgon", "världen")
    println(greetMsg)
```

Eftersom `config` här är en förändringsbar variabel, så kan en ändring på ett ställe påverka helt andra delar av programmet, vilket ibland kan vara en fördel, men ofta en nackdelen eftersom det kan vara svårt att förstå vad som händer bara genom att läsa en enskild del av programmet – en förändring av `config` kan ju ske varsomhelst. Det är lätt att glömma ändra till baka till default-värdet, om det är det som förväntas.

```
1 scala> testGlobalVar(); testGlobalVar(false); testGlobalVar()
2 Hello World!
3 Godmorgon världen!
4 Godmorgon världen!
```

```
1 def testDefaultArgs(useDefault: Boolean = true) =
2   import DefaultArgs.*
3   if useDefault then println(greetMsg()) else
4     println(greetMsg(GreetConfig("Godmorgon", "världen")))
```

Här sker ingen tillståndsförändring och default-användning är enkel, men det går inte enkelt att göra avsteg från default som gäller i en lokal kontext; vid *varje* enskilt anrop behöver du explicit ange alla de argument som inte ska vara default, så som visas på rad 4 ovan. Ändring av default har bara lokal påverkan. Om alla argument ska följa default, så gäller det att inte glömma anropa med tomt parentespar: `greetMsg()`. (Vad händer annars?)

```
1 scala> testDefaultArgs(); testDefaultArgs(false); testDefaultArgs()
2 Hello World!
3 Godmorgon världen!
4 Hello World!
```

Med kontextparametrar är flexibiliteten större; **using**-parametrar låter användaren själv styra vad som gäller i olika sammanhang och själva anropet blir enkelt oavsett om det är default-värdet eller andra, i den lokala kontexten, givna värden som önskas. Ändring av default har bara lokal påverkan, men den har påverkan på godtyckligt många anrop i den lokala kontexten – argument som skiljer sig kan alltså vara givna en gång utan att behöva upprepas vid varje anrop. Vid anrop där man vill låta kompilatorn framkallar givna värden för kontextparametern ska inga parenteser användas, och anropen bli därmed korta och enkla.

```
def testGivenVal(using g: GivenVal.GreetConfig) = println(g.greetMsg)
```

```
1 scala> testGivenVal
2 Hello World
3
4 scala> def localContext =
5     import GivenVal.*
6     given GreetConfig = GreetConfig("Godmorgon","världen")
7     testGivenVal
8
9 scala> localContext
10 Godmorgon världen
11
12 scala> testGivenVal
13 Hello World
```

c) Kompilatorn framkallar ett givet värde i den lokala kontexten:

```
1 scala> summon[GivenVal.GreetConfig]
2 val res0: GivenVal.GreetConfig = GreetConfig>Hello,World)
```

Kompilatorn följer denna prioritetsordning i sökandet efter ett unikt givet värde:

1. **Explicita** argument till kontextparametrar märkta med **using**
2. **given** och **import given** ... i aktuell namnrymd (eng. *current scope*)
3. **given**-värden i **kompanjonsobjekt** för den använda typen.

Om flera givna värden kan framkallas för typer som ingår i en gemensam arvshierarki så väljer kompilatorn det givna värdet som är av den *mest specifika* typen.

d) Det blir kompileringsfel om kompilatorn inte hittar ett givet värde för den typ som avses.

```
1 scala> summon[Long]
2 -- Error: -----
3 1 |summon[Long]
4   |          ^
5   |          no given instance of type Long was found for parameter x of
```

```

6         method summon in object Predef
7 1 error found

```

e) Ja! Det får *inte* vara tvetydigt vilket givet värde som ska framkallas:

```

1 scala> def tvetydigt =
2     |   given a: Int = 42
3     |   given b: Int = 43
4     |   summon[Int]
5     |
6 -- Error: -----
7 4 |   summon[Int]
8   |           ^
9   |ambiguous given instances: both given instance b and given instance a
10  |match type Int of parameter x of method summon in object Predef
11 1 error found

```

Läs mer om kontextuella abstraktioner här:

<https://docs.scala-lang.org/scala3/reference/contextual/>

## L.11.2 Extrauppgifter; träna mer

**Lösn. uppg. 4.** *Kontextparameter och givet värde.*

a)

```

1 scala> add(1)
2 -- Error: -----
3 1 |add(1)
4   |    ^
5   |    no given instance of type Int was found for parameter y of method add
6 1 error found

```

b) Nu finns ett givet värde som kompilatorn automatiskt kan fylla i på platsen vid anropet.

c) **def** sub(x: Int)(**using** y: Int) = x - y

## L.11.3 Fördjupningsuppgifter; utmaningar

**Lösn. uppg. 5.** *Varians och typgränser.*

a) Gör lådan flexibel i sin typparameter med ett + före typparametern enligt nedan.

```
case class Box[+A](x: A)
```

Kompilatorn tillämpar reglerna för kovarians eftersom typparametern har ett plustecken framför sig: Box[Cat] är en suptyp till Box[Any] om Cat är en subtyp till Any, vilket den ju är eftersom alla typer är subtyp till Any.

b) Förklaringen till beteendet har med olika varians att göra:

- Samlingen Vector är kovariant och därmed flexibel i sin typparameter (liksom andra oföränderliga sekvenser i Scalas standardbibliotek). Kompilatorn betraktar därmed Vector[Cat] som en subtyp till Vector[Pet] eftersom Cat är en subtyp till Pet. På platser i koden där en Vector[Pet] krävs så anses Vector[Cat] överensstämma med (eng. *conforms to*) Vector[Pet] och får därmed duga på dessa platser.

- En mängd har en apply-metod från elemnttypen till Boolean som ger innehållstest. Av det skälet har man låtit `Set[T]` ärva `Function1[T, Boolean]` som är deklarerad kontravariant i `T`, så att en mängd kan användas där en `T => Boolean` förväntas. Även om det skulle vara praktiskt om `Set[T]` vore kovariant i `T`, i likhet med `Vector`, `List`, `Seq` etc, så kan inte `T` vara både kovariant och kontravariant på en och samma gång. Man har därför valt att göra `Set` invariant och därmed är mängder ej flexibla i sin typparameter. `Set[Cat]` är alltså *inte* en subtyp till `Set[Pet]` *även* om `Cat` är en subtyp till `Pet`, vilket ger kompileringsfel i uppgiftens exempel. Se även <https://stackoverflow.com/questions/676615/why-is-scalas-immutable-set-not-covariant-in-its-type>
- Med `:settings -explain` ger kompilatorn en längre utskrift som förklarar den bevisföring som skedde under kompileringens typkontroll.

c) Det blir kompileringsfel då metoden `isHealthy` ej existerar för godtycklig typ.

d) Lägg till en övre gräns som garanterar att metoden `isHealthy` finns för alla typer som kan bindas till typparametern `A`:

```
class Vet[-A <: Pet]:
  def treat(x: A): Unit = x.isHealthy = true
```

Kompilatorn garanterar alltså att typparametern `A` är ”mindre än eller lika med” `Pet`.

e) Veterinären `Vet` är flexibel i sin typparameter och minustecknet anger kontravarians och därmed att `Vet[Pet]` är en subtyp till `Vet[Cat]` då `Cat` är en subtyp till `Pet`. Detta kan demonstreras med nedan exempel:

```
1 scala> val pinkPanther = Cat()
2 val pinkPanther: Cat = Cat@33e7ece5
3
4 scala> val somePet: Pet = Cat()
5 val somePet: Pet = Cat@57f1cb96
6
7 scala> val catVet = Vet[Cat]()
8 val catVet: Vet[Cat] = Vet@1060e784
9
10 scala> pinkPanther.isHealthy = false
11
12 scala> catVet.treat(pinkPanther)
13
14 scala> pinkPanther.isHealthy
15 val res2: Boolean = true
16
17 scala> somePet.isHealthy = false
18
19 scala> catVet.treat(somePet)
20 -- [E007] Type Mismatch Error: -----
21 1 |catVet.treat(somePet)
22   |             ^^^^^^^
23   |             Found:    (somePet : Pet)
24   |             Required: Cat
25
26 scala> val powerVet = Vet[Pet]()
27 val powerVet: Vet[Pet] = Vet@2eb90ae9
28
29 scala> pinkPanther.isHealthy = false
30
31 scala> powerVet.treat(pinkPanther)
32
33 scala> pinkPanther.isHealthy
```



```
34 val res3: Boolean = true
35
36 scala> val pluto = Dog()
37 val pluto: Dog = Dog@6f27db5d
38
39 scala> pluto.isHealthy = false
40
41 scala> powerVet.treat(pluto)
42
43 scala> pluto.isHealthy
44 val res4: Boolean = true
```

**Lösn. uppg. 6.** *Typklasser och kontextparametrar.*

- a) **TODO!!!**
- b) **TODO!!!**
- c) **TODO!!!**

**Lösn. uppg. 7.** *Användning av given ordning.—* **TODO!!!**

**Lösn. uppg. 8.** *Skapa egen implicit ordning med Ordering.*

- a) **TODO!!!**
- b) **TODO!!!**
- c) **TODO!!!**
- d) **TODO!!!**

**Lösn. uppg. 9.** *Specialanpassad ordning genom att ärva från Ordered*

a)

```
case class Team(name: String, rank: Int) extends Ordered[Team]:
  override def compare(that: Team): Int = -rank.compare(that.rank)
```

b)

```
scala> Team("fnatic",1499) < Team("gurka", 2)
val res1: Boolean = true
```

- c) Ad hoc polymorfism är mer flexibel. **TODO!!!** mer diskussion om likheter och skillnader här...

## L.12 Lösning extra

### L.12.1 Uppgifter om sökning och sortering

#### Lösn. uppg. 1. Tidmätning.

a) Exekvera koden och du bör finna att det tar längre tid att hitta värdet 1 i vårt Set *s* än i vektorn *v*.

b)

En vektor har en sekventiell ordning som *find* kan använda, medan Set är internt ordnad på ett annat sätt för att innehållskontroll ska gå extra snabbt. Anledningen att det tar tid för *find* på Set är att det först måste skapas en iterator innan vår mängd kan gås igenom från början till slut. Metoden *contains* på Set däremot är rasande snabb beroende på den interna strukturen hos objekt av typen Set (som är smart designad med s.k. hash-koder, där det går lika snabbt att hitta ett element oavsett vart det befinner sig).

#### Lösn. uppg. 2. Sökning med inbyggda sökmetoder.

a) Förslag på test av *indexOfSlice*:

```
scala> List(1,2,3,35,1,23).indexOfSlice(List(35,1,23))
res73: Int = 3
scala> List(1,2,3,35,1,23).indexOfSlice(List(35,1,3))
res74: Int = -1
```

b) Förslag på test av *lastIndexOfSlice*:

```
Vector(1,2,3,4,1,2).lastIndexOfSlice(Vector(1,2))
res2: Int = 4
Vector("apa", "banan", "majs", "banan").lastIndexOfSlice(Vector("banan"))
res3: Int = 3
Vector("apa", "banan", "majs", "banan").lastIndexOfSlice(Vector("banand"))
res4: Int = -1
```

c) Observera att metoden *search* antar att samlingen är sorterad i stigande ordning. När vi inverterar ordningen kan *search* oftast inte hitta det vi letar efter, eftersom den kommer leta i fel halva av samlingen.

```
scala> val udda = (1 to 1000000 by 2).toVector
scala> import scala.collection.Searching._
scala> udda.search(udda.last)
res18: collection.Searching.SearchResult = Found(499999)
//Search hittar det sista elementet på plats 499999 i samlingen.

scala> udda.search(udda.last + 1)
res19: collection.Searching.SearchResult = InsertionPoint(500000)
//Search kan inte hitta udda.last + 1 eftersom det inte existerar i samlingen
//och returnerar således ett objekt av typen InsertionPoint med värdet 500000.
//Vårt element udda.last + 1 hade alltså legat på plats 500000 om det funnits.

scala> udda.reverse.search(udda(0))
res20: collection.Searching.SearchResult = InsertionPoint(0)
//Som förklarat innan så förutsätter search att listan är sorterad i stigande
//ordning, så den kan inte hitta elementet udda(0) = 1 när listan är inverterad.

scala> def lin(x: Int, xs: Seq[Int]) = xs.indexOf(x)
scala> def bin(x: Int, xs: Seq[Int]) = xs.search(x) match
  case Found(i) => i
  case InsertionPoint(i) => -i
```

```
//Definierar en metod bin som använder sig av metoden search på en sekvens.
//Den ser sedan till med hjälp av "pattern matching" att bara returnera positionen
//i, och inte ett objekt av typen Found eller InsertionPoint.
```

```
scala> timed{ lin(udda.last, udda) }
time: 42.294821 ms
res22: (Int, Long) = (499999,42294821)
//För att hitta udda.last = 499999 med linjärsökning tog det ca 42ms.

scala> timed{ bin(udda.last, udda) }
time: 0.147314 ms
res23: (Int, Long) = (499999,147314)
//Binärsökning för att hitta värdet 499999 tog extremt mycket kortare tid.
//Detta för att vid varje steg i binärsökningen halveras mängden tal som
//sökningen måste kolla i. Detta är dock ett extremfall eftersom vi söker
//talet längst bak i listan. Om vi istället gjort en linjärsökning efter
//det första talet 1, hade detta gått minst lika snabbt som binärsökning.
```

d) Det behövs  $\log_2(n)$  jämförelser. Detta eftersom att vi hela tiden halverar antalet element i listan vi behöver söka igenom. Så efter första jämförelsen har vi  $\frac{n}{2}$  element kvar. Efter andra jämförelsen har vi  $\frac{n}{2*2}$  element kvar etc. När vi bara har ett element kvar har vi hittat det vi söker efter, och har då gjort  $b$  antal jämförelser. Ekvationen ser då ut på följande vis:

$$\frac{n}{2^b} = 1$$

Enligt lagarna för logaritmer kan vi nu komma fram till vad  $b$  är:

$$\log_2(n) = b$$

### Lösn. uppg. 3. Sök bland LTH:s kurser med linjärsökning.

- a) Första raden innehåller kolumnnamnen Kurskod KursSve KursEng Hskpoang Niva. Därefter kommer en rad för varje kurs med kursdata enligt kolumnnamnen.
- b) Koden laddar ner data och skapar en vektor med instanser av case-klassen Course med hjälp av metoden fromLine. Eftersom variabeln lth är deklarerad som **lazy** kommer inte download() bli anropad förrän första gången som variabeln lth refereras. Antalet kurser ges av:

```
scala> val n = courses.lth.length
n: Int = 1104
```

c)

```
1 scala> def isCS(s: String) = s.startsWith("EDA") || s.startsWith("ETS")
2 scala> val x = courses.lth.find(c => isCS(c.code) && c.level == "G2")
3 x: Option[courses.Course] = Some(Course(EDAF05,Algoritmer, datastrukturer och
4 komplexitet,Algorithms, Data Structures and Complexity,5.0,G2))
```

d)

```
def linearSearch[T](xs: Seq[T])(p: T => Boolean): Int =
  var i = 0
  while(i < xs.length && !p(xs(i))) i += 1
  if (i < xs.length) i else -1
```

e)

```
def rndCode: String =
  //randomizes from 0 to n (inclusive)
  def rnd(n: Int) = (math.random() * (n + 1)).toInt

  def letter = (rnd('Z' - 'A') + 'A').toChar
  def dig = ('0' + rnd(9)).toChar
  Seq(letter, letter, letter, letter, dig, dig).mkString
```

f)

```
val xs = Vector.fill(500000)(rndCode)
val (ixs, elapsedLin) =
  timed { xs.map(x => linearSearch(courses.lth)(_.code == x)) }
val found = ixs.filterNot(_ == -1).size
```

g)

```
def linearSearch[T](xs: Seq[T])(p: T => Boolean): Int = xs.indexOfWhere(p)
```

**Lösn. uppg. 4.** Sök bland LTH:s kurser med binärsökning.

a) —

b)

```
def binarySearch(xs: Seq[String], key: String): Int =
  var (low, high) = (0, xs.size - 1)
  var found = false
  var mid = -1

  while (low <= high && !found) do
    mid = (low + high) / 2
    if xs(mid) == key then found = true
    else if xs(mid) < key then low = mid + 1
    else high = mid - 1
  end while
  if found then mid else -(low + 1)
```

c) Med en i7-3770K @ 3.50Hz tog sökningarna följande tid:

- Binärsökning: time: 142.6 ms
- Linjärsökning: time: 3316.5 ms

Med en i7-8700T @ 2.40GHz tog sökningarna följande tid:

- Binärsökning: time: 81.5 ms
- Linjärsökning: time: 5138.6 ms

d) Binärsökningen var ca 23 gånger snabbare på en i7-3770K @ 3.50Hz och ca 63 gånger snabbare på en i7-8700T CPU @ 2.40GHz.

**Lösn. uppg. 5.** *Insättningssortering.*

a) —

b)

```
def insertionSort(xs: Seq[Int]): Seq[Int] =  
  val result = scala.collection.mutable.ArrayBuffer.empty[Int]  
  for e <- xs do  
    var pos = 0  
    while pos < result.size && result(pos) < e do pos += 1  
    result.insert(pos,e)  
  end for  
  result.toVector
```

**Lösn. uppg. 6.** *Sortering på plats.*

```
def selectionSortInPlace(xs: Array[String]): Unit =  
  def indexOfMin(startFrom: Int): Int =  
    var minPos = startFrom  
    var i = startFrom + 1  
    while (i < xs.size) do  
      if (xs(i) < xs(minPos)) minPos = i  
      i += 1  
    end while  
    minPos  
  end indexOfMin  
  
  def swapIndex(i1: Int, i2: Int): Unit =  
    val temp = xs(i1)  
    xs(i1) = xs(i2)  
    xs(i2) = temp  
  end swapIndex  
  
  for i <- 0 to xs.size - 1 do swapIndex(i, indexOfMin(i))  
end selectionSortInPlace
```

**Lösn. uppg. 7.** *Undersök om en sekvens är sorterad.*

a) Det tar i värsta fall  $O(n * \log(n))$  för timsort att sortera listan med  $n$  element. Sedan krävs  $n$  stycken jämförelser mellan den sorterade och osorterade listan. Det totala antalet jämförelser i värstafallet uppgår därför till max  $n + n * \log(n)$ . För  $10^6$  element blir det ca  $10^7$  jämförelser.

```
scala> val n = 1E6
val n: Double = 1000000.0

scala> def worstCase(n: Double) = n + n * math.log(n)
def worstCase(n: Double): Double

scala> println(s"i värsta fall med n=$n så blir det ${worstCase(n)} jämförelser")
i värsta fall med n=1000000.0 så blir det 1.4815510557964273E7 jämförelser
```

b) En mer effektiv version av `isSorted` som avbryter sökningen när ett osorterat element upptäcks:

```
def isSorted(xs: Vector[Int]): Boolean =
  if xs.length > 1 then
    var i = 0
    var result = true
    while i < xs.length-1 && result do
      if xs(i) > xs(i+1) then result = false
      i += 1
    end while
    result
  else true
end isSorted
```

c) I värsta fall behöver man göra  $n - 1$  parvisa jämförelser, om alla ligger i sorterad ordning utom den sista.

d) 2-tupeln är av typen `(Int, Int)`.

```
def isSorted(xs: Vector[Int]): Boolean =
  xs.zip(xs.tail).forall(x => x._1 <= x._2)
```

**Lösn. uppg. 8.** *Insättningssortering på plats.*

```
def insertionSort(xs: Array[Int]): Unit =
  for elem <- 1 until xs.length if xs.length > 0 do
    var pos = elem
    while pos > 0 && xs(pos) < xs(pos - 1) do
      val temp = xs(pos - 1)
      xs(pos - 1) = xs(pos)
      xs(pos) = temp
      pos -= 1
    end while
  end for
end insertionSort
```

**Lösn. uppg. 9.** *Sortering till ny sekvens med urvalssortering.*

```
def selectionSort(xs: Seq[String]): Seq[String] =
  def indexOfMin(xs: Seq[String]): Int = xs.indexOf(xs.min)
  val unsorted = xs.toBuffer
  val result = scala.collection.mutable.ArrayBuffer.empty[String]
  while !unsorted.isEmpty do
    val minPos = indexOfMin(unsorted)
    val elem = unsorted.remove(minPos)
    result.append(elem)
  end while
  result.toVector
end selectionSort
```

**L.12.2 Uppgifter om trådar och jämlöpande exekvering****Lösn. uppg. 10.** *Trådar.*

- a) -
- b) `java.lang.IllegalThreadStateException`. Det går inte att starta en tråd mer än en gång. Tråden kan därför inte startas om när den redan har exekverats.
- c) När start anropas exekveras koden i run parallellt. Därför skrivs Gurka och Tomat ut omlöpande. Om istället run anropas direkt blir det inte jämlöpande exekvering och Gurka skrivs ut 100 gånger, sedan skrivs Tomat ut 100 gånger.
- d) `Thread.sleep` pausar inte tråden i exakt den tiden som angets. Alltså kommer det skrivas ut zzz snark hej! i de flesta fall, men det är inte garanterat.

**Lösn. uppg. 11.** *Jämlöpande variabeluppdatering.*

- a) I `slösaSpara` hämtas saldot, ändras och placeras tillbaka i minnet - fördröjs - upprepas. Om bamse blir klar med att ladda, ändra och lagra innan skutt gör detsamma blir det problem, då de tävlar om vem som får uppdatera (eng. *race contiion*). Problemet innan en tråd kan lagra det förändrade värdet laddar den andra tråden det gamla värdet. Bara en av dessa trådar vinner racet och får lagra sitt ändrade tal och den andra ändringen går förlorad. skutt och bamse blir alltså upprörda för att inte alla dess uttag och insättningar registreras.

**Lösn. uppg. 12.** *Trådsäkra `AtomicInteger`.*

Nu är farmor-tråden garanterad att kunna ladda saldot, ta ut pengar/ändra och lagra innan vargen-tråden kan skriva över resultatet. I `slösaSpara` pausas tråden i en millisekund så vargen-tråden kan hinna ta ut pengar innan farmor-sätter hinner sätta in pengar igen och saldot blir negativt. Dock kommer alla uttag och insättningar registreras eftersom operationerna är atomära och saldot kommer återställas till noll, utan att insättningar går förlorade.

**Lösn. uppg. 13.** *Jämlöpande exekvering med `scala.concurrent.Future`.*

- a) `error: Cannot find an implicit ExecutionContext`. `Future` behöver en `ExecutionContext` för att kunna köras. `f` är av typen `Future[Unit]`.
- b) Funktionen `printLater` har en `Future`, vilket innebär att när både `printLater` och `println` anropas i `foreach`-loopen exekveras de jämlöpande. Eftersom det tar längre tid att

starta upp en Future för datorn är println snabbare och skriver ut att alla är igång först. Sedan skrivs siffrorna från 1 - 42 ut med oregelbundna mellanrum eftersom tråden pausas olika länge.

c) big är en Future[Int]. Det stora talet har 7 520 383 siffror. r är av typen Try[Int] (se dokumentationen för Future om du är osäker)

d) Eftersom exekveringen blockas tills den har fått ett resultat går det inte att fortsätta skriva i REPL medan uträkningen pågår. Väntar man för kort tid får man ett TimeoutException och uträkningen avbryts.

### Lösn. uppg. 14. Använda Future för att göra flera saker samtidigt.

a) -

b) -

c) Varje sida fördröjs med mellan 2 upp till 3 sekunder (2000-3000 millisekunder). Så i medeltal tar det 2.5 sekunder för varje sida att laddas. Vektorn måste fyllas innan exekveringen kan fortsätta. Därför laddas alla 10 stycken sidor in innan man kan se första websidan. Det tar därför i medeltal  $2.5 \times 10 = 25$  sekunder.

d) f ger en Vektor fylld med strängar där varje element ges av en rad på hemsidan. Då f körs i bakgrunden kan programmet fortlöpa medan innehållet räknas ut. Du kan därför skriva f i REPL:n men det är inte säkert att processen är klar och det slutgiltiga resultatet visas.

e) Samma som ovan, förutom att det blir en vektor där varje element är i sig en vektor med strängar.

f) Ladda data parallellt så att nedladdningen sker samtidigt, men det går olika snabbt pga metoden seg.

g) Eftersom datan laddas i parallella trådar utan blockering blir de inte klara i ordning, utan i den ordningen tråden körs klart. Till slut blir alla klara och resultatet visar en vektor med **true** värden.

h) Metoden lycka är väldefinierad och kastar därför inga undantag. Den skriver alltid ut :). Metoden olycka är inte väldefinierad då division med 0 ger java.lang.ArithmeticException. Detta fångas upp vid callbacken och det skrivs ut :( samt det specificerade undantaget.

### L.12.3 Extrauppgifter; träna mer

### Lösn. uppg. 15.

```
def isPrime(n: BigInt): Boolean = n match
  case _ if (n <= 1) => false
  case _ if (n <= 3) => true
  case _ if n % 2 == 0 || n % 3 == 0 => false
  case _ =>
    var i = BigInt(5)
    while i * i < n do
      if (n % i == 0 || n % (i + 2) == 0) false
      i += 6
    end while
    true
```



```
end isPrime

import scala.concurrent.*
import ExecutionContext.Implicits.global

val primes = Vector.fill(10)(Future{nextPrime(randomBigInt(16))})
primes.foreach(_._onSuccess{case i => println(i)})
```

### Lösn. uppg. 16. Svara på teorifrågor.

a) Stackoverflow ger följande förklaring:

A thread is an independent set of values for the processor registers (for a single core). Since this includes the Instruction Pointer (aka Program Counter), it controls what executes in what order. It also includes the Stack Pointer, which had better point to a unique area of memory for each thread or else they will interfere with each other.

b)

```
val thread = new Thread(new Runnable{
  def run(){println('Det här är en tråd')}
})
```

c) thread.start

d) Det kan bli kapplöpning(race conditions) om vilken tråds resurser blir sparade. Vilket leder till att de andra trådarnas ändringar blir ignorerade.

e) Trådsäkerhet innebär att flera trådar kan köras parallellt utan felaktigheter i resultatet. Exempelvis får man vara väldigt försiktig om man vill ha en muterbar variabel som alla trådar ska ändra samtidigt.

f) Till exempel slipper man skapa instanser av klassen Thread eftersom man kan placera koden i en Future istället. Den löser även mycket under huven för kodaren.

### Lösn. uppg. 17. –

### Lösn. uppg. 18. Skapa din egen multitrådade webbserver.

a) abbasillen skrivs ut baklänges till nellisabba.

b)

c)

d)

e)

f)

g)

h)

i)

Lösningförslag:

```
1 package fibserver.threaded.memcache.whileloop
2
3 import java.net.{ServerSocket, Socket}
4 import java.io.OutputStream
```

```

5 import java.util.Scanner
6 import scala.util.{Try, Success, Failure}
7 import scala.concurrent._
8 import ExecutionContext.Implicits.global
9
10 object html:
11   def page(body: String): String = //minimal web page
12     s"""<!DOCTYPE html>
13       |<html><head><meta charset="UTF-8"><title>Min Sörver</title></head>
14       |<body>
15       |$body
16       |</body>
17       |</html>
18       """.stripMargin
19
20   def header(length: Int): String = //standardized header of reply to client
21     s"HTTP/1.0 200 OK\nContent-length: $length\nContent-type: text/html\n\n"
22
23   def insertBreak(s: String, n: Int = 80): String =
24     if s.size < n then s else s.take(n) + "<br>" + insertBreak(s.drop(n), n)
25
26 object compute:
27   import java.util.concurrent.ConcurrentHashMap
28   val memcache = new ConcurrentHashMap[BigInt, BigInt]
29
30   def fib(n: BigInt): BigInt =
31     if memcache.containsKey(n) then
32       println("CACHE HIT!!! no need to compute: " + n)
33       memcache.get(n)
34     else
35       println("cache miss :( must compute fib: " + n)
36       val f = superFib(n)
37       memcache.put(n, f)
38       f
39
40   private def superFib(n: BigInt): BigInt =
41     if n <= 0 then 0
42     else if n == 1 || n == 2 then 1
43     else
44       var secondLast: BigInt = 1
45       var last: BigInt = 1
46       var sum: BigInt = secondLast + last
47       var i = 3
48       while i < n do
49         if memcache.containsKey(i) then
50           sum = memcache.get(i)
51         else
52           secondLast = last
53           last = sum
54           sum = secondLast + last
55           memcache.put(i, sum)
56       i += 1
57       sum
58
59
60 object start:
61
62   def fibResponse(num: String) =
63     num.toIntOption match
64       case Some(n) => html.page(s"fib($n) == " + compute.fib(n))
65       case None    => html.page(s"FEL: skriv ett heltal, inte $num")
66
67   def errorResponse(uri:String) = html.page(s"Error: $uri <br> use /fib/heltal")
68

```

```
69 def handleRequest(cmd: String, uri: String, socket: Socket): Unit =
70   val os = socket.getOutputStream
71   val afterSlash = uri.toString.drop(1) // skip initial slash
72   println(s"afterSlash:$afterSlash")
73   val response: String =
74     if afterSlash.startsWith("fib/") then fibResponse(afterSlash.stripPrefix("fib/"))
75     else errorResponse(uri)
76   os.write(html.header(response.size).getBytes("UTF-8"))
77   os.write(response.getBytes("UTF-8"))
78   os.close
79   socket.close
80 end handleRequest
81
82 def serverLoop(server: ServerSocket): Unit =
83   println(s"http://localhost:${server.getLocalPort}/hej")
84   while true do
85     Try {
86       var socket = server.accept // blocks thread until connect
87       val scan = new Scanner(socket.getInputStream, "UTF-8")
88       val (cmd, uri) = (scan.next, scan.next)
89       println(s"Request: $cmd $uri")
90       Future { handleRequest(cmd, uri, socket) }.recover {
91         case e => println(s"Request failed: $e")
92       }
93     }.recover{ case e: Throwable => s"Connection failed: $e" }
94
95 def main(args: Array[String]) =
96   val port = Try{ args(0).toInt }.getOrElse(8089)
97   serverLoop(new ServerSocket(port))
```

## L.13 Lösning examprep

**Lösn. uppg. 1.** *Gör klart ditt projekt. —*

**Lösn. uppg. 2.** *Gör en extenta. —*

**Lösn. uppg. 3.** *Förbered din projektredovisning. —*

**Lösn. uppg. 4.** *Skapa dokumentation för ditt projekt.—*

**Lösn. uppg. 5.** *Repetera övningar och laborationer. —*

## L.14 Lösning java

### L.14.1 Grunduppgifter; förberedelse inför laboration

**Lösn. uppg. 1.** *Översätta metoder från Java till Scala.*

a)

```

1  import java.net.URL;
2  import java.util.ArrayList;
3  import java.util.{Set => JSet};
4  import java.util.{HashSet => JHashSet};
5  import java.util.Scanner;
6
7  object Hangman { // This is Java-like, non-idiomatic Scala code!
8      private var hangman: Array[String] = Array[String](
9          " ===== ",
10         " | /   | ",
11         " |    0  ",
12         " |   -|-  ",
13         " |   / \  ",
14         " |       ",
15         " |       ",
16         " ===== RIP  :(");
17
18     private def renderHangman(n: Int): String = {
19         var result: StringBuilder = new StringBuilder();
20         for (i: Int <- 0 until n){
21             result.append(hangman(i));
22             if (i < n - 1) {
23                 result.append("\n");
24             }
25         }
26         return result.toString();
27     }
28
29     private def hideSecret(secret: String,
30         found: JSet[Character]): String = {
31         var result: String = "";
32         for (i: Int <- 0 until secret.length()) {
33             if (found.contains(secret.charAt(i))) {
34                 result += secret.charAt(i);
35             } else {
36                 result += '_';
37             }
38         }
39         return result;
40     }
41
42     private def foundAll(secret: String,
43         found: JSet[Character]): Boolean = {
44         var foundMissing: Boolean = false;
45         var i: Int = 0;
46         while (i < secret.length() && !foundMissing) {
47             foundMissing = !found.contains(secret.charAt(i));
48             i += 1;
49         }
50         return !foundMissing;
51     }
52
53     private def makeGuess(): Char = {
54         var scan: Scanner = new Scanner(System.in);
55         var guess: String = "";
56         while ({

```

```

57         System.out.println("Gissa ett tecken: ");
58         guess = scan.next();
59         guess.length() > 1;
60     }) ()
61     return Character.toLowerCase(guess.charAt(0));
62 }
63
64 def download(address: String, coding: String): String = {
65     var result: String = "lackalänga";
66     try {
67         var url: URL = new URL(address);
68         var words: ArrayList[String] = new ArrayList[String]();
69         var scan: Scanner = new Scanner(url.openStream(), coding);
70         while (scan.hasNext()) {
71             words.add(scan.next());
72         }
73         var rnd: Int = (Math.random() * words.size()).asInstanceOf[Int];
74         result = words.get(rnd);
75     } catch { case e: Exception =>
76         System.out.println("Error: " + e);
77     }
78     return result;
79 }
80
81 def play(secret: String): Unit = {
82     var found: JSet[Character] = new JHashSet[Character]();
83     var bad: Int = 0;
84     var won: Boolean = false;
85     while (bad < hangman.length && !won){
86         System.out.println(renderHangman(bad));
87         System.out.print("Felgissningar: " + bad + "\t");
88         System.out.println(hideSecret(secret, found));
89         var guess: Char = makeGuess();
90         if (secret.indexOf(guess) >= 0) {
91             found.add(guess);
92         } else {
93             bad += 1;
94         }
95         won = foundAll(secret, found);
96     }
97     if (won) {
98         System.out.println("BRA! :)");
99     } else {
100         System.out.println("Hängd! :(");
101     }
102     System.out.println("Rätt svar: " + secret);
103     System.out.println("Antal felgissningar: " + bad);
104 }
105
106 def main(args: Array[String] ): Unit = {
107     if (args.length == 0) {
108         var runeberg: String =
109             "http://runeberg.org/words/ord.ortsnamn.posten";
110         play(download(runeberg, "ISO-8859-1"));
111     } else {
112         var rnd: Int = (Math.random() * args.length).asInstanceOf[Int];
113         play(args(rnd));
114     }
115 }
116 }

```

b)

1 **object** hangman:

```

2  val hangman = Vector(
3    " ===== ",
4    " |/   | ",
5    " |    0 ",
6    " |   -|- ",
7    " |   / \ ",
8    " |      ",
9    " |      ",
10   " ===== RIP  :(")
11
12  def renderHangman(n: Int): String = hangman.take(n).mkString("\n")
13
14  def hideSecret(secret: String, found: Set[Char]): String =
15    secret.map(ch => if found(ch) then ch else '_')
16
17  def makeGuess(): Char =
18    val guess = scala.io.StdIn.readLine("Gissa ett tecken: ")
19    if guess.length == 1 then guess.toLowerCase.charAt(0)
20    else makeGuess()
21
22  def download(address: String, coding: String): Option[String] =
23    scala.util.Try {
24      import scala.io.Source.fromURL
25      val words = fromURL(address, coding).getLines.toVector
26      val rnd = (math.random() * words.size).toInt
27      words(rnd)
28    }.toOption
29
30  def play(secret: String): Unit =
31    def loop(found: Set[Char], bad: Int): (Int, Boolean) =
32      if secret forall found then (bad, true)
33      else if bad >= hangman.length then (bad, false)
34      else
35        println(renderHangman(bad) + s"\nFelgissningar: $bad\t")
36        println(hideSecret(secret, found))
37        val guess = makeGuess()
38        if secret contains guess then loop(found + guess, bad)
39        else loop(found, bad + 1)
40
41    val (badGuesses, won) = loop(Set(), 0)
42    val msg = if won then "BRA! :)" else "Hängd! :("
43    println(s"$msg\nRätt svar: $secret")
44    println(s"Antal felgissningar: $badGuesses")
45
46  def main(args: Array[String]): Unit =
47    if args.length == 0 then
48      val runeberg = "http://runeberg.org/words/ord.ortsnamn.posten"
49      val secret = download(runeberg, "ISO-8859-1").getOrElse("läckalånga")
50      play(secret)
51    else play(args((math.random() * args.length).toInt))

```

**Lösn. uppg. 2.** Översätta mellan klasser i Scala och klasser i Java.

a)

```

1  import java.util.List;
2  import java.util.ArrayList;
3

```

```
4 public class JPoint {
5     private int x, y;
6
7     public JPoint(int x, int y){
8         this.x = x;
9         this.y = y;
10    }
11
12    public JPoint(int x, int y, boolean save){
13        this(x, y);
14        if (save) {
15            saved.add(0, this);
16        }
17    }
18
19    public JPoint(){
20        this(0, 0);
21    }
22
23    public int getX(){
24        return x;
25    }
26
27    public int getY(){
28        return y;
29    }
30
31    public double distanceTo(JPoint that) {
32        return distanceBetween(this, that);
33    }
34
35    @Override public String toString() {
36        return "JPoint(" + x + ", " + y + ")";
37    }
38
39    private static List<JPoint> saved = new ArrayList<JPoint>();
40
41    public static Double distanceBetween(JPoint p1, JPoint p2) {
42        return Math.hypot(p1.x - p2.x, p1.y - p2.y);
43    }
44
45    public static void showSaved() {
46        System.out.print("Saved: ");
47        for (int i = 0; i < saved.size(); i++){
48            System.out.print(saved.get(i));
49            if (i < saved.size() - 1) {
50                System.out.print(", ");
51            }
52        }
53    }
54 }
```



```

52     }
53     System.out.println();
54 }
55 }

```

b) -

c)

```
case class Person(name: String, age: Int = 0)
```

d) p.\*TAB\* - copy, producArity, ProductIterator, productElement, productPrefix  
 Person.\*TAB\* - apply, curried, tupled, unapply

```

scala> p.copy
      def copy(name: String, age: Int): Person

scala> p.copy()
res0: Person = Person(Björn,49)

scala> p.copy(age = p.age + 1)
res1: Person = Person(Björn,50)

scala> Person.unapply(p)
res2: Option[(String, Int)] = Some((Björn,49))

```

### Lösn. uppg. 3. Oföränderlig Java-klass.

```

1  // Översätt:  class Point3D(val x: Int, val y: Int, val z: Int = 0)
2
3  public class JPoint3D {
4      private int x;           // Attributen måste vara privata eftersom
5      private int y;           //   val-attribut ej finns i Java.
6      private int z;           // Typen skrivs före namnet i deklARATIONER.
7
8      public JPoint3D(int x, int y, int z){// Konstruktor heter som klassen.
9          this.x = x;             // Satser måste sluta med ;
10         this.y = y;             // this behövs p.g.a. skuggning.
11         this.z = z;
12     }
13
14     public JPoint3D(int x, int y){ // I stället för default-argument.
15         this(x, y, 0);           // Anropa konstruktor i konstruktor.
16     }
17     public int getX(){           // I Java brukar man ha get i namnet på getter.
18         return x;               // Metoder som ej är procedurer måste ha return.
19     }
20
21     public int getY(){           // Metoder måste ha parenteser, även getters.
22         return y;
23     }
24
25     public int getZ(){           // Synlighet public måste anges explicit.

```

```

26     return z;
27 }
28 }

```

```

1  > code JPoint3D.java
2  > javac JPoint3D.java
3  > ls
4  JPoint3D.class  JPoint3D.java
5  > scala
6
7  scala> val p = new JPoint3D(1,2)
8  val p: JPoint3D = JPoint3D@53b1a3f8
9
10 scala> p.x
11 1 |p.x
12   |^^
13   |value x is not a member of JPoint3D
14
15 scala> p.getX
16 val res0: Int = 1

```

#### Lösn. uppg. 4. Förändringsbar Java-klass.

```

1  //Översätt class MutablePoint3D(var x: Int, var y: Int, var z: Int = 0)
2
3  public class JMutablePoint3D {
4      private int x; // Attributen brukar vara privata även i föränngsbara
5      private int y; // klasser eftersom enhetlig access och syntax för
6      private int z; // setter med tilldelning ej finns i Java.
7
8      public JMutablePoint3D(int x, int y, int z){
9          this.x = x; // Satser måste sluta med ;
10         this.y = y; // this behövs p.g.a. skuggning.
11         this.z = z;
12     }
13
14     public JMutablePoint3D(int x, int y){ // Motsv. default-argument.
15         this(x, y, 0); // Anropa konstruktor i konstruktor.
16     }
17     public int getX(){ // I Java brukar man ha get i namnet på getter.
18         return x; // Metoder som ej är procedurer måste ha return.
19     }
20
21     public int getY(){ // Metoder måste ha parenteser, även getters.
22         return y;
23     }
24
25     public int getZ(){ // Synlighet public måste anges explicit.
26         return z;
27     }
28
29     public void setX(int x){ // I Java brukar man ha set i namnet på setter.

```

```

30     this.x = x;
31 }
32
33 public void setY(int y){ // Nyckelordet void liknar Unit i Scala, men
34     this.y = y;          // det finns inget äkta tomt värde som ()
35 }
36
37 public void setZ(int zz){ // Om namn ej krockar behövs inte this, men
38     z = zz;               // man brukar ha samma parameternamn som
39 }                          // attributnamn i setters trots skuggning så
40 }                          // att man slipper hitta på nytt namn.

```

```

1 > code JMutablePoint3D.java
2 > javac JMutablePoint3D.java
3 > ls
4 JMutablePoint3D.class  JMutablePoint3D.java
5 > scala
6
7 scala> val p = new JMutablePoint3D(1,2)
8 val p: JMutablePoint3D = JMutablePoint3D@625b215b
9
10 scala> p.x
11 1 |p.x
12 |^^
13 |value x is not a member of JMutablePoint3D
14
15 scala> p.getZ
16 val res0: Int = 0
17
18 scala> p.setZ(3)
19
20 scala> p.getZ
21 val res1: Int = 3

```

### Lösn. uppg. 5. Jämföra strängar i Java.

a)

```

1 String = java.lang.String
2 Boolean = true
3 Int = 0

```

b) Exempel på 3 olika uttryck för att testa compareTo:

1. Hej kommer först då H < h.

```

"hej".compareTo("Hej")
res: Int = 32

```

2. Dessa är ekvivalenta, så compareTo returnerar 0.

```

"hej".compareTo("hej")
res: Int = 0

```

3. h kommer före ö.

```
"hej".compareTo("ö")
res: Int = -142
```

c) Exempel på 3 olika uttryck för att testa compareToIgnoreCase:

1.

```
"hej".compareToIgnoreCase("HEj")
res: Int = 0
```

2.

```
"hej".compareToIgnoreCase("Ö")
res: Int = -142
```

3. Samma som ovan, då Ö omvandlas till ö innan jämförelse.

```
"hej".compareToIgnoreCase("ö") \\ res: Int = -142
```

d)

```
1 false
2 true
3 0
```

### Lösn. uppg. 6. Linjärsökning i Java.

a)

```
public static boolean isYatzy(int[] dice){
    int col = 1;
    boolean allSimilar = true;
    while(col < dice.length && allSimilar){
        allSimilar = (dice[0] == dice[col]);
        col++; //denna raden saknades
    }
    return allSimilar;
}
```

b)

```
public static int findFirstYatzyRow(int[][] m){
    int row = 0;
    int result = -1;
    while(row < m.length){
        if(isYatzy(m[row])){
            result = row;
            break;
        }
        row++;
    }
    return result;
}
```

**Lösn. uppg. 7.** *Jämförelsestöd i Java.*

a)

b)

```
val teamComparator = new Comparator[Team]{
  def compare(o1: Team, o2: Team) = o2.rank - o1.rank
}
```

c)

d)

e)

```
case class Point(x: Int, y: Int) extends Comparable[Point] {
  def distanceFromOrigin: Double = math.hypot(x, y)
  def compareTo(that: Point): Int =
    (distanceFromOrigin - that.distanceFromOrigin).round.toInt
}
```

**Lösn. uppg. 8.** *java.util.Arrays.binarySearch***Lösn. uppg. 9.** *Auto(un)boxing.*

a) -

b) Cell har typen java.lang.Integer. När man hämtar ut värdet med c.value hämtas den primitiva typ int ut.

c) Med hjälp av autoboxing förvandlas 42 till typen Integer och kan därför jämföras med en annan Integer.

d) i.compareTo(42) fungerar på grund av autoboxing. Då JVM packar in den primitiva typ int i en Integer-objekt automatiskt.

e)

```
0 10 20 30 40 50 60 ... 390 400 410
```

```
[0]: 0
[42]: 0
NOT EQUAL
```

f)

```
1 import java.util.ArrayList;
2
3 public class Autoboxing2 {
4     public static void main(String[] args) {
5         ArrayList<Integer> xs = new ArrayList<Integer>();
6         for (int i = 0; i < 42; i++) {
7             xs.add(i);
8         }
9         for (int x: xs) {
10             int y = x * 10;
11             System.out.print(y + " ");
```

```

12     }
13     int pos = xs.size();
14     xs.add(pos, 0);
15     System.out.println("\n\n[0]: " + xs.get(0));
16     System.out.println("[ " + pos + "]: " + xs.get(pos));
17     if (xs.get(0).equals(xs.get(pos))) {
18         System.out.println("EQUAL");
19     } else {
20         System.out.println("NOT EQUAL");
21     }
22 }
23 }

```

g) 42 kommer läggas längst fram i listan istället för längst bak, då autounboxing kommer göra Integer(0) till 0 och tvärtom med variabeln pos.

h) Om man ska undersöka om två int-variabler är lika ska man använda ==, men om variablerna är av typen Integer måste man använda equals.

JVM kommer inte varna om man vänder på Integer och int, som i xs.add(0, pos).

### Lösn. uppg. 10. CollectionConverters.

a)

```

Vector[Int] -> java.util.List[Int]
Set[Char] -> java.util.Set[Char]
Map[String, Int] -> java.util.Map[String, Int]

```

b)

```

ArrayList[Int] -> scala.collection.mutable.Buffer[Int]
HashSet[Char] -> scala.collection.mutable.Set[Char]

```

Båda blir föränderliga motsvarigheter. Det visas genom att de tillhör scala.collection.mutable och både ArrayList och HashSet är föränderliga i Java.

c) scala.collection.immutable.Set

d) sm.asJava.asScala ger typen scala.collection.mutable.Map[String,Int]  
 sm.asJava.asScala.toMap ger typen scala.collection.immutable.Map[String,Int]

e) -

### Lösn. uppg. 11. Hur fungerar en switch-sats i Java (och flera andra språk)?

a) Beroende på första bokstaven i din favoritgrönsak får du olika svar såsom *gurka är gott!* vid första bokstaven *g*.

Javas switch-sats testar den första bokstaven på favoritgrönsaken genom att stegvis jämföra den med case-uttrycken. Om första bokstaven firstChar matchar bokstaven efter ett case körs koden efter kolonet till switch-satsens slut eller tills ett break avbryter switch-satsen. Matchar inte firstChar något case så finns även default, som körs oavsett vilken första bokstaven är, ett generellt fall.

b) Om case 't' körs kommer både *tomat är gott!* och *broccoli är gott!* skrivas ut, man säger att koden "faller igenom". Utan break-satsen i Java körs koden i efterkommande case tills ett break avbryter exekveringen eller switch-satsen tar slut.

**Lösn. uppg. 12.** *Fånga undantag i Java med en **try-catch**-sats.*

a)

1. Eftersom första argumentet inte är strängen *safe* görs en oskyddad division av 42 med 42 där slutsvaret 1 visas.
2. Eftersom första argumentet inte är strängen *safe* görs en oskyddad division av 42 med 0 som ger `ArithmeticException` eftersom ett tal inte kan delas med noll.
3. Eftersom första argumentet är strängen *safe* görs en skyddad division av 42 med 42 där slutsvaret 1 visas.
4. Eftersom första argumentet är strängen *safe* görs en skyddad division av 42 med 0. Denna gång fångas `ArithmeticException` av **try-catch**-satsen vilket ersätter den gamla division med en säker division med 1 där slutsvaret 42 visas.
5. Eftersom inga argument givits kastas ett `ArrayIndexOutOfBoundsException` när programmet försöker anropa `equals` metoden hos en sträng som inte finns. Detta kunde också kontrollerats av en **try-catch**-sats.

b)

```
1 TryCatch.java:16: error: variable input might not have been initialized
```

Ett kompileringsfel uppstår på grund av risken att `input` inte blivit definierad vid division.

**Lösn. uppg. 13.** *Matriser med array i Java.*

a) Vid initialisering fylls alla element i `xss` med standardvärdet för typen, 0 i fallet med `int`. Den yttre **for**-loopen i `showMatrix()` itererar över raderna i `xss`. Den inre **for**-loopen itererar i sin tur längs med elementen på den aktuella raden och skriver ut rad, kolumn och innehåll. Efter varje rad sker en radbrytning, så att en rad i utskriften även motsvarar en rad i matrisen.

Exempel på skillnader mellan användning av matriser i `scala` och `java`:

- åtkomst: `minArray(rad)(kolumn)` respektive `minArray[rad][kolumn]`
- typnamn: `Array[Array[elementTyp]]` respektive `elementTyp[][][]`
- allokering: `Array.ofDim[typ](xDim,yDim)` respektive `new typ[xDim][yDim]`

b)

```
public class JavaArrayTest {

    public static void showMatrix(int[][] m){
        System.out.println("\n--- showMatrix ---");
        for (int row = 0; row < m.length; row++){
            for (int col = 0; col < m[row].length; col++) {
                System.out.print "[" + row + " ]";
                System.out.print "[" + col + " ] = ";
                System.out.print(m[row][col] + ";");
            } System.out.println();
        }
    }
}
```

```

}

public static void fillRnd(int[][] m, int n){
    for (int row = 0; row < m.length; row++){
        for (int col = 0; col < m[row].length; col++) {
            m[row][col] = (int) (Math.random() * n + 1);
        }
    }
}

public static void main(String[] args) {
    System.out.println("Hello JavaArrayTest!");
    int[][] xss = new int[10][5];
    fillRnd(xss, 6);
    showMatrix(xss);
}
}

```

### Lösn. uppg. 14. Översätta från Java till Scala.

```

1 object showInt {
2     def show(obj: Any, msg: String = ""): Unit = println(msg + obj)
3
4     def repeatChar(ch: Char, n: Int): String = ch.toString * n
5
6     def showInt(i: Int): Unit = {
7         val leading = Integer.numberOfLeadingZeros(i)
8         val binaryString = repeatChar('0', leading) + i.toBinaryString
9         show(i, "Heltal : ")
10        show(i.asInstanceOf[Char], "Tecken : ")
11        show(binaryString, "Binärt : ")
12        show(i.toHexString, "Hex : ")
13        show(i.toOctalString, "Oktal : ")
14    }
15
16
17    import scala.io.StdIn.readLine
18    import scala.util.{Try, Success, Failure}
19
20    def loop: Unit =
21        Try { readLine("Heltal annars pang: ").toInt } match {
22            case Failure(e) => show(e); show("PANG!")
23            case Success(i) => showInt(i); loop
24        }
25
26    def main(args: Array[String]): Unit =
27        if(args.length > 0) args.foreach(i => showInt(i.toInt))
28        else loop

```



29 }

**Lösn. uppg. 15.** *Innehållslikhet och referenslikhet i Java.*

**Lösn. uppg. 16.** *Implementera innehållslikhet i Java.*

**Lösn. uppg. 17.** *Array och **for**-sats i Java.*

a) Programmet simulerar 10000 tärningskast (med slumptalsfrö 42) och skriver ut förekomsten av respektive tärningskast.

```
1 Rolling the dice 10000 times with seed 42
2 Number of 1's: 1654
3 Number of 2's: 1715
4 Number of 3's: 1677
5 Number of 4's: 1629
6 Number of 5's: 1643
7 Number of 6's: 1682
```

b) I Java används hakparenteser medan Scala har "vanliga" parenteser. En array i scala deklareras så här:

```
val scalaArray = Array.ofDim[Int](6)
```

vilket i java motsvarar: `int[] javaArray = new int[6];`

**for**-sats i scala skrivs: `for(i <- 0 to n) {...}` medan i Java skrivs:

```
for (int i = 0; i < n; i++) { ... }.
```

I Java måste semikolon skrivas efter varje sats och typen måste anges explicit vid varje variabeldeklaration.

I scala behövs inte semikolon (förutom för att separera satser på samma rad) och typer kan ofta härledas i Scala av kompilatorn och behöver inte alltid skrivas explicit.

c) Lägg till `System.out.println(i);` i for-looparna

d)

```
// DiceReg2.java
import java.util.Random;
public class DiceReg2{
    public static int[] diceReg = new int[6];
    private static Random rnd = new Random();

    public static int parseArguments(String[] args){
        int n = 100;
        if(args.length > 0) {
            n = Integer.parseInt(args[0]);
        }
        if(args.length > 1) {
            int seed = Integer.parseInt(args[1]);
            rnd.setSeed(seed);
        }
        return n;
    }

    public static void registerPips(int n) {
```

```

    for(int i = 0; i<n; i++) {
        int pips = rnd.nextInt(6);
        diceReg[pips]++;
    }
}

public static void main(String[] args) {
    int n = parseArguments(args);
    registerPips(n);
    printReg();
}
}

```

e)

```

1 // Skriver ut förekomsten av 1000 tärningskast med slumpvalsfrö 42.
2 Number of 1's: 165
3 Number of 2's: 163
4 Number of 3's: 178
5 Number of 4's: 183
6 Number of 5's: 156
7 Number of 6's: 155
8
9 // Skriver ut diceReg-attributet
10 res1: Array[Int] = Array(165, 163, 178, 183, 156, 155)
11
12 // Skriver ut diceReg-attributet efter 1000 till kast.
13 res2: Array[Int] = Array(329, 325, 349, 360, 324, 313)
14
15 // Skriver ut diceReg-attributet efter 1000 till kast.
16 res3: Array[Int] = Array(498, 484, 531, 513, 485, 489)
17
18 // Det blir kompileringsfel då attributet rnd är privat
19 <console>:11: error: value rnd is not a member of object DiceReg2
20 DiceReg2.rnd
21 ^

```

### Lösn. uppg. 18. Läs in sekvens av tal med Scanner i Java.

a)

hasNextInt() kollar om det finns ett till tal och returnerar **true** eller **false**. nextInt() läser nästa tal.

Se <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html#hasNextInt%28%29> och <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html#nextInt%28%29>.

b)

```

1 import java.util.Random;
2 import java.util.Scanner;
3
4 public class DiceScanBuggy {
5     public static int[] diceReg = new int[6];
6     public static Scanner scan = new Scanner(System.in);
7

```

```
8  public static void registerPips() {
9      System.out.println("Enter pips separated by blanks: ");
10     System.out.println("End with -1 and <Enter>.");
11     boolean isPips = true;
12     while(isPips && scan.hasNextInt()){
13         int pips = scan.nextInt();
14         if(pips >= 1 && pips <= 6) {
15             diceReg[pips-1]++;
16         } else {
17             isPips = false;
18         }
19     }
20 }
21
22 public static void printReg(){
23     for(int i = 1; i<7; i++) {
24         System.out.println("Number of " + i + "'s: " + diceReg[i-1]);
25     }
26 }
27
28 public static void main(String[] args) {
29     registerPips();
30     printReg();
31 }
32 }
```