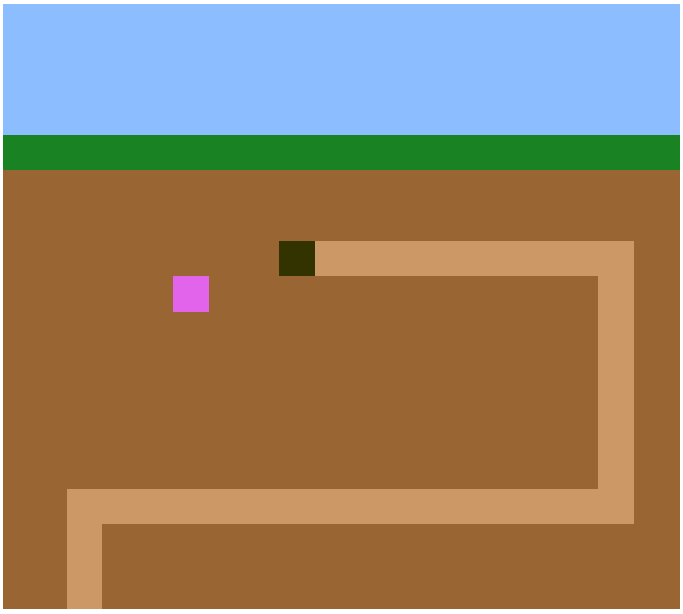
Introduktion till programmering med Scala

Kompendium 1

Första läsperioden: Modul 1 – 7



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 Isacsson, 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

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`	☐	☐	☐
`programs`	☐	☐	☐
`functions`	☐	☐	☐
`objects`	☐	☐	☐
`classes`	☐	☐	☐
`patterns`	☐	☐	☐
`sequences`	☐	☐	☐
`matrices`	☐	☐	☐
`lookup`	☐	☐	☐
`inheritance`	☐	☐	☐
`context`	☐	☐	☐
`extra`	☐	☐	☐
`examprep`	☐	☐	☐

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`	.....	.....
Projektuppgift	.....	.....

☐ `bank` *Om egendef., ge kort beskrivning här:*

☐ `music`

☐ `photo`

☐ *egendefinerad*

Muntligt prov

☐ 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 läranderesa 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 ingenjörsstudenter 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 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

Del I

Om kursen

Kapitel -2

Kursens arkitektur

-2.0.1 Veckoöversikt

W	*Modul*	*Övn*	*Lab*
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 förebereder 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	**Introduktion**	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	**Program och kontrollstrukturer**	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	**Funktioner och abstraktion**	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	**Objekt och inkapsling**	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	**Klasser och datamodellering**	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	**Mönster och felhantering**	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	**Sekvenser och enumerationer**	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	**Nästlade och generiska strukturer**	matris, nästlad samling, nästlad for-sats, typparameter, generisk funktion, generisk klass, fri och bunden typparameter, generiska datastrukturer, generiska samlingar i Scala
W09	**Mängder och tabeller**	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	**Arv och komposition**	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
W11	**Kontextuella abstraktioner och varians**	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	**Valfri fördjupning, Projekt**	välj valfritt fördjupningsområde, påbörja projekt
W13	**Repetition**	träna på extentor, 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)
  $\Longrightarrow$ 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

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. Hjälps 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 föreläsningspass 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 oumbärliga 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 arbetskapacitet, 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 fördjupnings-
★ uppgifterna. 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:

```
scala> val msg = "Hello world!"
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 ”baksätesförare”. 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 iii 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 egendefinierad 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[1] 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 ”rättningströskel”, 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.

[1] `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 beskriv 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
- Scala: en.wikipedia.org/wiki/Scala_%28programming_language%29
- git: https://en.wikipedia.org/wiki/Git_%28software%29
- GitHub: en.wikipedia.org/wiki/GitHub
- sbt: en.wikipedia.org/wiki/SBT_%28software%29

Läs mer om hur du bidrar här:
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 mostvarande 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

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[1]. 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 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 probelm. 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.

[1] Ordet repo är en förkortning av repositorium, här i betydelsen en lagringsplats för kod.

- 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.
- 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

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

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 avlutade) 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 inträddes. Den brukar sluta med att en eller flera andra personer kommenterar PR i webbgränssnitttet 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/ 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

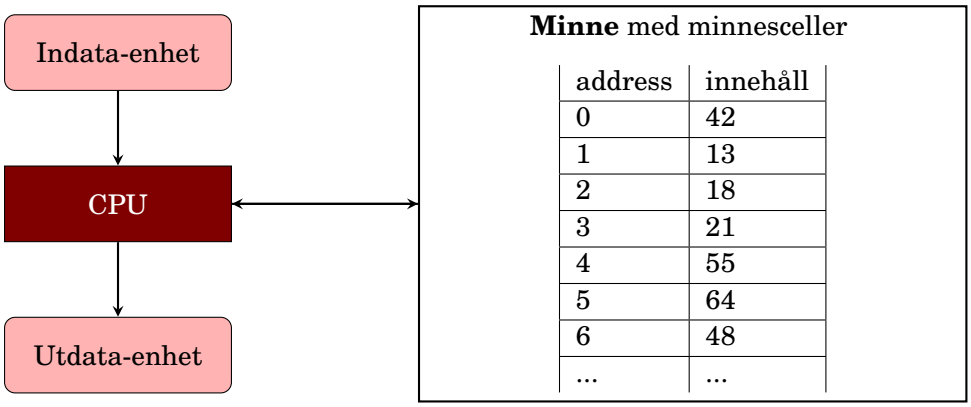
Introduktion

Begrepp som ingår i denna veckas studier:

- ☐ 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

1.1 Teori

1.1.1 Hur fungerar en dator?



address	innehåll
0	42
1	13
2	18
3	21
4	55
5	64
6	48
...	...

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

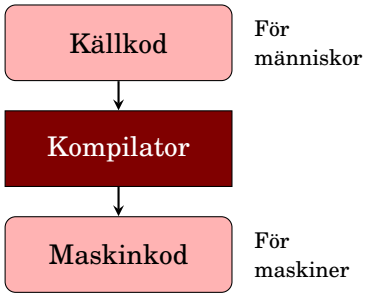
1.1.2 Vad är programmering?

- 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
- en.wikipedia.org/wiki/Computer_programming
- Ha picknick i Ada Lovelace-parken på Brunnshög!

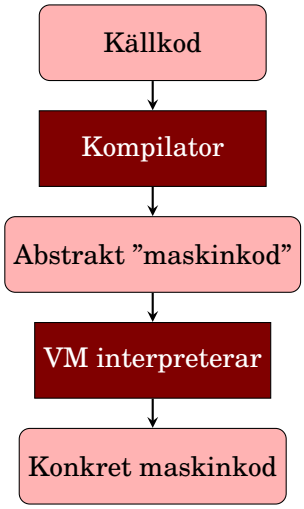
1.1.3 Vad är en kompilator?



Grace Hopper uppfann kompilatorn 1952.
en.wikipedia.org/wiki/Grace_Hopper

1.1.4 Virtuell 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 öppenkällkodsprojekt som leds av Virtuslab.
- Under 2023 ersätter **`scala-cli`** det gamla **`scala`**-kommandot.[1]
- 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"

- Literaler har en **typ** som avgör vad man kan göra med dem.

[1] 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*)[2].
- 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`

[2]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?

1.1.17 Grundtypernas omfång

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

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 deklareras 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 operatorn `+`
  `"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):

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

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

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

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

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

- **Moduloräkning** med restoperatorn `%`

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

1.1.23 Flyttalsaritmetik

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

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

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

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

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

```
scala> 0.1 + 0.2
val res6: Double = 0.30000000000000004

scala> 1E10 + 0.0000000000001
val res7: Double = 1.0E10

scala> BigDecimal("0.1") + BigDecimal("0.2")  // BigDecimal funkar
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:

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

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

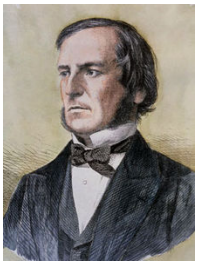
- I paketet **`scala.math`** finns många användbara funktioner: t.ex. `math.random()` ger slumptal mellan `0.0` och `0.99999999999999999`

```
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. booelsk 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 expanderas 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`

- 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`

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:

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 deklarera 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 slumptal 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>
```

a) 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.

b) 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.*

a) Skriv ett uttryck som konkatenerar två strängar, t.ex. `"gurk"` och `"burk"`, med hjälp av operatorn + 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 operatorn $*$ som upprepar resultatet från förra deluppgiften 42 gånger.

Uppgift 4. *När upptäcks felet?*

a) Vad har uttrycket `"hej" * 3` för typ och värde? Testa i REPL.

b) 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?

c) Välj ett värde på argumentet efter operatorn $*$ 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.*

a) Ta hjälp av REPL-kommadot `: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`

b) Vad händer om du adderar 1 till det största möjliga värdet av typen `Int`?
Tips: se snabbreferensen [3] under rubriken ”The Scala type system” avsnitt ”Methods on numbers”.

c) Vad är skillnaden mellan typerna `Long` och `Int`?

d) Vad är skillnaden mellan typerna `Double` och `Float`?

Uppgift 6. *Matematiska funktioner. Använda dokumentation.*

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?[4]

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:

[3] http://cs.lth.se/pgk/quickref/
[4] https://en.wikipedia.org/wiki/Wheat_and_chessboard_problem

```
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 returtypannoteringen? 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`.

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	`true : Boolean`
`4 % 42`	5	E	`false: 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*[5].

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
fram; vänster
färg(grön)
fram
```

[5] https://en.wikipedia.org/wiki/Turtle_graphics

Om du kör Kojo Desktop är det bra att börjar programmet med `sudda` (varför det?[6]).

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.

[6] 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.

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

a) Vad är det för skillnad på en sats och ett uttryck?

b) Ge exempel på satser som inte är uttryck?

c) Förklara vad som händer för varje evaluerad rad:

```
scala> def värdeSaknas = ()
scala> värdeSaknas
scala> värdeSaknas.toString
scala> println(värdeSaknas)
scala> println(println("hej"))
```

d) Vilken typ har litteralen `()`?

e) 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.[7]

a) Deklarera en förändringsbar variabel `highscore` som initieras till 0.

b) 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.

c) 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.

[7] 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.*

a) Vilket är det minsta positiva värdet av typen `Double`?

b) Vad är värdet av detta uttryck? Varför blir det så?

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

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

```
scala> if !true then println("sant") else println("falskt")
scala> if !false then println("sant") else println("falskt")
scala> def singlaSlant = if math.random() < 0.5 then "krona" else "klave"
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.*

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

b) 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`.*

a) 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.

```
scala> var x = 30
scala> x + 1
scala> x = x + 1
scala> x == x + 1
scala> val y = 20
scala> y = y + 1
scala> var z = { println("hej z!"); math.random() }
scala> def w = { println("hej w!"); math.random() }
scala> z
scala> z
scala> z = z + 1
scala> w
scala> w
scala> w = w + 1
```

b) 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`.

a) `poäng > 100 && poäng > 1000`

b) `poäng > 100 || poäng > 1000`

c) `!(poäng > highscore)`

d) `!(poäng > 0 && poäng < highscore)`

e) `!(poäng < 0 || poäng > highscore)`

f) `klar == true`

g) `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}"`

a) Vad skrivs ut nedan?

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

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

```
Kim har 3 bokstäver.
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.

```
scala> var a = 40
scala> var b = a + 40
scala> a += 10
scala> b -= 10
scala> a *= 2
scala> b /= 2
```

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

a) Läs om `BigInt` och `BigDecimal` på http://www.scala-lang.org/api/current/ Notera vad de kan användas till.

b) 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.

[8] https://en.wikipedia.org/wiki/Wheat_and_chessboard_problem

c) Vilka nackdelar finns med `BigInt` och `BigDecimal`?

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 π i `java.lang.Math` (notera stort M) respektive `scala.math.`?

```
scala> java.lang.Math.    //tryck TAB efter punkten
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 kvadratrot, 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 moduloräkning här:
en.wikipedia.org/wiki/Modulo_operation
och undersök hur det blir med olika tecken (positivt resp. negativt) på moduloräkning med $dividend\%divisor$ i Scala.

★ **Uppgift 33.** *Bokstavliga identifierare.* Läs om identifierare i Scala och speciellt *literal identifiers* här: `http://www.artima.com/pins1ed/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 ”backticks” 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 dokumentationtionen 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 deklarera 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.

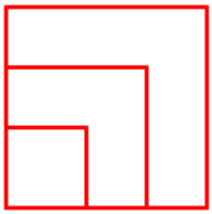
e) Studera hur anrop av proceduren `kvadrat` påverkar exekveringssekvensen av dina satser genom att göra lämpliga utskrifter så att du kan se när olika delar av koden exekveras. Vid vilka punkter i programmet sker ett ”hopp” i sekvensen i stället för att efterföljande sats exekveras? Använd lämpligt argument till `sakta` för att du ska hinna studera exekveringen.

f) Rita samma bild med 10 staplade kvadrater (se bild 1.1 på sidan 59), 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 59).

```
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 slumptal 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.

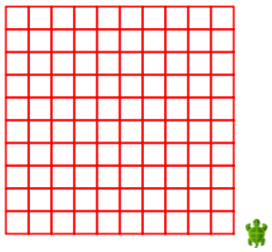
Uppgift 5. *Abstraktion och generalisering.*

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.*

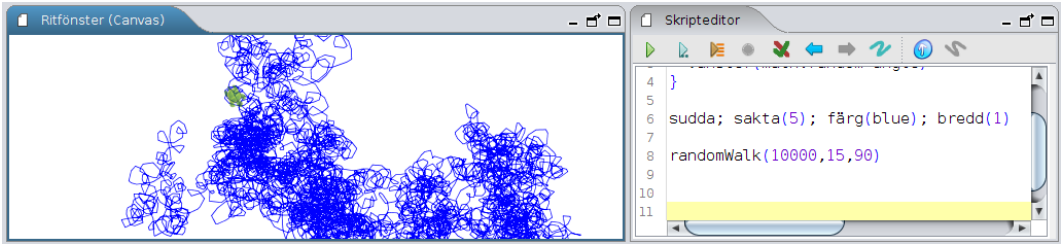
a) Bygg vidare på programmet i uppgift 4 och lägg till nedan kod i början av programmet. Lägg även till kod som gör så att om man trycker på tangenten G så sätts rutnätet omväxlande på och av. Observera att det är exakt *en* procedur som anropas vid `onKeyPress`.

```
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 formatterade 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 kontextmenyn 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?[9]

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
```

[9] 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.

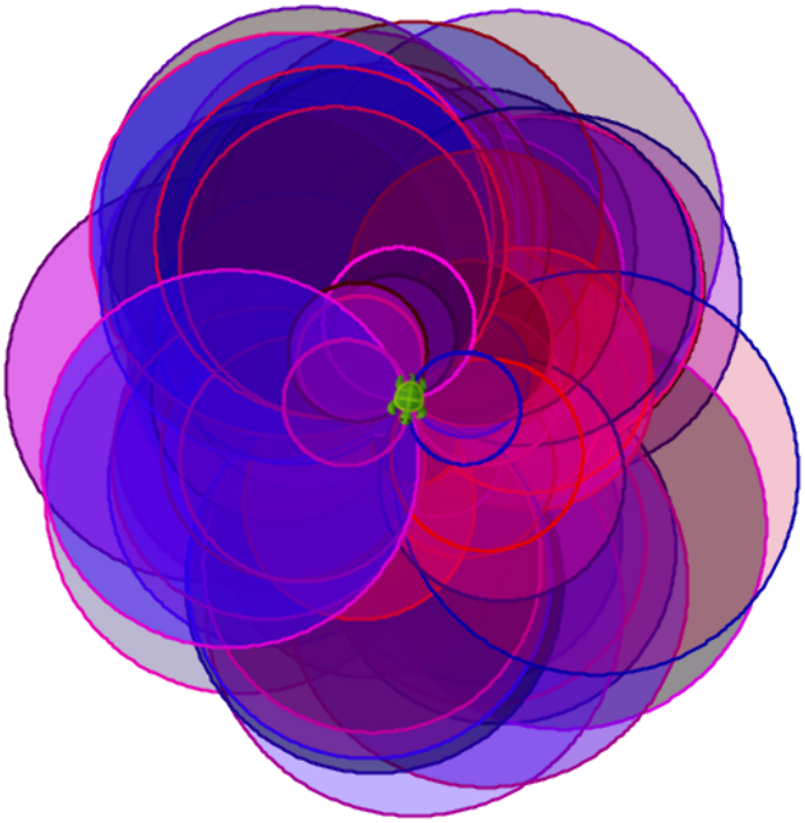
```
  var i = 0
  while (i < n) { i += 1 }
  timer.elapsedSeconds
}

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

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:

```
scala> val c = new java.awt.Color(124,10,78,100)
c: java.awt.Color = java.awt.Color[r=124,g=10,b=78]

scala> c.  // tryck på TAB
asInstanceOf    getColorComponents      getRGBComponents
brighter        getColorSpace           getRed
createContext   getComponents           getTransparency
darker          getGreen                isInstanceOf
getAlpha        getRGB                  toString
getBlue         getRGBColorComponents

scala> c.getAlpha
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 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

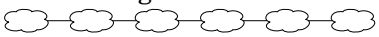
2.1 Teori

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.

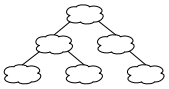
2.1.1 Vad är en datastruktur?

- 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:

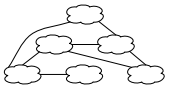
Sekvens



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 *uppräkning*) (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**[1] (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)`:

```
scala> val heltal = Vector(42, 13, -1, 0, 1)
val heltal: scala.collection.immutable.Vector[Int] = Vector(42, 13, -1, 0, 1)

scala> heltal.apply(0)   // platsnummer räknas från noll
val res0: Int = 42

scala> heltal(1)         // man kan i Scala skippa .apply före (
val res1: Int = 13

scala> heltal(5)         // ger körtidsfel då sjätte platsen inte finns
java.lang.IndexOutOfBoundsException: 5
  at scala.collection.immutable.Vector.checkRangeConvert(Vector.scala:132)
```

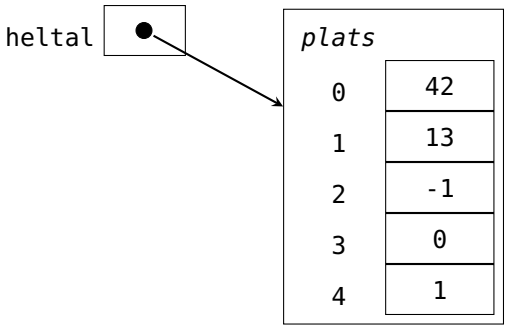
Utelämnar du `.apply` så skapar kompilatorn automatiskt ett anrop av `apply`.

---

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

2.1.5 En konceptuell bild av en vektor

```
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.)

```
scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")
grönsaker: scala.collection.immutable.Vector[String] =
  Vector(gurka, tomat, paprika, selleri)

scala> val g = grönsaker(1)
val g: String = tomat

scala> val xs = Vector(42, "gurka", true, 42.0)
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:

```
scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")

scala> for g <- grönsaker do println(g)
gurka
tomat
paprika
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"
```

```
scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")

scala> val åsikter = for (g <- grönsaker) yield s"god $g"
val åsikter: Vector[String] =
  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:

```
scala> val jättestortIntervall = Range(0, Int.MaxValue)
val jättestortIntervall: Range = Range(0, 1, 2, 3, 4, 5, ...

scala> jättestortIntervall.end
val res1: Int = 2147483647

scala> jättestortIntervall.toVector
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`

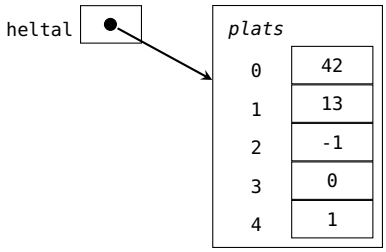
Med `to` efter ett heltal får du en `Range` till och **med** sista:

```
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 öppenkällkodsprojekt som leds av Virtuslab.
- Under 2023 ersätter **`scala-cli`** det gamla **`scala`**-kommandot.[2]
- 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
```

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:

```
> cat hej.scala
@main def run = println("Hej Scala!")

> scala-cli run hej.scala
Hej Scala!
```

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

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

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

2.1.19 Strängargument till i ett program med primitiv main

Skriv och spara nedan kod i filen `helloargs1scala`

```
> 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 `--`

```
> scala-cli run helloargs1.scala -- morot gurka tomat
morot
gurka
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 `--`

```
> scala-cli run helloargs2.scala -- 2 morot gurka tomat
morot
gurka
> scala-cli run helloargs2.scala -- aj morot gurka tomat
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**.[3]

[3] En nackdel med Scala-skript är att de ej kan inkludera skript i andra kodfiler.

- 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
```

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 ← 1
3 i ← 2
4 while i ≤ n do
5     prod ← prod * i
6     i ← 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 ← det största heltalet som kan uppkomma
3  n ← antalet heltal
4  i ← 0
5  while i < n do
6      x ← args(i).toInt
7      if ( x < min) then
8          min ← x
9      end
10     i ← 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.

```
scala> def a = { val lokaltNamn = 42; println(lokaltNamn) }
scala> a
42

scala> println(lokaltNamn)
1 |println(lokaltNamn)
  |         ^^^^^^^^^^
  |         Not found: lokaltNamn

scala> def b = { val x = 42; { val x = 76; println(x) }; println(x) }
scala> def c = { val x = 42; { val b = x + 1; println(b) } }
scala> b  // vad händer?
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/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

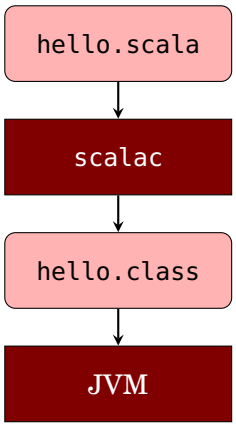
Kojo-labben gav exempel på **funktionell nedbrytning** där ett antal abstraktioner skapas och återanvänds.

```
// 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



Källkodsfil

Kompilatorn skapar *abstrakt* maskinkod
(s.k. bytekod)

`.class`-fil med bytekod

Java Virtual Machine
Översätter bytekod till *konkret*
maskinkod som passar din specifika CPU
under körning (s.k. interpretering)

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.[4]

[4] 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`.

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***.

2.1.38 Jar-filer

- `jar`-filer liknar `zip`-filer och används för att sammanföra många kompilerade 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 teminalfönster 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:

```
> 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[5]. 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):

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

[5]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	`xs == ys` är **`true`** om alla element lika
Array	2	B	olikt andra Scala-samlingar kollar == 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:

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

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

Uppgift 8. *Algoritm: 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.
Lösningsidé: 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`:

```
scala> var x = 42; var y = 43
scala> ???
scala> println(s"x är $x, y är $y")
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:

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

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

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 $\quad sum \leftarrow sum + 1$
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
max ← det minsta heltalet som kan uppkomma
n ← antalet heltal
i ← 0
while i < n do
    x ← args(i).toInt
    if ( x > max) then
        max ← x
    end
end
max
```

a) Kör med penna och papper. Det finns en bugg i algoritmen ovan. Vilken? Rätta buggen.

b) 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`

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

c) Skriv om algoritmen så att variabeln *max* initialiseras med det första talet i sekvensen.

d) 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.
minPos ← 0
i ← 1
while i < n do
    if xs(i) < xs(minPos) then
        minPos ← i
    end
    i ← i + 1
end
if n > 0 then
    minPos
else
    −1
end
```

a) 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 loopvariablen 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?

a) `Range(1, 10)`
b) `Range(1, 10).inclusive`
c) `Range(0, 50, 5)`
d) `Range(0, 50, 5).size`
e) `Range(0, 50, 5).inclusive`
f) `Range(0, 50, 5).inclusive.size`
g) `0.until(10)`
h) `0 until (10)`
i) `0 until 10`
j) `0.to(10)`
k) `0 to 10`
l) `0.until(50).by(5)`
m) `0 to 50 by 5`
n) `(0 to 50 by 5).size`
o) `(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[6]. 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?

[6] 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 uppräkning, men alla talen i uppräkningen 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) `sekvens` refererar till nedan?

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

```
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. 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
- ☐ slumptalsfrö

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:

```
scala> def öka(a: Int): Int = a + 1

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

scala> öka(1)
val res0: Int = 2

scala> öka(1,1)
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 parameterlistorna**.
- 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 **deklareras 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*[1], *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.

[1] en.wikipedia.org/wiki/Call_stack

- **Objektheapen**: I objektheapen[2,3] 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.

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 b x	5 10 1	g
x y z	2 15 17	h

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*):

```scala
// Program i filen bmi.scala

@main
def bmi(heightCm: Int, weightKg: Int) =
  safeDiv(weightKg, heightCm * heightCm)

def safeDiv(numerator: Int, denominator: Int): (Int, String) =
  if denominator == 0 then (numerator / denominator, "")  // ser du buggen?
  else (0, "division by zero")
```

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

---

[2] en.wikipedia.org/wiki/Memory_management
[3] Ej att förväxlas med datastrukturen heap sv.wikipedia.org/wiki/Heap

3.1.11 Hur läsa en stack trace?

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

3.1.13 Funktioner är äkta värden i Scala

- En funktion är ett **äkta värde**.
- Vi kan till exempel tilldela en variabel ett **funktionsvärde**.

[4] stackoverflow.com/questions/5388584/does-java-support-inner-local-sub-methods

- 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 = Lambda7210/0x0000000841e4e040@1ce2db23

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

- 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:

```
scala> def tvåGånger(x: Int, f: Int => Int) = f(f(x))

scala> def öka(x: Int) = x + 1

scala> def minska(x: Int) = x - 1

scala> tvåGånger(42, öka)
val res1: Int = 44

scala> tvåGånger(42, minska)
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:

```
scala> xs.map(öka)
val res0: ???   // vad blir resultatet?

scala> xs.map(minska)
```

```
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:

```
scala> xs.map(öka)
val res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)

scala> xs.map(minska)
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 **äkta** (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 parametrarna 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: slumptal

- 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 slumptalsfrö (eng. *random seed*) i veckans uppgifter.

3.1.19 Slumptalsfrö: få samma slumptal varje gång

- Om man använder slumptal 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**-slumptalssekvenser.
- 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**.[5]
- 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?
```

[5] Även kallat ”lambda-värde” eller bara ”lambda” efter den s.k. lambdakalkylen. en.wikipedia.org/wiki/Anonymous_function

- 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:

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

scala> xs.map((x: Int) => x + 1)
res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

Eftersom kompilatorn här kan härleda typerna så behövs de inte:

```
scala> xs.map(x => x + 1)
res1: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

Om man bara använder parametern en enda gång i funktionen så kan man byta ut parameternamnet mot ett understreck.

```
scala> xs.map(_ + 1)
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 expanderas 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.

```
scala> def summa(x: Int, y: Int) = x + y

scala> val xs = Vector(1, 2, 3, 4, 5)

scala> xs.reduceLeft(summa)
res20: Int = 15

scala> xs.reduceLeft((x, y) => x + y)
res21: Int = 15

scala> xs.reduceLeft(_ + _)
res22: Int = 15

scala> xs.reduceLeft(_ * _)
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 kontrollstrukturer likt upprepa ovan:

- klammerparentes vid ensam paramenter
- multipla parameterlistor
- namnanrop (fördröjd 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

---

3.1.28 Värdeanrop och namnanrop

Det vi sett hittills är **värdeanrop**: argumentet evalueras **först** innan dess **värde** *sedan* appliceras:

```
scala> def byValue(n: Int): Unit = for i <- 1 to n do print(" " + n)

scala> byValue(21 + 21)
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42

scala> byValue({print(" hej"); 21 + 21})
 hej 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
```

Men man kan med **=>** före parametertypen åstadkomma **namnanrop**: argumentet **”klistras in”** i stället för **namnet** och evalueras **varje gång** (kallas även **fördröjd evaluering**):

```
scala> def byName(n: => Int): Unit = for i <- 1 to n do print(" " + n)

scala> byName({print(" hej"); 21 + 21})
 hej hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej
```

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:

```
scala> def twice(action: => Unit): Unit = { action; action }

scala> twice( { print("hej"); print("san ") } )
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(uttryck)`
mot krullparenteser: `f{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 gurka tomat tomat tomat t
```

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.[6]

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 = Lambda7339/0x0000000841eb7040@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)
```

---

[6] Den nyfikne kan läsa förslaget före omröstning här:
https://docs.scala-lang.org/sips/fewer-braces.html

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 = Lambda7356/0x0000000841ed2840@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.
- Rekrusionen 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. fakultets-beräkning. 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, defaultargument, 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 pseudoslumptal
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
slumptalsfrö	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
```

a) Definiera funktionen `öka` som har en heltalsparameter `x` och vars returvärde är argumentet plus 1. Defaultargument ska vara 1. Ange returtypen explicit.

b) Vad har uttrycket `öka(öka(öka(öka())))` för värde?

c) Definiera funktionen `minska` som har en heltalsparameter `x` och vars returvärde är argumentet minus 1. Defaultargument ska vara 1. Ange returtypen explicit.

d) Vad är värdet av uttrycket `öka(minska(öka(öka(minska(minska()))))))`

e) Vad är det för skillnad mellan parameter och argument?

Uppgift 3. *Textspelet AliensOnEarth.* Ladda ner spelet nedan [7] och studera koden.

```scala
object AliensOnEarth:
  def readChoice(msg: String, options: Vector[String]): String =
    options.indices.foreach(i => println(s"$i: ${options(i)}"))
    val selected = scala.io.StdIn.readLine(msg).toInt
    options(selected)

  def isAnswerYes(msg: String): Boolean =
    scala.io.StdIn.readLine(s"$msg (Y/n)").toLowerCase.startsWith("y")

  def randomChoice(options: Vector[String]): String =
    val selected = scala.util.Random.nextInt(options.size)
    options(selected)

  def playGame(alien: String, maxPoints: Int = 1000): Int =
    val os = Vector("penguin", "window", "apple")
    val correct = if math.random() < 0.5 then os(0) else randomChoice(os)
    val cheatCode = (os.indexOf(correct) + 1) * math.Pi
    println(s"""|Hello $alien!
                |You are an alien on Earth.
                |Your encrypted password is $cheatCode.
                |You see three strange Earth objects.""".stripMargin)
    val choice = readChoice(s"$alien wants? ", os)
    if choice == correct then maxPoints else 0

  def main(args: Array[String]): Unit =
    try
      val name = if args.size > 0 then args(0) else "Captain Zoom"
      val points = playGame(alien = name)
      if points > 0 then println(s"Congratulations $name! :)")
      println(s"You got $points points.")
    catch case e: Exception =>
      println(s"Game over. The Earth was hit by an asteroid. :(")
      if isAnswerYes("Do you want to trace the asteroid?") then
        e.printStackTrace()
```

a) 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?

[7] https://raw.githubusercontent.com/lunduniversity/introprog/master/compendium/examples/AliensOnEarth.scala

b) Para ihop kodsnuttarna nedan med bästa beskrivningen.[8]

`options.indices`	1	A	gör om en sträng till små bokstäver
`"1X2".toLowercase`	2	B	heltalssekvens med alla index i en sekvens
`Random.nextInt(n)`	3	C	slumptal i intervallet `0 until n`
`try { } catch { }`	4	D	tar bort marginal till och med vertikalstreck
`""" ... """`	5	E	fångar undantag för att förhindra krasch
`s.stripMargin`	6	F	sträng som kan sträcka sig över flera kodrader
`e.printStackTrace`	7	G	skriver ut information om ett undantag

Tips: Med hjälp av REPL kan du ta reda på hur olika delar fungerar, t.ex.:

```
scala> val os = Vector("p", "w", "a")
scala> os.indices
scala> os.indices.foreach(i => println(i))
scala> os.indexOf("w")
scala> os.indexOf("gurka")
scala> Vector("hej", "hejsan", "hej").indexOf("hej")
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`.

[8] Gör så gott du kan även om allt inte är solklart. Vissa saker kommer vi att gå igenom i detalj först under senare kursmoduler.

Uppgift 4. *Äkta funktioner.* En äkta funktion[9] (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?

```
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:

```
scala> def öka(x: Int) = x + 1
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.

`for i <- 1 to 3 yield 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[10]. Det betyder att variabler kan ha funktioner som värden och funktionsvärden kan vara argument till funktioner som har funktionsparametrar[11].

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.

```
scala> def hälsa(): Unit = println("Hej!")
scala> def fleraAnrop(antal: Int, f: () => Unit): Unit =
```

[9] Äkta funktioner uppfyller per definition *referentiell transparens* (eng. *referential transparency*) som du kan läsa mer om här: en.wikipedia.org/wiki/Referential_transparency

[10] I likhet med t.ex. Javascript, men till skillnad från t.ex. Java.

[11] Funktioner som tar funktioner som argument kallas *högre ordningens funktioner*

```
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")
```

`fleraAnrop(1, hälsa)`	1	A	`f2("Hej!\nHej!")`
`fleraAnrop(3, hälsa)`	2	B	`fleraAnrop(3, f1)`
`fleraAnrop(2, f1)`	3	C	`f3()`
`fleraAnrop(1, f3)`	4	D	`f2("Hej!")`

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 expanderas 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:

`(0 to 2).map(i => i + 1)`	1	A	`Vector(9.0, 16.0, 25.0)`
`(1 to 3).map(_ + 1)`	2	B	`(2 to 4).map(i => i - 1)`
`(2 to 4).map(math.pow(2, _))`	3	C	`Vector(2.0, 2.5, 3.0)`
`(3 to 5).map(math.pow(_, 2))`	4	D	`Vector(2, 3, 4)`
`(4 to 6).map(_.toDouble).map(_ / 2)`	5	E	`Vector(4.0, 8.0, 16.0)`

Funktionslitteraler kallas även *anonyma funktioner*[12], 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.

[12] Ett annat vanligt namn är *lambda* efter det datalogiska matematikverktyget lambdakalkyl: https://sv.wikipedia.org/wiki/Lambdakalkyl

3.2.2 Extrauppgifter; träna mer

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 deklarera 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.[13]

```
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.

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.

[13] Det är så loopen `upprepa` i Kojo är definierad.

```
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 parametrarna 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*[14] 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)
```

[14] 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?

```
scala> def explode = throw Exception("BANG!!!")
scala> explode
scala> def countdown(n: Int): Unit =
         if n == 0 then explode else countdown(n-1)
scala> countdown(10)
scala> countdown(10000)
scala> def countdown2(n: Int): Unit =
         if n == 0 then explode else {countdown2(n-1); print("no tailrec")}
scala> countdown2(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.

```
scala> def countNoTailrec(n: Long): Unit =
         if n <= 0L then println("Klar! " + n) else {countNoTailrec(n-1L); ()}
scala> countNoTailrec(1000L)
scala> countNoTailrec(100000L)
scala> import scala.annotation.tailrec
scala> @tailrec def countNoTailrec(n: Long): Unit =
         if n <= 0L then println("Klar! " + n) else {countNoTailrec(n-1L); ()}
scala> @tailrec def countTailrec(n: Long): Unit =
         if n <= 0L then println("Klar! " + n) else countTailrec(n-1L)
scala> countTailrec(1000L)
scala> countTailrec(100000L)
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 | sektionscaféer. 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

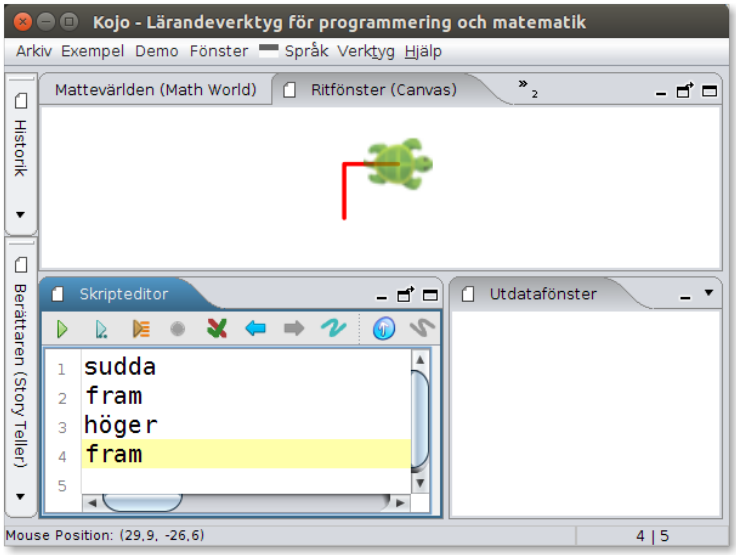
Objekt och inkapsling

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, rikting, 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, allokera, 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?

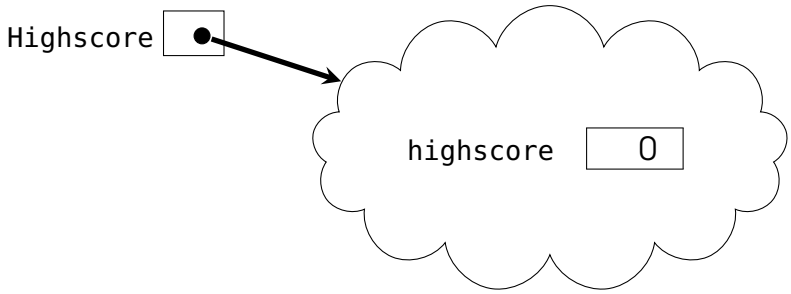
- Ett singelobjekt (eng. *singelton*) deklareras med nyckelordet `object` och används för att samla **medlemmar** (eng. *members*) som **hör ihop**.
- Ett singelobjekt kallas också **modul** (eng. *module*).
- Medlemmarna kan t.ex. vara **variabler** (`val`, `var`) och **metoder** (`def`).
- En **metod** 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.

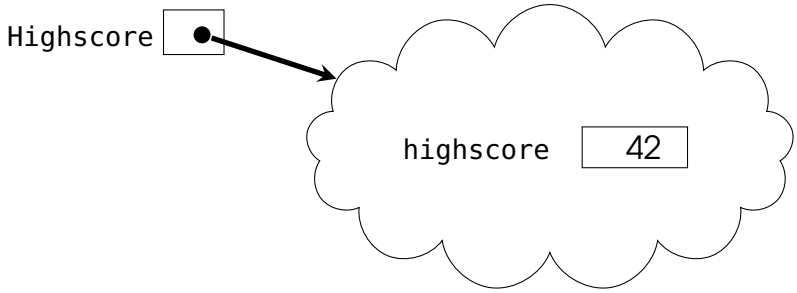
4.1.6 Allokering: minne reserveras med plats för data

```
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 metodanropet 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**[1] (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.

[1] https://sv.wikipedia.org/wiki/Namnrymd

Nackdelar:

- Begränsar användningen, har ej tillgång till alla delar.
- Svårare att experimentera med ett API medan man försöker förstå det.

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
```

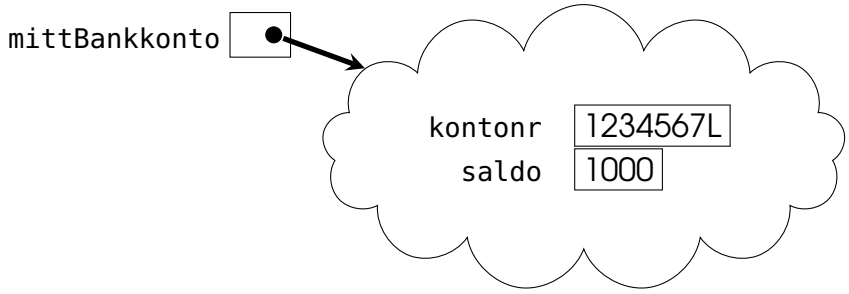
```
scala> mittBankkonto.saldo -= 25000

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.

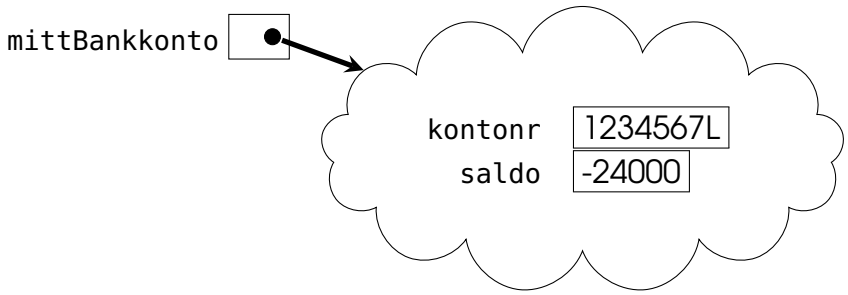
```
object mittBankkonto
  val kontonr: Long        = 1234567L
  var saldo: Int           = 1000
  def ärSkuldsatt: Boolean = saldo < 0
```



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:[2]
  - **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`.

[2] en.wikipedia.org/wiki/Modular_programming

4.1.18 Kompilera paket

Paketdeklarationer medför att kompilatorn placerar bytekodfiler i en katalog med samma namn som paketet:

```
> scalac f1.scala f2.scala     // samkompilering av två filer
> ls
f1.scala  f2.scala  mittpaket
> ls mittpaket
A.class  'A$.class'   A.tasty
B.class  'B$.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)`**

```
scala> val t = ("hej", 42, math.Pi)
t: (String, Int, Double) = (hej,42,3.141592653589793)

scala> t._1    // direkt access
res0: String = hej

scala> t(1)    // notera användningen av apply
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**:

```
scala> polär(3,4)
res3: (Double, Double) = (5.0,0.6435011087932844)
```

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**:

```
scala> ("Ålder", 42)
res0: (String, Int) = (Ålder,42)

scala> "Ålder".->(42)
res1: (String, Int) = (Ålder,42)

scala> "Ålder" -> 42
res2: (String, Int) = (Ålder,42)

scala> Vector("Ålder" -> 42, "Längd" -> 178, "Vikt" -> 65)
res3: scala.collection.immutable.Vector[(String, Int)] =
        Vector((Ålder,42), (Längd,178), (Vikt,65))
```

4.1.23 Typalias för att abstrahera typnamn

Med hjälp av nyckelordet **type** kan man deklarera ett **typalias** för att ge ett **alternativt** namn till en viss typ. Exempel:

```
scala> type Pt = (Int, Int)            // typalias
scala> type Pts = Vector[Pt]           // nästlat typalias

scala> def distToOrigo(pt: Pt): Double = math.hypot(pt._1, pt._2)

scala> val xs: Pts = Vector((1,1), (2,2), (3,4))
val xs: Pts = Vector((1,1), (2,2), (3,4))

scala> xs.head
val res0: Pt = (1,1)

scala> xs.map(distToOrigo)
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.

```
scala> val strikt = Vector.fill(1000000)(math.random())
strikt: scala.collection.immutable.Vector[Double] =
 Vector(0.7583305221813246, 0.9016192590993339, 0.770022134260162, 0.1566771818

scala> lazy val lat = Vector.fill(1000000)(math.random())
lat: scala.collection.immutable.Vector[Double] = <lazy>

scala> lat
res0: scala.collection.immutable.Vector[Double] =
  Vector(0.5391685014341797, 0.14759775960530275, 0.722606095900537, 0.90255727
```

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

---

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
```

`-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 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:

- **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
```

```
scala> val bk1 = new Bankkonto(1234567L, 1000 )   // instansiera en klass
bk1: Bankkonto = Bankkonto@5d7399f9

scala> val bk2 = new Bankkonto(6789012L, -200 )
bk2: Bankkonto = Bankkonto@286855ea

scala> bk1.saldo
res0: Int = 1000

scala> bk2.ärSkuldsatt
res1: Boolean = true
```

4.1.34 Använda klassen `Color`

- I JDK (Java Development Kit) finns hundratals paket (moduler) och tusentals färdiga klasser. [3]
- 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.

```
scala> val röd = java.awt.Color(255, 0, 0)    //  en maximalt röd färg

scala> import java.awt.Color  // namnet Color tillgängligt i aktuell namnrymd

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.

```
scala> extension (s: String) def skrikBaklänges = s.reverse.toUpperCase
def skrikBaklänges(s: String): String

scala> "hejsan".skrikBaklänges
val res1: String = NASJEH

scala> skrikBaklänges("goddag")
val res2: String = GADDOG
```

4.1.36 Kollektiva extensionsmetoder

- Det går bra att sammanföra flera funktioner under en och samma **`extension`** så här:

[3] https://stackoverflow.com/questions/3112882/

```
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:

```
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:

```
scala> import java.awt.Color as JColor //importera och byt namn

scala> val grön = JColor(0, 255, 0)  //skapa instans med nya namnet
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:

```
scala> import java.awt.{Event as _, *}  // importera allt UTOM Event
```

- Kan kombineras med namnbyte och allimport:

```
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 funktions-kropp.
- 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)

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 + kodbiblitek 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 = ???
```

a) 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.

b) 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.

c) 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.*

a) Jämför **`import`** och **`export`** genom att beskriva en likhet och en skillnad.

b) 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 = ???
```

a) Funktionen `djup` ska ge z-koordinaten för Mullvaden. Vilken typ har `djup`?

b) Vilken typ har `hemlis`?

c) 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.

d) 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.

a) 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
```

b) Prova ovan igen men med så stor array att minnet blir fullt. När sker allokeringen?

c) 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?

d) Vad händer i REPL om du refererar de tre olika `a`-variablerna?

e) 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.

a) Skapa extensionsmetod på heltal som möjliggör inkrementering.

```
scala> 42.inc
val res0: Int = 43
```

b) Skapa extensionsmetod på heltal som möjliggör dekrementering.

```
scala> 42.dec
val res1: Int = 41
```

c) 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):

```
> scala-cli repl --jar introprog_3-1.3.1.jar
scala> val w = introprog.PixelWindow(400,300,"HEJ")
scala> w.line(100, 100, 200, 100)
scala> w.line(200, 100, 200, 200)
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/

```scala
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 exempelhuvudprogram 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 nämna `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[4]. 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`.

[4] 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 horisontell 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.

```
> scala-cli repl --dep se.lth.cs::introprog:1.3.1
scala> val w = new introprog.PixelWindow(400,300,"Tre nyanser av brunt")
scala> type Pt = (Int, Int)
scala> var color = java.awt.Color.red
scala> def rak(p: Pt)(d: Int) = w.line(p._1, p._2, ???, ???, color)
scala> def fyll(p: Pt)(s: Int) = for i <- ??? do rak((p._1, ???))(s)

scala> object Color:
     |   ???

scala> color = Color.soil
scala> fyll(100,100)(75)

scala> color = Color.tunnel
scala> fyll(100,100)(50)

scala> color = Color.mole
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.

```
scala> object plus { def apply(x: Int, y: Int) = x + y }
scala> plus.apply(42, 43)
```

Går det att utelämna `.apply` och anropa `plus` som en funktion?

Uppgift 11. *Skapa moduler med hjälp av singelobjekt.*

a) Undersök i REPL vad uttrycket `"päronisglass".split('i')` har för värde.

b) 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--- FREKVENSANALYS 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()
```

c) Vilket av objekten i modulen `stringUtils` har tillstånd? Är det förändringsbart?

d) 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.

```
> scala-cli repl --dep se.lth.cs::introprog:1.3.1
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`

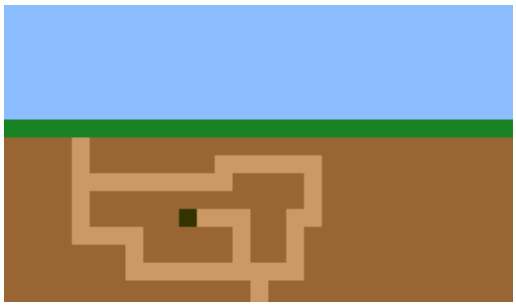
Mål

- ☐ 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 kvadratiska 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 importdeklaration 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` [5]. Använd namngivna argument vid skapandet av fönstret. Tycker du att koden blir mer läsbar med namngivna argument? [6]

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)
```

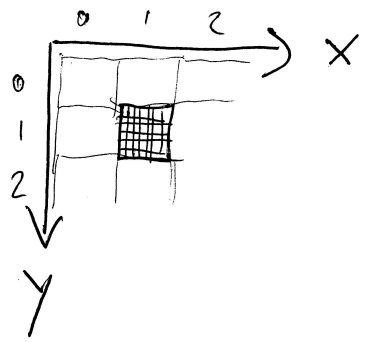
Kompilera och kör och säkerställ att allt fungerar som förväntat.

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 500 bildpunkter högt. Använd figur 4.1 för att förklara relationen mellan underliggande fönsterkoordinater och blockkoordinater. Notera att y-axeln pekar nedåt.

[5] http://cs.lth.se/pgk/api/

[6] 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-gånger-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 ritar en rektangel med hjälp av proceduren `block`. Sen ska du använda `rectangle` i `Main.drawWorld` för att rita upp mullvadens underjor-

diska 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 ritar 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 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 tangentbordsavläsning 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:

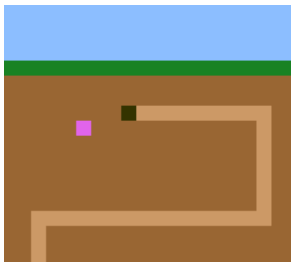
```
def delay(millis: Int): Unit = Thread.sleep(millis)
```

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.currentTimeMillis` 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
- ☐ isInstanceOf
- ☐ 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 deklareras 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 **stämplar** – motsvarar **instansiering**.
- Då skapas **avbildningar** av stämpeln – motsvarar **allokering av ett objekt** som är en **instans** av klassen.
- Allokering kallas också **konstruktion** och funktionen/koden som gör själva allokeringen kallas **konstruktor**.

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 deklarationssystem, 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.

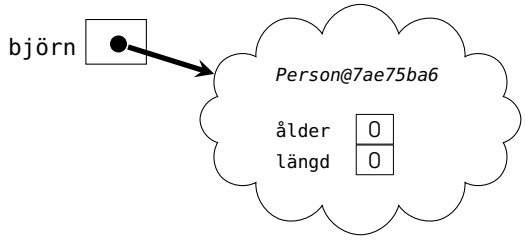
5.1.4 Singelobjekt jämfört med klass

Vi har tidigare deklarerat **singelobjekt** som bara finns i **en** enda upplaga:

```
scala> object Björn { var ålder = 54; val längd = 178 }
```

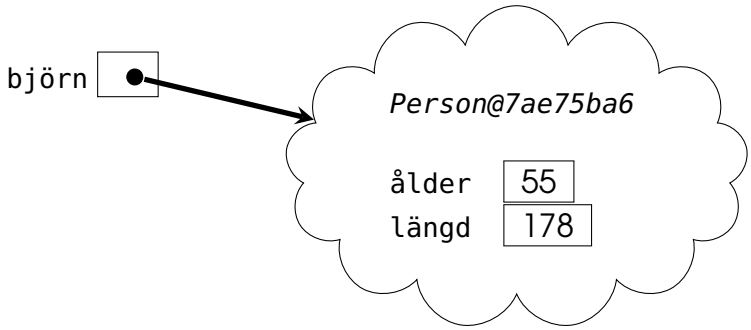
Med en **klass** kan man skapa **godtyckligt många instanser av klassen** med hjälp av nyckelordet **`new`** följt av klassens namn:

```
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
```



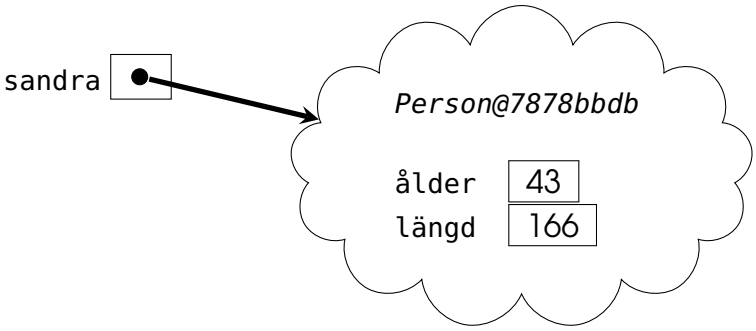
5.1.5 Förändring av objektets tillstånd

```
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:

```
scala> class C(a: Int){ def add(other: C): Int = a + other.a }
-- Error:
1 |class C(a: Int){ def add(other: C): Int = a + other.a }
  |                                            ^^^^^^^
  | value a cannot be accessed as a member of (other : C) from class C.
```

- Men detta fungerar fint:

```
scala> class D(private val a: Int){ def add(other: D): Int = a + other.a }

scala> D(42).add(D(43))
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 infixa 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.

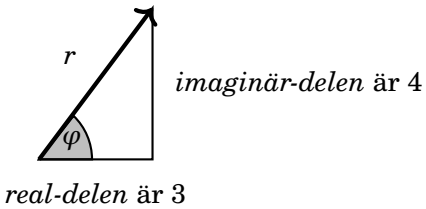
```
> scala-cli repl --scala-opt -source:future -deprecation
scala> Box(41) plus Box(1)
-- Warning:
1 |Box(41) plus Box(1)
  |        ^^^^
  |Alphanumeric method plus is not declared `infix`; it should not be used as infix ope
  |The operation can be rewritten automatically to `plus` under -deprecation -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"
```

```
scala> val z1 = new Complex(3, 4)  // konstruktion av instans av Complex
z: Complex = 3.0 + 4.0i

scala> val polärForm = (z1.r, z1.fi)
polärForm: (Double, Double) = (5.0,0.6435011087932844)

scala> val z2 = Complex(1, 2)   // new behövs inte i Scala 3
z2: Complex = 1.0 + 2.0i

scala> z1 + z2
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

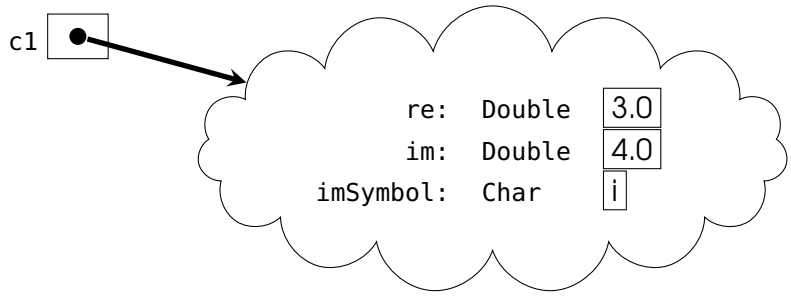
```
class Complex(val re: Double, val im: Double):
  val r  = math.hypot(re, im)
  val fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

- 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**:

```
scala> import MyFactory.*

scala> createComplex(3, 4)
res0: Complex = 3.0 + 4.0i

scala> createReal(42)
res1: Complex = 42.0 + 0.0i

```

```
scala> createImaginary(-1)
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"
```

```
scala> new Complex()
res0: Complex = 0.0 + 0.0i

scala> new Complex(re = 42)  //anrop med namngivet argument
res1: Complex = 42.0 + 0.0i

scala> new Complex(im = -1)
res2: Complex = 0.0 + -1.0i

scala> new Complex(1)
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()
```

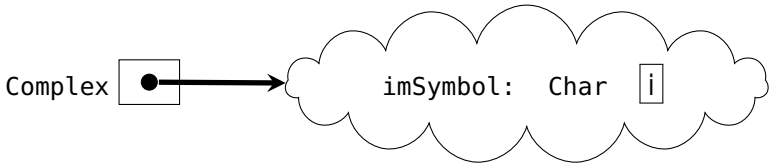
5.1.25 Medlemmar som bara behövs i en enda upplaga

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()
```

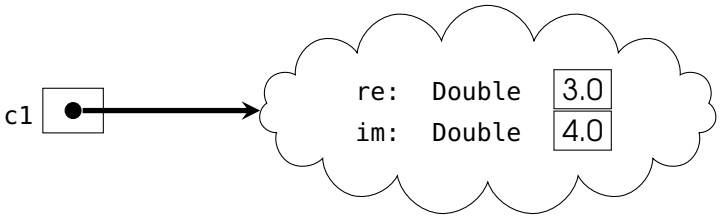
5.1.26 Medlemmar i singelobjekt är statiskt allokerade

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)
```



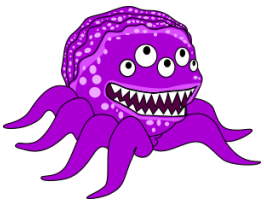
5.1.27 Attribut i kompanjonsobjekt användas för sådant som är gemensamt för alla instanser

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`.
2. Deklarera två `val`-variabler som kallas `fem1` 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[1] 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...

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:

[1] alltså **inte** instansprivata som i vanliga klasser.

- 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

```
scala> val p1 = Point(1)
p1: Point = Point@21312342

scala> val p2 = Point(1, 2)
p2: Point = Point@43254325

scala> val p3 = Point(1, 2, 3)
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`

```
scala> class Gurka(val vikt: Int)

scala> var g: Gurka = null        // ingen instans allokerad än
```

```
var g: Gurka = null

scala> g.vikt
java.lang.NullPointerException

scala> g = Gurka(42)          // instansen allokeras
g: Gurka = Gurka@1ec7d8b3

scala> g.vikt
val res0: Int = 42

scala> g = null         // instansen kommer att destrueras av skräpsamlaren
```

- Scala har **`null`** av kompabilitetsskäl, 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`	**`null`**

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"
```

```
scala> val ib1 = new InitBug1
scala> val ib2 = new InitBug2
scala> ib2.b
scala> val ib3 = new InitBug3
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"
```

```
scala> val ib1 = new InitBug1   // java.lang.NullPointerException
scala> val ib2 = new InitBug2
scala> ib2.b                    // val res0: Int = 0  // WHAT????
scala> val ib3 = new InitBug3
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 lösing 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 jagSjälv = this}

scala> val g = Gurka(42)
val g: Gurka = Gurka@5ae9a829

scala> g.jagSjälv
val res0: Gurka = Gurka@5ae9a829

scala> g.jagSjälv.vikt
val res1: Int = 42

scala> g.jagSjälv.jagSjälv.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. [2]
- 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.

---

[2] Java har visserligen **`final`** men det är annorlunda som vi ska se senare.

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;
    }
}
```

```
> scala-cli repl .
scala> val p = JPerson("Björn")
val p: JPerson = JPerson@7e77408

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 JPe
```

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
```

```
scala> val p = Person("Björn")
val p: Person = Person@28ac3dc3

scala> p.age = 42      // najs syntax om getter parad med setter enl ovan
val p.age: Int = 42

scala> p.age = -1      // nu förhindras negativ ålder
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. 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 referens**o**likhet 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 strukturlikhet.

- 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:

```
scala> val v1 = Vector(1,2,3)
v1: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> val v2 = Vector(1,2,3)
v2: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> v1 eq v2                //referenslikhetstest: olika instanser
res0: Boolean = false

scala> v1 ne v2
res1: Boolean = true

scala> v1 == v2                //innehållslikhetstest: samma innehåll
res2: Boolean = true

scala> v1 != v2
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 strukturlikhet.

```
scala> case class Gurka(vikt: Int)

scala> val g1 = Gurka(42)
g1: Gurka = Gurka(42)

scala> val g2 = Gurka(42)
g2: Gurka = Gurka(42)

scala> g1 eq g2          // olika instanser
res0: Boolean = false

scala> g1 == g2          // samma innehåll!
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 innehållslikhet (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 bottentypen `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

```scala
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 deklarera 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
`null`	12	L	indirekt åtkomst av attributvärde
`new`	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.[3]

`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 object`
`{ 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?

Uppgift 4. *Oföränderlig klass med defaultargument.* Det som gäller för parametrar och argument till funktioner är även tillämpligt på klassparametrar, t.ex. defaultargument

[3] 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

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[4].

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.

`val p1 = Point3D()`	1	A	`false`
`val p2 = Point3D(y = 1)`	2	B	`Reassignment to val`
`Point3D(z = 2).z`	3	C	`p1: Point3D = Point3D@2eb37eee`
`p2.y = 0`	4	D	`true`
`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.

`val p1 = Pt(1, 2)`	1	A	`MPt(5,6)`
`val p2 = Pt(y = 3)`	2	B	`false`
`val p3 = MutablePt(5, 6)`	3	C	`Pt(0,3)`
`val 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	`true`

[4] 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) 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 `blockmole`-labben 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 kommano. 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:

```
> scala-cli repl .
Welcome to Scala 3.3.0 (17.0.6, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> blockbattle.Pos(1,2)
val res0: blockbattle.Pos = Pos(1,2)

scala> import blockbattle.*

scala> val p = Pos(1,2)
val p: blockbattle.Pos = Pos(1,2)

scala> p.moved(0,1)
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 `has` ska ge **`true`** om tangentnamnet finns i någon av de fyra riktningstangenterna i denna denna `KeyControl`-instans, annars **`false`**.

```scala
package blockbattle

case class KeyControl(left: String, right: String, up: String, down: String):
  def direction(key: String): (Int, Int) = ???

  def has(key: String): Boolean = ???
```

```
scala> import blockbattle.*

scala> val kc1 = KeyControl(right="d",left="a",up="w",down="s")
val kc1: blockbattle.KeyControl = KeyControl(a,d,w,s)

scala> val kc2 = KeyControl("Left","Right","Up","Down")
val kc2: blockbattle.KeyControl = KeyControl(Left,Right,Up,Down)

scala> kc2.left
val res0: String = Left

scala> kc2.has("a")
val res1: Boolean = false

scala> kc2.has("Up")
val res2: Boolean = true

scala> kc1.direction("a")
val res3: (Int, Int) = (-1,0)

scala> kc1.direction("s")
val res4: (Int, Int) = (0,1)

scala> kc1.direction("d")
val res5: (Int, Int) = (1,0)

scala> kc1.direction("w")
val res6: (Int, Int) = (0,-1)

scala> Pos(1,2).moved(kc1.direction("a"))
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.

```scala
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*).[5]

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 kvadratiskt 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(
```

[5] 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.

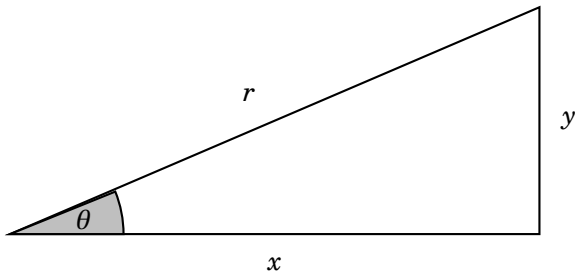
Uppgift 8. *Skapa en punktklass som kan hantera polära koordinater.* Du ska skapa en oföränderlig case-klass `Point` som kan representera en koordinat både med ”vanliga” kartesiska koordinater[6] och med polära koordinater[7].

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 koortdinater. Metoden ska ha detta huvud:
   `def polar(r: Double, theta: Double): Point`

Tips:

- Du har nytta av metoderna $r =$ `math.hypot(x, y)` och $\theta =$ `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 upprepad 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`.

[6] https://sv.wikipedia.org/wiki/Kartesiskt_koordinatsystem
[7] https://sv.wikipedia.org/wiki/Pol%C3%A4ra_koordinater

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?

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.

```
scala> val (s1, s2) = (Square(), Square(1, 10, 1))
scala> val s3 = s1 moved (1,-5)
scala> s1 isEqualSizeAs s3
scala> s2 isEqualSizeAs s1
scala> s1 isEqualSizeAs Square.unit
scala> s2.scale(math.Pi) isEqualSizeAs s2
scala> s2.scale(math.Pi) isEqualSizeAs s2.scale(math.Pi)
scala> s2.scale(math.Pi) eq s2.scale(math.Pi)
scala> Square.unit eq Square.unit
```

5.2.3 Fördjupningsuppgifter; utmaningar

Uppgift 11. *Innehållslikhet mellan olika typer.* Klistra in nedan klasser i REPL och undersök vad som händer.

```
class Gurka(val vikt: Int)

class Bil(val typ: String)
```

```
scala> class Gurka(val vikt: Int)
     |
     | class Bil(val typ: String)
// defined class Gurka
// defined class Bil

scala> 42 == "Fyrtiotvå"

scala> Gurka(50) == Bil("Sedan")
```

Finns det något resultat som är problematiskt, och i så fall, varför?

Uppgift 12. *Attributrepresentation. Privat konstruktor. 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()
```

a) Vad händer om du försöker instansiera Kim Kodkunnigs klass direkt med nyckelordet `new`?

b) Varför använder Kim Kodkunnig ett kompanjonsobjekt med en fabriksmetod? Vilka accessregler gäller mellan ett kompanjonsobjekt och klassen med samma namn?

c) 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 konstruktor,* ***`private`****`[this]`*.

a) 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?

```
scala> new Gurka1(42).vikt
scala> new Gurka2(42).vikt
scala> new Gurka3(42).vikt
scala> val ingenGurka: Gurka4 = null
scala> new Gurka4(42, ingenGurka).kompisVikt
scala> new Gurka4(42, new Gurka4(84, null)).kompisVikt
scala> new Gurka6(42)
scala> new Gurka7(-42)
scala> Gurka7(-42)
scala> val g = Gurka7(42)
scala> g.vikt
scala> g.vikt = -1
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?

```
scala> val g = Gurka8(-42)
scala> val g = Gurka8(42)
scala> g.vikt
scala> g.vikt = 0
scala> g.vikt = -1
scala> g.vikt += 42
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-postitionen. 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 slumpässigt 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. [8]

```
/** 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 */
```

[8] D.v.s. ett anrop av metoden `scale` orsakar ingen omedelbar kostnad.

```
  def cost: Double = ???

  /** Returns the accumulated cost. Sets the accumulated cost to zero. */
  def pay: Double = ???

  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:

```
scala> val xs = Vector.fill(10)(Square())
scala> xs.foreach(_.move(2, 3))
scala> xs.foreach(_.scale(2.9))
scala> val (m, c) = (Square.totalNumberOfMoves, Square.totalCost)
val m: Int = 10
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[9], är att definiera flera konstruktorer 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(...) = ...` Hjälpkonstruktorer 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 hjälpkonstruktorer här:
www.artima.com/pins1ed/functional-objects.html#6.7

b) Hitta på en egen uppgift med hjälpkonstruktorer, baserat på någon av klasserna i tidigare övningar.

[9] 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`

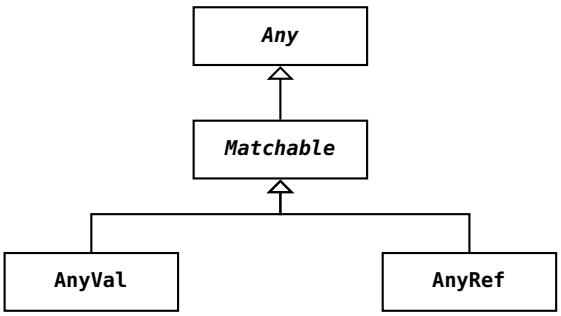
Scalas typsystem är **fullständigt**:

- 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`



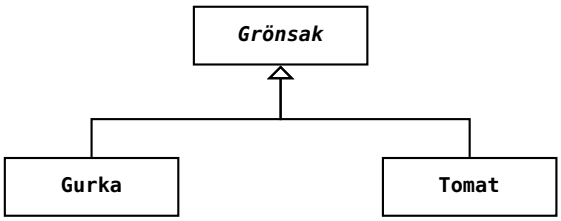
- Alla **värdetyper**, t.ex. `Int`, `Double`, `Boolean`, är subtyper till `AnyVal`
- Alla **referenstyper** t.ex. `String` är subtyper till `AnyRef`
- Värden av typen `Matchable` kan användas vid s.k mönstermatchning.
- (Det finns även s.k. opaka typer som inte kan mönstermatchas.)

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 topptypen `Any`.

```
scala> (f().isInstanceOf[Matchable], f().isInstanceOf[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 basklass 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:[1]

```
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 uppräkning 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?

```
scala> parallellFärg(Ruter)
scala.MatchError: Ruter
```

Undantag vid körtid :(

6.1.18 Mönstermatchning och förseglade typer

Med nyckelordet **sealed** får vi en kompileringsvarning.

```
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
```

[1] https://youtu.be/aboZctrHfK8

```
1 |def parallellFärg(f: Färg): Färg = f match
  |                                    ^
  |                            match may not be exhaustive.
  |
  |                            It would fail on pattern case: Ruter
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. uppräkning, 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
```

```
def parallellFärg(f: Färg): Färg
3 |  f match
  |  ^
  |  match may not be exhaustive.
  |
  |  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.

```
scala> val livetsMening = 42

scala> def ärLivetsMeningBuggig(svar: Int) = svar match
         case livetsMening => true    // lokalt namn som matchar allt!
         case _ => false

scala> ärLivetsMeningBuggig(43)
val res0: Boolean = true

scala> val LivetsMening = 42   // stor begynnelsebokstav

scala> def ärLivetsMening(svar: Int) = svar match
         case LivetsMening => true    // funkar fint!
         case _ => false

```

```
scala> ärLivetsMening(43)
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!

```
scala> val livetsMening = 42

scala> def ärLivetsMeningBackTicks(svar: Int) = svar match
         case `livetsMening` => true    // nu funkar det!
         case _ => false

scala> ärLivetsMeningBackTicks(43)
val res2: Boolean = false
```

6.1.22 Mönster på andra ställen än i `match`

Mönster i **deklarationer**:

```
scala> case class Point(x: Int, y: Int)

scala> val p = Point(0, 1)

scala> val Point(x, y) = p          // konstruktormönster med case-klass
val x: Int = 0
val y: Int = 1

scala> val (x, y, z) = (0, 1, 2)    // konstruktormönster med tupel
val x: Int = 0
val y: Int = 1
val z: Int = 2
```

Mönster i **for-uttryck**:

```
scala> val xs = for (x, y) <- Vector((1,2), (3,4)) yield x
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`.

```
scala> case class Gurka(vikt: Int, ärRutten: Boolean)

scala> Gurka.unapply // tryck ENTER för att se typen
val res0: Gurka => Gurka = Lambda1914/0x00000008408cf840@b0e7bde

scala> val g = Gurka(100, false)

scala> Gurka.unapply(g)
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/

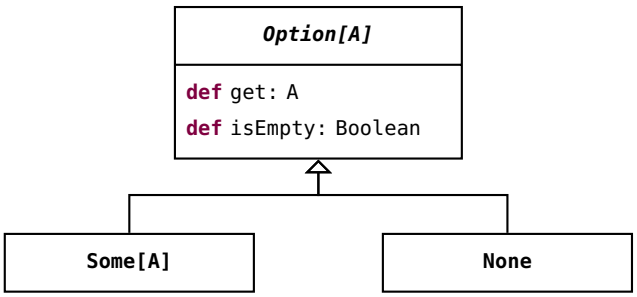
pattern-matching.html

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




```
scala> var x: Option[Int] = Some(42)

scala> x.isEmpty
val res0: Boolean = false

scala> x = None

scala> x.isEmpty
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])
```

```
scala> val p1 = Person("Björn",  Some(Gender.Male))
scala> val p2 = Person("Sandra", Some(Gender.Female))
scala> val p3 = Person("Kim",  None)
scala> val g2 = p2.gender
scala> def show(g: Option[Gender]): String = g match {
         case Some(x) => x.toString
         case None    => "unknown"
       }
scala> show(g2)
scala> show(p3.gender)
scala> val ps = Vector(p1,p2,p3)
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

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

scala> xs.headOption
res0: ???

scala> ys.headOption
res1: ???

scala> xs.find(_ > 1)
res2: ???

scala> xs.find(_ > 5)
res3: ???
```

```

scala> val huvudstad = Map("Sverige" -> "Sthlm", "Skåne" -> "Malmö")

scala> huvudstad.get("Skåne")
res4: ???

scala> huvudstad.get("Danmark")
res5: ???
```

6.1.31 Några samlingsmetoder som ger en `Option`, svar

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

scala> xs.headOption
res0: Option[Int] = Some(1)

scala> ys.headOption
res1: Option[Nothing] = None

scala> xs.find(_ > 1)
res2: Option[Int] = Some(2)

scala> xs.find(_ > 5)
res3: Option[Int] = None

scala> val huvudstad = Map("Sverige" -> "Sthlm", "Skåne" -> "Malmö")

scala> huvudstad.get("Skåne")
res4: Option[String] = Some(Malmö)

scala> huvudstad.get("Danmark")
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

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**.

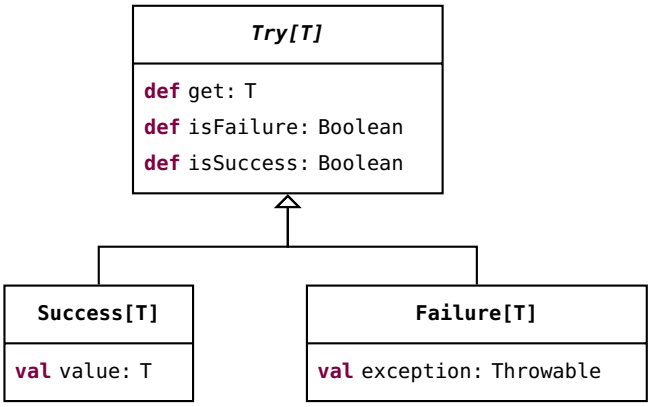
```
scala> def pang = throw Exception("PANG!")
pang: Nothing

scala> pang
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 bastyp 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
```

```
scala> Vector.fill(5)(carola)
Fångad av en stormvind
Fångad av en stormvind
Fångad av en stormvind
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 - t

scala> def min(op: => Unit, n: Int = 1000): Long = Seq.fill(n)(time(op)).drop(n / 20).m

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 subklasser. 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 subklasser 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 subklass 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/pins1ed/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-equality.html

---

6.2 Övning `patterns`

Mål

- ☐ Kunna skapa och använda **`match`**-uttryck med konstanta värden, garder 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 och ge som argument din favoritgrönsak. Vad händer? Förklara hur ett **`match`**-uttryck fungerar.

```
object Match:
  def main(args: Array[String]): Unit =
    val favorite = if args.length > 0 then args(0) else "selleri"
    println("Din favoritgrönsak: " + favorite)
    val firstChar = favorite.toLowerCase.charAt(0)
    val meThink = firstChar match
      case 'g' => "gurka är gott!"
      case 't' => "tomat är gott!"
      case 'b' => "broccoli är gott!"
      case  _  =>  s"$favorite är mindre gott..."
    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 deklareras explicit). Detta kallas för **mönstermatchning**. Vad skrivs ut nedan? Varför? Prova att byta namn på `v` och `rutten`.

```
scala> case class Gurka(vikt: Int, ärRutten: Boolean)
scala> val g = Gurka(100, true)
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.

```
scala> sealed trait Färg
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 nås enkelt.

```
scala> parafärg(Spader)
scala> val xs = Vector.fill(5)(Färg.values((math.random() * 4).toInt))
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 paralellfärgen till argumentet som ges. Tänk på att denna gången är `Färg` inget `sealed trait`, utan istället en enumeration (`enum`). Klistra in funktionen i REPL.

```
scala> parafärg(Färg.Ruter)
scala> val xs = Vector.fill(5)(Färg.values((math.random() * 4).toInt))
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 deklareras 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?

```
scala> val x = "urka"
scala> x match
         case str if str.startsWith("g") => println("kanske gurka")
         case vadsomhelst => println("ej gurka: " + vadsomhelst)
scala> val g = "gurka"
scala> g match
         case str if str.startsWith("g") => println("kanske gurka")
         case vadsomhelst => println("ej gurka: " + vadsomhelst)
```

b) *Fallgrop med små begynnelsbokstäver.* Innan du provar nedan i REPL, försök gissa vad som kommer att hända. Vad händer? Hur lyder varningarna och vad innebär de?

```
scala> val any: Any = "varken tomat eller gurka"
scala> case object Gurka
scala> case object tomat
scala> any match
         case Gurka => println("gurka")
         case tomat => println("tomat")
         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 tangentbordstryck.[2] 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.

```
scala> val xss = Vector(Vector("hej"),Vector("på", "dej"),Vector("4","x","2"))
scala> xss.map( _ match
  case Vector() => "tom"
  case Vector(a) => a.reverse
  case Vector(_, b) => b.reverse
  case Seq(a, "x", b) => a + b
  case _ => "ANNARS DETTA"
  ).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[3] 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.

```
scala> var kanske: Option[Int] = None
scala> kanske.size
scala> kanske = Some(42)
scala> kanske.size
scala> kanske.isEmpty
scala> kanske.isDefined
scala> def ökaOmFinns(opt: Option[Int]): Option[Int] = opt match
         case Some(i) => Some(i + 1)
         case None    => None
scala> val annanKanske = ökaOmFinns(kanske)
scala> def öka(i: Int) = i + 1
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.

```
val meningen = Some(42)
val ejMeningen = Option.empty[Int]
meningen.map(_ + 1)
```

[2] Fråga någon om du inte hittar hur man gör backtick ` på ditt tangentbord.
[3] Scala har också `null` men det behövs bara vid samverkan med Java-kod.

```
ejMeningen.map(_ + 1)
ejMeningen.map(_ + 1).orElse(Some("saknas")).foreach(println)
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?

```
val xs = (42 to 84 by 5).toVector
val e = Vector.empty[Int]
xs.headOption
xs.headOption.get
xs.headOption.getOrElse(0)
xs.headOption.orElse(Some(0))
e.headOption
e.headOption.get
e.headOption.getOrElse(0)
e.headOption.orElse(Some(0))
Vector(xs, e, e, e)
Vector(xs, e, e, e).map(_.lastOption)
Vector(xs, e, e, e).map(_.lastOption).flatten
xs.lift(0)
xs.lift(1000)
e.lift(1000).getOrElse(0)
xs.find(_ > 50)
xs.find(_ < 42)
e.find(_ > 42).foreach(_ => println("HITTAT!"))
```

d) Vilka är fördelerna 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 situtation 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?

```
scala> throw new Exception("PANG!")
scala> java.lang.   // Tryck TAB efter punkten
scala> throw new IllegalArgumentException("fel fel fel")
scala> val carola =
         try
           throw new Exception("stormvind!")
           42
         catch
           case e: Throwable =>
             println("Fångad av en " + e)
             -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 undantantag 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.

```
scala> def pang = throw new Exception("PANG!")
scala> import scala.util.{Try, Success, Failure}
scala> Try{pang}
scala> Try{pang}.recover{case e: Throwable =>   "desarmerad bomb: " + e}
scala> Try{"tyst"}.recover{case e: Throwable => "desarmerad bomb: " + e}
scala> def kanskePang = if math.random() > 0.5 then "tyst" else pang
scala> def kanskeOk = Try{kanskePang}
scala> val xs = Vector.fill(100)(kanskeOk)
scala> xs(13) match
         case Success(x) => ":)"
         case Failure(e) => ":( " + e
scala> xs(13).isSuccess
scala> xs(13).isFailure
scala> xs.count(_.isFailure)
scala> xs.find(_.isFailure)
scala> val badOpt = xs.find(_.isFailure)
scala> val goodOpt = xs.find(_.isSuccess)
scala> badOpt
scala> badOpt.get
scala> badOpt.get.get
scala> badOpt.map(_.getOrElse("bomben desarmerad!")).get
scala> goodOpt.map(_.getOrElse("bomben desarmerad!")).get
scala> xs.map(_.getOrElse("bomben desarmerad!")).foreach(println)
scala> xs.map(_.toOption)
scala> xs.map(_.toOption).flatten
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 innehållslikhet 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 innehållslikhet 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 innehållslikhet?

```
scala> val g1 = new Gurka(42, true)
scala> val g2 = g1
scala> val g3 = new Gurka(42, true)
scala> g1 == g2
scala> g1 == g3
```

e) Hur märker man ovan att den överskuggade `equals` medför att == nu ger innehållslikhet? 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 innehållslikhet och hash-koder. I Scala får man ett objekts hash-kod med metoden ##.[4]

Uppgift 13. *Polynom.* Med hjälp av koden nedan, kan man göra följande:

```
scala> import polynomial.*

scala> Const(1) * x
res0: polynomial.Term = x

scala> (x*5)^2
res1: polynomial.Prod = 25x^2

scala> Poly(x*(-5), y^4, (z^2)*3)
res2: polynomial.Poly = -5x + y^4 + 3z^2
```

a) Förklara vad som händer ovan genom att studera koden nedan[5].

```
object polynomial:

  sealed trait Term:
    def *(that: Term): Term

  case class Const(value: BigDecimal) extends Term:

    def toSilentString: String = this match
      case Const.One      => ""
      case Const.MinusOne => "-"
      case _              => value.toString

    override def toString = value.toString

    override def *(that: Term): Term = that match
      case Const(d)    => Const(d * value)
      case v: Var      => Prod(this, Set(v))
      case Prod(c, vs) => Prod(Const(c.value * value), vs)

    def *(d: BigDecimal): Const = Const(d * value)

    def ^(e: Int): Const = Const(value.pow(e))


  object Const:
    final val Zero     = Const(BigDecimal(0))
    final val One      = Const(BigDecimal(1))
    final val MinusOne = Const(BigDecimal(-1))

```

[4] Om du är nyfiken på hash-koder, läs mer här: en.wikipedia.org/wiki/Hash_function

[5] Koden finns även här:
github.com/lunduniversity/introprog/tree/master/compendium/examples/polynomial

```
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
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 227 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 227 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**[6] 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`

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 = ???
```

[6] Detta recept bygger på http://www.artima.com/pins1ed/object-equality.html

Uppgift 20. *Speciella matchningar.* Läs om användning av speciella matchningar här:
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
Skapa ditt eget extraktor-objekt för http-addresser 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 227.
Läs mer om polynomdivision här: sv.wikipedia.org/wiki/Polynomdivision

6.3 Laboration: `blockbattle1`

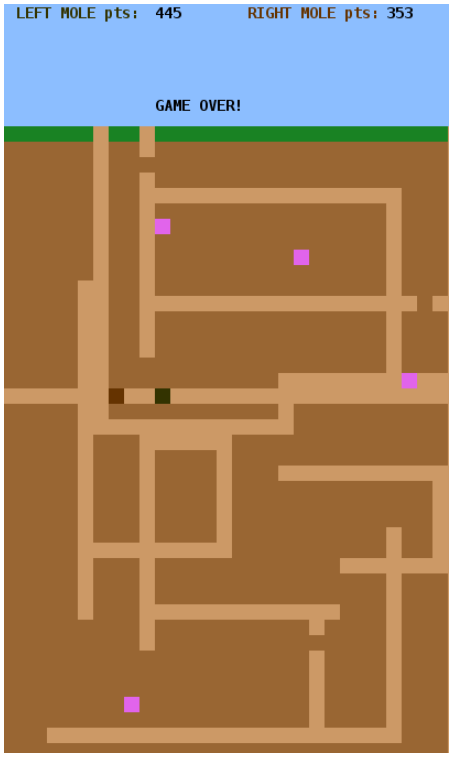
Mål

- ☐ 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. 158).
- ☐ 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 grävhastighet 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 spelkonstanter 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.[7]

```
window.setBlock(mole.nextPos, mole.color) // draw new
window.setBlock(mole.pos, Color.tunnel)   // erase old
mole.move()                               // update
```

[7] 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 dubbletter:

```
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? Dubbletter?

```
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)`

```
scala> xs.+:(42)
res4: scala.collection.immutable.Vector[Int] = Vector(42, 1, 2, 3)

```

- Konkatenering (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
  - **Transformering**: 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

`xs.map(f)`	transformering, motsv. `for x <- xs yield f(x)`
`xs.map(x => x)`	kopiering, motsv. `for x <- xs yield x`
`xs.filter(p)`	filtrering, ta med x om p(x)
`xs.filterNot(p)`	filtrering, ta med x om !p(x)
`xs.distinct`	filtrering, ta bort dubbletter
`xs.take(n)`	ny sekvens med de första n elements, resten skippade
`xs.drop(n)`	ny sekvens där de första n elements är skippade
`xs.takeWhile(p)`	filtrera, ta med i början så länge p(x)
`xs.dropWhile(p)`	filtrera, skippa i början så länge p(x)
`xs.find(p)`	sök framifrån efter första element x där p(x) är sant
`xs.indexOf(x)`	sök framifrån efter index för element som är samma som x
`xs.lastIndexOf(x)`	sök bakifrån efter index för element som är samma som x
`xs.sorted`	sortera med inbyggd (implicit given) ordning
`xs.sorted.reverse`	sortera i omvänd ordning
`xs.sortBy(f)`	sortera i ordning enligt f(x)
`xs.sortWith(lt)`	sortera enligt "less than"-funktionen `lt: (A, A) => Boolean`
`xs.count(p)`	räkna antalet element där p(x) 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*)
- borttagna (eng. *removed*)
- tillagda (eng. *inserted*)

.. med nya element ur `ys`

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

scala> xs.patch(2, Vector(-1), 1)     // replaced one elem
res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, -1)

scala> xs.patch(1, Vector(42), 0)     // inserted one elem
res11: scala.collection.immutable.Vector[Int] = Vector(1, 42, 2, 3)

```

```
scala> xs.patch(0, Vector(), 2)       // removed two elems
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 gard 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

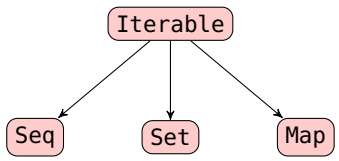
```
def copy(xs: Vector[Int]): Vector[Int] = for x <- xs yield x

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 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` där flera är **automatiskt** importerade
`scala.collection.mutable` som **måste importeras** explicit
(undantag: primitiva `scala.Array`)

7.1.19 Lämna det öppet: använd `Seq`

Typen **`collection.immutable.Seq`** är supertyp till alla sekvenssamlingar 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
```

```
scala> val xs = Vector(1,2,3)
xs: Vector[Int] = Vector(1, 2, 3)

scala> val ys = copyIntVector(xs)
ys: Vector[Int] = Vector(1, 2, 3)

scala> val zs = copyIntSeq(xs)
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`**:

```
scala> import scala.jdk.CollectionConverters.*

scala> Vector(1,2,3).asJava
res0: java.util.List[Int] = [1, 2, 3]

scala> val xs = new java.util.ArrayList[String]()
xs: java.util.ArrayList[String] = []

scala> xs.add("hej")
res1: Boolean = true

scala> xs.asScala
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.

```
scala> def fyll[T](n: Int, x: T): Array[T] = Array.fill(n)(x)
-- Error:
1 |def fyll[T](n: Int, x: T): Array[T] = Array.fill(n)(x)
  |                                                     ^
  |  No ClassTag available for T

scala> def fyll[T: reflect.ClassTag](n: Int, x: T): Array[T] = Array.fill(n)(x)

scala> fyll(42, "hej")
res2: Array[String] = Array(hej, hej, hej, hej, hej, hej, hej, hej, hej, hej, hej
```

- 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 uppräkning 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:

```
scala> Suit.Heart              // alla element är av typen Suit
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 slumptal 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 $result \leftarrow$ en ny array med plats för $xs.length$ element
2 $i \leftarrow 0$
3 **while** $i < xs.length$ **do**
4 $\quad result(i) \leftarrow xs(i)$
5 $\quad i \leftarrow i + 1$
6 **end**
7 $result$

7.1.36 Implementation av SEQ-COPY med `while`

```
object seqCopy:

  def arrayCopy(xs: Array[Int]): Array[Int] =
    val result = new Array[Int](xs.length)
    var i = 0
    while i < xs.length do
      result(i) = xs(i)
      i += 1
    result

  def test: String =
    val xs = Array(1,2,3,4,42)
    val ys = arrayCopy(xs)
    if xs sameElements ys then "OK!" else "ERROR!"

  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:

```
scala> type Pt = (Int, Int)            // typalias
scala> type Pts = Vector[Pt]           // nästlad typalias

scala> def distToOrigo(pt: Pt): Double = math.hypot(pt._1, pt._2)

scala> val xs: Pts = Vector((1,1), (2,2), (3,4))
val xs: Pts = Vector((1,1), (2,2), (3,4))

scala> xs.head
val res0: Pt = (1,1)

scala> xs.map(distToOrigo)
```

```
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`

Utdata: En kopia av pts men där pt är infogat på plats pos

$result \leftarrow$ en ny `Array[Pt]` med plats för $pts.length + 1$ element
for $i \leftarrow 0$ **to** $pos - 1$ **do**
  $result(i) \leftarrow pts(i)$
end
$result(pos) \leftarrow pt$
for $i \leftarrow pos + 1$ **to** $xs.length$ **do**
  $result(i) \leftarrow pts(i - 1)$
end
$result$

Övning: Skriv pseudo-kod för SEQ-REMOVE-COPY

7.1.40 Insättning/borttagning i kopia av primitiv Array

```
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] =
    val result = new Array[Pt](pts.length + 1)  // initialized with null
    for i <- 0 until pos do result(i) = pts(i)
    result(pos) = pt
    for i <- pos + 1 to pts.length do result(i) = pts(i - 1)
    result

  def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] =
    if pts.length > 0 then
      val result = new Array[Pt](pts.length - 1) // initialized with null
      for i <- 0 until pos do result(i) = pts(i)
      for i <- pos + 1 until pts.length do result(i - 1) = pts(i)
      result
    else Array.empty

  // ovan metoder implementerade med hjälp av den kraftfulla metoden patch:

  def insertCopy(pts: Array[Pt], pos: Int, pt: Pt) = pts.patch(pos, Array(pt), 0)

  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.

```
class PolygonWindow(width: Int, height: Int):
  val w = new introprog.PixelWindow(width, height, title = "PolygonWindow")

  def draw(pts: Seq[(Int, Int)]): Unit =
    if pts.size > 0 then
      for i <- 1 until pts.size do
        w.line(pts(i - 1)._1, pts(i - 1)._2, pts(i)._1, pts(i)._2)
      val last = pts.length - 1
      w.line(pts(last)._1, pts(last)._2, pts(0)._1, pts(0)._2)
```

```
object PolygonTest:
  val star =  Array((100,180), (150,100), (180,180), (90,130), (200, 130))
  val pw = new PolygonWindow(400,400)
  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

```
class PolygonArray(val maxSize: Int):
  type Pt = (Int, Int)
  private val points = new Array[Pt](maxSize)  // initialized with null
  private var n = 0
  def size = n

  def draw(w: PolygonWindow): Unit = w.draw(points.take(n).toSeq)

  def append(pts: Pt*): Unit =
    for i <- pts.indices do points(n + i) = pts(i)
    n += pts.length

  def insert(pos: Int, pt: Pt): Unit = // exercise: change pt to varargs pts
    for i <- n until pos by -1 do points(i) = points(i - 1)
    points(pos) = pt
    n += 1

  def remove(pos: Int): Unit = // exercise: change pos to fromPos, replaced
    for i <- pos until n do points(i) = points(i + 1)
    n -= 1

  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

```
class PolygonVector:
  type Pt = (Int, Int)
  private var points = Vector.empty[Pt]  // note var declaration to allow mutation
  def size = points.size

  def draw(w: PolygonWindow): Unit = w.draw(points.take(size))

  def append(pts: Pt*): Unit =
    points ++= pts.toVector

  def insert(pos: Int, pt: Pt): Unit = // exercise: change pt to varargs pts
    points = points.patch(pos, Vector(pt), 0)

  def remove(pos: Int): Unit = // exercise: change pos to fromPos, replaced
    points = points.patch(pos, Vector(), 1)

  override def toString = points.mkString("PrimitivePolygon(",",",")")
```

7.1.45 Exempel: Polygon som oföränderlig case class

```
object Polygon:
  type Pt = (Int, Int)
  type Pts = Vector[Pt]
  def apply(pts: Pt*) = new Polygon(pts.toVector)

case class Polygon(points: Polygon.Pts):
  import Polygon.Pt

  def size = points.size // for convenience but not really necessary (why?)

  def append(pts: Pt*): Polygon = copy(points ++ pts.toVector)

  def insert(pos: Int, pts: Pt*): Polygon = copy(points.patch(pos, pts, 0))

  def remove(pos: Int, replaced: Int = 1): Polygon =
    copy(points.patch(pos, Seq(), replaced))

  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:

```
scala> Vector("gurka","tomat","broccoli").indexOf("tomat")
res0: Int = 1

scala> Vector("gurka","tomat","broccoli").indexWhere(_.contains("o"))
res1: Int = 1

scala> Vector("gurka","tomat","broccoli").find(_.contains("o"))
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 problement:

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 **lösningsidé**: 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
- 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
```

```
scala> val xs = Vector(1,2,1,2,12,42,1)

scala> sortWith(xs)(_ < _)
val res0: Vector[Int] = Vector(1, 1, 1, 2, 2, 12, 42)

scala> sortWith(xs)(_ > _)
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
```

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".

`x +: xs`	1	A	`true`
`xs +: x`	2	B	`Vector(2, 2, 3)`
`xs :+ x`	3	C	`1`
`xs ++ xs`	4	D	`error: value tail is not a member of Int`
`xs.indices`	5	E	`(0 until 3)`
`xs apply 0`	6	F	`Vector(1, 2, 3)`
`xs(3)`	7	G	`Vector(0, 1, 2, 3)`
`xs.length`	8	H	`false`
`xs.take(4)`	9	I	`java.lang.IndexOutOfBoundsException`
`xs.drop(2)`	10	J	`Vector(1, 2, 3, 0)`
`xs.updated(0, 2)`	11	K	`Vector(3)`
`xs.tail.head`	12	L	`error: value +: is not a member of Int`
`xs.head.tail`	13	M	`Vector(1, 2, 3, 1, 2, 3)`
`xs.isEmpty`	14	N	`2`
`xs.nonEmpty`	15	O	`3`

b) Vid tre tillfällen blir det fel. Varför? Är det kompileringsfel 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:

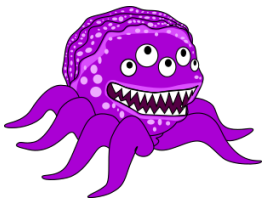
- snabbreferensen (enda tentahjälpmedel):
  http://cs.lth.se/pgk/quickref
- översikt (av Prof. Martin Odersky, uppfinnare av Scala, m.fl.):

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.[1]

```
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/dpdOUEe9mm4

```
scala> val fem = new Mutant(5)
scala> val xs = Vector(fem, fem, fem)
scala> val ys = xs.toArray    // kopierar referenserna till ny Array
scala> val zs = xs.map(x => new Mutant(x.int)) // djupkopierar till ny Vector
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).

`xs(0)`	
`ys(0).int`	
`zs(0).int`	
`xs(0) eq ys(0)`	
`xs(0) eq zs(0)`	
`(ys.toBuffer :+ new Mutant).apply(0).int`	

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:

[1] 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 $i \leftarrow 0$
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:

```
scala> Vector(1,2,3,4).patch(from = 1, other = Vector(0,0), replaced = 3)
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 loopvariabeluppräkning?
>
> 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.[2]

a) Vad ger följande jämförelser för värde?

```
scala> 'a' < 'b'
scala> "aaa" < "aaaa"
scala> "aaa" < "bbb"
scala> "AAA" < "aaa"
scala> "ÄÄÄ" < "ÖÖÖ"
scala> "ÅÅÅ" < "ÄÄÄ"
```

Tyvärr så följer ordningen av ÄÅÖ 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 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.

[2] Överkurs: Alla tecken i en `java.lang.String` representeras enligt UTF-16-standarden (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.

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.

`xs.indexOf(0)`	1	A	`Vector(1, 1)`
`xs.indexOf(6)`	2	B	`-1`
`xs.indexWhere(_ < 2)`	3	C	`true`
`xs.indexWhere(_ != 5)`	4	D	`Some(1)`
`xs.find(_ == 1)`	5	E	`Vector(1, 0, 1)`
`xs.find(_ == 6)`	6	F	`5`
`xs.contains(0)`	7	G	`Vector(4, 6)`
`xs.filter(_ == 1)`	8	H	`4`
`xs.filterNot(_ > 1)`	9	I	`1`
`xs.zipWithIndex.filter(_._1 == 1).map(_._2)`	10	J	`None`

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:

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

scala> indexOf(xs, _.contains('p'))
val res0: Int = 1

scala> indexOf(xs, _.contains('q'))
val res1: Int = -1

scala> indexOf(Vector(), _.contains('q'))
val res2: Int = -1

scala> indexOf(Vector("q"), _.length == 1)
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` håller 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.[3]
Läs om sorteringsmetoderna i snabbreferensen och prova i REPL.

[3] Ordningen kommer ursprungligen från föråldrade teckenkodningsstandarder: https://sv.wikipedia.org/wiki/ASCII

`'a' < 'A'`	1	A	`"ka"`
`"AÄÖö" < "AÅÖö"`	2	B	`1`
`xs.sorted.head`	3	C	`-1`
`xs.sorted.reverse.head`	4	D	`error: ...`
`ys.sorted.head`	5	E	`false`
`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	`true`
`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 inbygg 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 deklarera 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 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 checklistan 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.

```
scala> val fel = Vector("ö","å","ä","z").sorted
scala> val svColl = java.text.Collator.getInstance(new java.util.Locale("sv"))
scala> val svOrd = Ordering.comparatorToOrdering(svColl)
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

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 [4] 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.

[4] sv.wikipedia.org/wiki/Fibonaccital

Indata : En array xs med tecken
Utdata : xs uppdaterat på plats, med tecknen i omvänd ordning

```
n ← antalet element i xs
for i ← 0 to n/2 − 1 do
    temp ← xs(i)
    xs(i) ← xs(n − i − 1)
    xs(n − i − 1) ← temp
end
```

Uppgift 23. *Palindrompredikat.* En palindrom[5] 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
```

a) Implementationen ovan kan innebära att alla tecken i strängen gås igenom två gånger och behöver minnesutrymme för dubbla antalet tecken. Varför?

b) Skapa ett palindromtest som går igenom elementen max en gång och som inte behöver extra minnesutrymme för en kopia av strängen. *Lösningsidé:* 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
```

a) Hur fungerar likhetstest mellan två `ArraySeqs`? Vad har `xs` ovan för typ? Går det att uppdatera en wrappad array?

b) Vilken typ av argumentsekvens får du tillgång till i kroppen för en funktion med repeterande parametrar?

[5] 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 innehållslikhetstest.

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? [6]

```
> 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
```

[6] 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: 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
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
Läs även kritiken av förenklingar i ovan beskrivning här:
www.lihaoyi.com/post/ScalaVectoroperationsarentEffectivelyConstanttime.html
Läs denna grundliga empirisk genomgång av prestanda i Scalas samlingsbibliotek:
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[7] (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

```
for i ← (n − 1) to 0 do
    dra slumptal r så att 0 <= r <= i
    byt plats på xs(i) och xs(r)
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”[8]. 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 med dina lösningar. Med hjälp av klasserna `TestHand` och `TestDeck` kan du testa så att dina implementationer fungerar.

[7] https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_modern_algorithm
[8] https://sv.wikipedia.org/wiki/Pokerhand

```scala
package cards

case class Card(rank: Int, suit: Int):
  import Card._

  require(rankRange.contains(rank), s"rank=$rank, must be in $rankRange")
  require(suitRange.contains(suit), s"suit=$suit, must be in $suitRange")

  val rankString: String = ranks(rank - 1)
  val suitChar:   Char   = suits(suit - 1)

  override def toString() = s"$rankString$suitChar "

object Card:
  val suitRange: Range = 1 to 4
  val rankRange: Range = 1 to 13
  val suits: Vector[Char] = "♠♡♣♢".toVector
  val ranks: Vector[String] =
    "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`.[9]

- `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 testar klassen `Hand`.

[9] Du kan bläddra bland klasserna i paketet cards här:
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.

```
package cards

case class Hand(cards: Vector[Card]):
  import Hand._

  /**
   * A vector of length 14 with positions 1-13 containing the number of
   * cards of that rank. Position 0 contains 0.
   */
  def tally: Vector[Int] = ???

  def ranksSorted: Vector[Int] = cards.map(_.rank).sorted.toVector

  def isFlush: Boolean = cards.length > 0 && cards.forall(_.suit == cards(0).suit)

  def isStraight: Boolean =
    def isInSeq(xs: Vector[Int]): Boolean =
      xs.length > 1 && (0 to xs.length - 2).forall(i => xs(i) == xs(i + 1) - 1)

    isInSeq(ranksSorted) ||  // special case with ace interpreted as 14:
      (ranksSorted(0) == 1) && isInSeq(ranksSorted.drop(1) :+ 14)

  def isStraightFlush: Boolean = isStraight && isFlush
  def isFour:          Boolean = tally.contains(4)
  def isFullHouse:     Boolean = tally.contains(3) && tally.contains(2)
  def isThrees:        Boolean = tally.contains(3)
  def isTwoPair:       Boolean = tally.count(_ == 2) == 2
  def isOnePair:       Boolean = tally.contains(2)

  def category: Int = // TODO: add more tests when tally is implemented
    if isStraight && isFlush then Category.StraightFlush
    else if isFlush          then Category.Flush
    else if isStraight       then Category.Straight
    else                          Category.HighCard

object Hand:
  def apply(cardSeq: Card*): Hand = new Hand(cardSeq.toVector)
  def from(deck: Deck): Hand = new Hand(deck.peek(5))
  def removeFrom(deck: Deck): Hand = new Hand(deck.remove(5))

  object Category:
    val RoyalFlush = 0
    val StraightFlush = 1
    val Fours = 2
    val FullHouse = 3
    val Flush = 4
    val Straight = 5
    val Threes = 6
    val TwoPair = 7
    val OnePair = 8
    val HighCard = 9
    val values = RoyalFlush to HighCard

    object Name:
      val english = Vector("royal flush", "straight flush", "four of a kind", "full house",
        "flush", "straight", "three of a kind", "two pairs", "pair", "high card")
      val swedish = Vector("royal flush", "färgstege", "fyrtal", "kåk", "färg",
        "stege", "tretal", "två par", "par", "högt kort")
```

Figur 7.4: Den ofärdiga, oföränderliga klassen `Hand` som representerar en pokerhand.

```
package cards

object PokerProbability:
  /**
   * For a given number of iterations, shuffles a deck, draws a hand and
   * returns a vector with the frequency of each hand category.
   */
  def register(n: Int, deck: Deck): Vector[Int] = ???

  def main(args: Array[String]): Unit =
    val n = scala.io.StdIn.readLine("number of iterations: ").toInt
    val deck = Deck.full()
    val frequencies = register(n, deck)
    for i <- Hand.Category.values do
      val name = Hand.Category.Name.english(i).capitalize
      val percentages = frequencies(i).toDouble / n * 100
      println(f"$name%16s $percentages%10.6f%%")
```

Figur 7.5: Det ofärdiga singelobjektet `PokerProbability` som tar reda på sannolikheter för olika pokerhänder.

7.3.2 Obligatoriska uppgifter

Uppgift 1. Implementera algoritmen SHUFFLE.

a) 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`.

b) 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.

a) Implementera metoden `full` som skapar en 52-korts standardlek ordnad efter färg och valör. Använd `Range`-värdena i kompanjonsobjektet `Card`.

b) 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`.

a) 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.)

b) Testa klassen `Hand` med hjälp av `TestHand`.

Uppgift 4. Ta fram sannolikheterna för ”straight flush”, ”straight” och ”flush”.

a) 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`.

b) Kör `PokerProbability`, förslagsvis med en miljon iterationer. Du bör få ungefär

dessa sannolikheter[10]:

hand	*sannolikhet*
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...

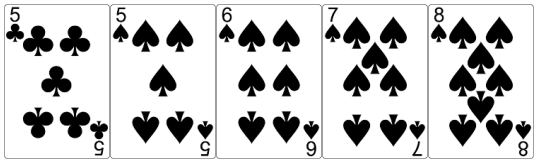
Uppgift 10. Implementera ett interaktivt kortspel, t.ex. någon enkel pokervariant. Börja med något mycket enkelt, till exempel högst-kort-vinner, och bygg vidare med sådant som du tycker verkar roligt.

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

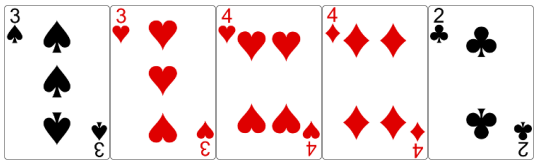
7.3.4 Bilder med exempel på olika pokerhänder

Figurerna 7.6 – 7.14 visar bilder på olika korthänder i poker.

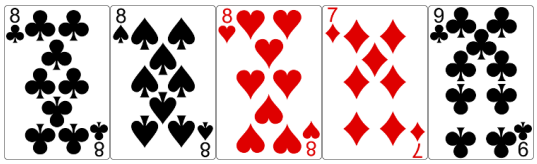
[10] http://www.forum.gpcdata.se/pdf/poker.pdf



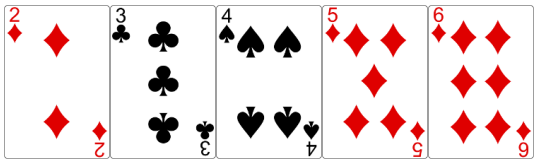
Figur 7.6: Par: två kort har samma valör.



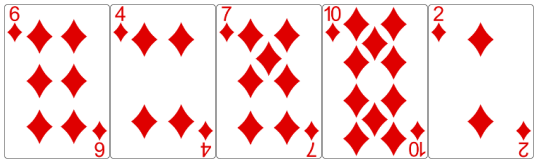
Figur 7.7: Två par: handen har två *olika* par.



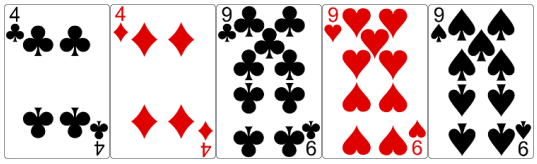
Figur 7.8: Triss: tre kort har samma valör.



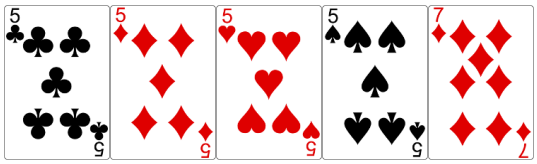
Figur 7.9: Stege: kortens valörer bildar en följd, ess kan vara antingen 1 eller 14.



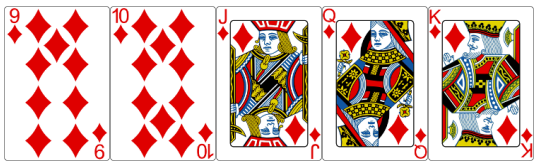
Figur 7.10: Färg: alla kort har samma färg.



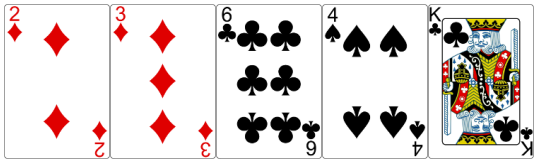
Figur 7.11: Kåk: både triss och par.



Figur 7.12: Fyrtal: fyra kort har samma valör.



Figur 7.13: Färgstege: både stege och färg.



Figur 7.14: Högt kort: inget mönster finns.

Del III

Lösningar

Kapitel 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 = gurkatomatgurkatomatgurkatomatgurkatomatgurkatomatgurkatomatgurk
```

Lösn. uppg. 4. *När upptäcks felet?*

a) Typ: `String`, värde: `"hejhejhej"`

b) Körtiddsfel:

```
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 överensstämma 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	$\rightsquigarrow$	E	`Int`
`1L`	2	$\rightsquigarrow$	G	`Long`
`1.0`	3	$\rightsquigarrow$	J	`Double`
`1D`	4	$\rightsquigarrow$	F	`Double`
`1F`	5	$\rightsquigarrow$	H	`Float`
`'1'`	6	$\rightsquigarrow$	I	`Char`
`"1"`	7	$\rightsquigarrow$	A	`String`
`true`	8	$\rightsquigarrow$	C	`Boolean`
`false`	9	$\rightsquigarrow$	B	`Boolean`
`()`	10	$\rightsquigarrow$	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 aritmetik med begränsat antal bitar fungerar i CPU:n.

```
scala> Int.MaxValue + 1
res3: Int = -2147483648

scala> Int.MinValue
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 $1.8 \cdot 10^{19}$

```
scala> math.pow(2, 64) - 1
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.

```
scala> omkrets(12750 / 2) / 80
res0: Double = 500.6913291658733
```

Lösn. uppg. 7. *Variabler och tilldelning. Förändringsbar och oföränderlig variabel.*

a)

Efter rad 1:	a: Int 13		
Efter rad 2:	a: Int 13	b: Int 14	
Efter rad 3:	a: Int 13	b: Int 14	c: Double 54.0
Efter rad 4:	a: Int 13	b: Int 14	c: Double 54.0
Efter rad 5:	a: Int 0	b: Int 14	c: Double 54.0
Efter rad 6:	a: Int 0	b: Int 14	c: Double 55.0

b) Oföränderliga variabler deklareras 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`. (Verktyget `scaladoc` kan med hjälp av dokumentationskommentarerna automatiskt generera webbsajter med speciella dokumentationssidor och sökfunktioner.)

b)

```
scala> def roll: Int = (math.random() * 6 + 1).toInt

scala> roll
res0: Int = 4

scala> roll
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 definera 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 utryck kan evalueras och resulterar då i ett användbart värde. En sats *gör* något (t.ex. skriver ut något), men resulterat 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)

```
scala> Double.MinPositiveValue
res0: Double = 4.9E-324
```

b)

```
scala> Double.MaxValue + Double.MinPositiveValue == Double.MaxValue
res2: Boolean = true
```

Lösn. uppg. 20. ***`if`****-sats.*

1. Utskrift: `falskt`
2. Utskrift: `sant`
3. Inget skrivs ut, funktionen deklareras men körs ej.
4. Utskrift: `1:krona 2:klave 3:krona 4:krona 5:klave` eller liknande beroende på vilka slumptal `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)

```
scala> def isEven(n: Int): Boolean = n % 2 == 0

scala> isEven(42)
res0: Boolean = true

scala> isEven(43)
res1: Boolean = false
```

b)

```
scala> def isOdd(n: Int): Boolean = !isEven(n)

scala> isOdd(42)
res2: Boolean = false

scala> isOdd(43)
res3: Boolean = true
```

Lösn. uppg. 23. *Skillnader mellan* `var`, `val`, `def`.

a)

```
  scala> var x = 30
  x: Int = 30

  scala> x + 1
  res6: Int = 31

  scala> x = x + 1
  x: Int = 31

  scala> x == x + 1
  res7: Boolean = false

  scala> val y = 20
  y: Int = 20

  scala> y = y + 1
  <console>:12: error: reassignment to val
         y = y + 1
           ^

  scala> var z = { println("hej z!"); math.random() }
  hej z!
  z: Double = 0.3381365875903367

  scala> def w = { println("hej w!"); math.random() }
  w: Double

  scala> z
  res8: Double = 0.3381365875903367

  scala> z
  res9: Double = 0.3381365875903367

  scala> z = z + 1
  z: Double = 1.3381365875903368

  scala> w
  hej w!
  res10: Double = 0.06420209879434557

  scala> w
  hej w!
  res11: Double = 0.5777951341051852
```

```
scala> w = w + 1
<console>:12: error: value w_= is not a member of object
       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	40		
Efter rad2:	a: Int	40	b: Int	80
Efter rad3:	a: Int	50	b: Int	80
Efter rad4:	a: Int	50	b: Int	70
Efter rad5:	a: Int	100	b: Int	70
Efter rad6:	a: Int	100	b: Int	35

Lösn. uppg. 28. *Stora tal.*

a) `BigInt` kan användas i stället för `Int` vid mycket stora heltal. Det finns förståss 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)

```
scala> BigInt(2).pow(64)
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) `45.00000000000001` (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.

```
scala> 1 % 2
res0: Int = 1

scala> -1 % 2
res1: Int = -1

scala> -1 % -2
res2: Int = -1

scala> 1 % -2
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)

```
scala> import Integer.{toBinaryString => toBin, toHexString => toHex}

scala> for i <- Seq(33, 42, 64) do println(s"$i \t ${toBin(i)} \t ${toHex(i)}")
33   100001    21
42   101010    2a
64   1000000   40
```

b) Det hexadecimala heltalet $10c$ kan anges med litteralen `0x10c` i Scala, Java och många andra språk: [1]

```
scala> 0x10c
res0: Int = 268
```

c) [2]

```
scala> val c = 299792458
c: Int = 299792458

scala> BigDecimal(0x10).pow(c)
res68: scala.math.BigDecimal = 2.1248929632279066130609861108876 72E+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: integer overflow`

```
scala> Math.multiplyExact(1, 2)
res70: Int = 2

scala> Int.MaxValue * 2
res71: Int = -2
```

[1] https://en.wikipedia.org/wiki/0x10c
[2] https://c418.bandcamp.com/album/0x10c

```
scala> Math.multiplyExact(Int.MaxValue, 2)
java.lang.ArithmeticException: integer overflow
  at java.lang.Math.multiplyExact(Math.java:867)
  ... 42 elided
```

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)

```
scala> Int.MaxValue *! 1
res0: Int = 2147483647

scala> Int.MaxValue *! 2
java.lang.ArithmeticException: integer overflow
  at java.lang.Math.multiplyExact(Math.java:867)
  at IntExtra.$times$bang(<console>:16)
  ... 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 *verkligen* 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
algoritm	14	⇝	M	stegvis beskrivning av en lösning på ett problem
implementation	15	⇝	D	en specifik realisering av en algoritm

Lösn. uppg. 2. *Använda terminalen.*

a)

```
> mkdir hello
> cd hello
> pwd
```

b)

```
> cd ..
> ls
```

Lösn. uppg. 3. *Skapa och köra ett Scala-skript.*

a)

```
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:

```
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.*

`val xs = Vector(2)`	1	⇝	I	ny referens till sekvens av längd 1
`val 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 **`true`** om alla element lika
Array	2	⇝	B	olikt andra Scala-samlingar kollar == ej innehållslikhet

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:

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

scala> val as = Array.fill(1e6.toInt)(0)
val as: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
large output truncated, print value to show all

scala> val vs = Vector.fill(1e6.toInt)(0)
val vs: Vector[Int] = Vector(0, 0, 0, 0, 0, ...
large output truncated, print value to show all

scala> val ast = (for i <- 1 to 10 yield time(as :+ 0)).sum / 10.0
val ast: Double = 1.8719819999999998

scala> val vst = (for i <- 1 to 10 yield time(vs :+ 0)).sum / 10.0
val vst: Double = 0.006485099999999999

scala> ast / vst
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:

```
public int plusxy(int, int);
  descriptor: (II)I
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=3
       0: iload_1
       1: iload_2
       2: iadd
       3: ireturn
    LineNumberTable:
      line 2: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       4     0  this   Lplusxy$package$;
          0       4     1     x   I
          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.

```
public int plusxyz(int, int, int);
  descriptor: (III)I
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=2, locals=4, args_size=4
       0: iload_1
       1: iload_2
       2: iadd
       3: iload_3
       4: iadd
       5: ireturn
    LineNumberTable:
      line 2: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       6     0  this   Lplusxyz$package$;
          0       6     1     x   I
          0       6     2     y   I
          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 indata 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)

```
Namn: Triangelsson, Stina
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 tomma värdet och `println` är en procedur. När tomma värdet skrivs ut visas `()`.

b) Procedurer returnerar tomma 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 fördröjnig innan 42 returneras. `42 + 42 == 84` vilket blir värdet av uttrycket.

```
scala> snark + snark
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.

```
callByValue(snark)
snark val res2: Int = 84
```

c) Evalueringen av uttrycket `snark` fördröjs tills varje förekomst av parametern `x` i funktionskroppen. Utskriften sker två gånger.

```
callByName(snark)
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.

```
callByName(zzz)
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.

```
scala> görDettaTvåGånger { println("goddag") }
goddag
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)

```
scala> :load Underjorden.scala
scala> import Underjorden.*
scala> Masken.ärMullvadsmat
val res0: Boolean = false
scala> Masken.y = Mullvaden.y
scala> Masken.ärMullvadsmat
```

```
val res1: Boolean = true
```

c)

```
scala> import Mullvaden.*
scala> import Masken.*
scala> x = -1
scala> Mullvaden.x
val res2: Int = -1

scala> Masken.x
1 |Masken.x
  |^^^^^^^^
  |variable x cannot be accessed as a member of Underjorden.Masken.type from mo

scala> Underjorden.x
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.

```
scala> z
nu!
val res19: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> z
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).

```
scala> lazy val z = { println("nu!"); Array.fill(1e9.toInt)(0)}
val z: Array[Int] = <lazy>

scala> z
nu!
java.lang.OutOfMemoryError: Java heap space
```

c) Utskriften av "nu!" sker först när singelobjektet zzz används för första gången. Vi borde lägga initialiseringen av b före a eller göra a till en **lazy val**.

d)

```
scala> import test.*
import test.*

scala> zzz.a      // först när vi använder zzz skrivs "nu!"
nu!               // detta skedde *inte* när vi importerade test
val res0: Int = 42

scala> buggig.a   // a blir 0 eftersom b inte är initialiserad
val res1: Int = 0

scala> funkar.a   // med lazy val unviker vi problemet
val res2: Int = 42


scala> zzz.a     // andra gången är init redan gjort och ingen "nu!"
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)

```
> scala repl --jar introprog_3-1.3.1.jar
scala> val w = new introprog.PixelWindow(400,300,"HEJ")
scala> w.line(100, 100, 200, 100)
scala> w.line(200, 100, 200, 200)
scala> w.line(200, 200, 100, 200)
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 hel
```

f)

```
> scala-cli run hello-window.scala --dep se.lth.cs::introprog:1.3.1 --main-clas
```

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:

```
lastEventType: 1 => KeyPressed
lastKey == Up
lastEventType: 2 => KeyReleased
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:

```
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))
```

```
scala> statistics(Vector(0, 2.5, 5))
val res10: (Int, Double, (Double, Double)) = (3,2.5,(0.0,5.0))
```

Lösn. uppg. 14. *Skapa moduler med hjälp av paket.*

a)

```
> code paket.scala
> scala-cli paket.scala
> find . -type d         # linuxkommando som listar alla subkataloger
./.scala-build/project_103be31561-3d0d386400/classes
./.scala-build/project_103be31561-3d0d386400/classes/main
./.scala-build/project_103be31561-3d0d386400/classes/main/gurka
./.scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat
./.scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan
./.scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p2
./.scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p2/
./.scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p1
./.scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p1/
./.scala-build/project_103be31561-3d0d386400/classes/main/gurka/tomat/banan/p1/
```

b)

```
> scala-cli run paket.scala --main-class gurka.tomat.banan.Main
Hej paket p1.p11!
Hej paket p1.p12!
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**

```
scala> introprog.Dialog.selectColor()
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)

```
scala> introprog.Dialog.show("Game over!")
```

b) Funktionen `input` returnerar en sträng som blir tomma strängen `""` om användaren klickar **Cancel**

```
scala> val name = introprog.Dialog.input("Vad heter du?")
name: String = Oddput Superkodare
```

c) Funktionen `select` returnerar en sträng med texten på knappen som användaren tryckte på.

```
scala> introprog.Dialog.select("Vad väljer du?",Vector("Sten","Sax","Påse"))
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)

```
> jar -c -v -f my.jar hello
> ls
> scala-cli repl --jar my.jar
scala> hello.Main.main(Array())
Hello package!
```

d)

```
> 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
`null`	12	⇝	G	ett värde som ej refererar till någon instans
`new`	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 object`
`val p  = new Singelpunkt`	3	⇝	C	`Not found: type`
`val p1 = new Punkt`	4	⇝	D	`p1: Punkt = Punkt@27a1a53c`
`val p2 = Punkt()`	5	⇝	F	`p2: Punkt = Punkt@51ab04bd`
`{ p1.x = 1; p2.x }`	6	⇝	E	`3`
`(new Punkt).y`	7	⇝	H	`2`
`{ val p: Punkt = null; p.x }`	8	⇝	A	`java.lang.NullPointerException`

b)

fel	*typ*	*förklaring*
`value is not a member of object`	kompileringsfel	det finns ingen instans med namnet `Punkt`. *Felmeddelandet syftar på att det i klassens autogenererade konstruktor-ombud saknas en variabel med namnet* `x`.
`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)

`val p1 = Point(1, 2)`	1	⇝ C	`p1: Point = Point@30ef773e`
`val p2 = Point()`	2	⇝ A	`missing argument for parameter`
`val 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)

fel	*typ*	*förklaring*
`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)

`val p1 = Point3D()`	1	⇝ C	`p1: Point3D = Point3D@2eb37eee`
`val 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 val`
`p2.y == 0`	5	⇝ A	`false`
`p1.x == Point3D().x`	6	⇝ D	`true`

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)

`val p1 = Pt(1, 2)`	1	⇝ E	`Pt(1,2)`
`val p2 = Pt(y = 3)`	2	⇝ C	`Pt(0,3)`
`val p3 = MutablePt(5, 6)`	3	⇝ A	`MPt(5,6)`
`val p4 = Mutable()`	4	⇝ D	`Not found`
`p2.moved(dx = 1) == Pt(1, 3)`	5	⇝ F	`true`
`p3.move(dy = 1) == MutablePt(5, 7)`	6	⇝ B	`false`

b) Kompilatorn härleder `MutablePt` eftersom det är typen på självreferensen this.

```
scala> :type new MutablePt().move()
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) // Koden 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*.

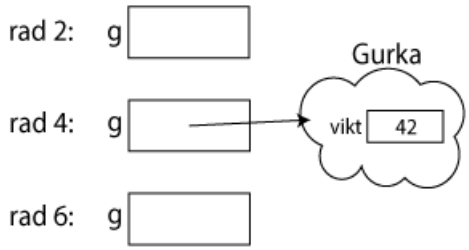
Lösn. uppg. 6. *Implementera delar av klasserna* `Pos`, `KeyControl`, `Mole` *och* `BlockWindow` *som behövs under laborationen* `blockbattle1`.Denna uppgift är laborationsförberedelse. Utvärdera dina lösningar genom egna tester i REPL.

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!!!

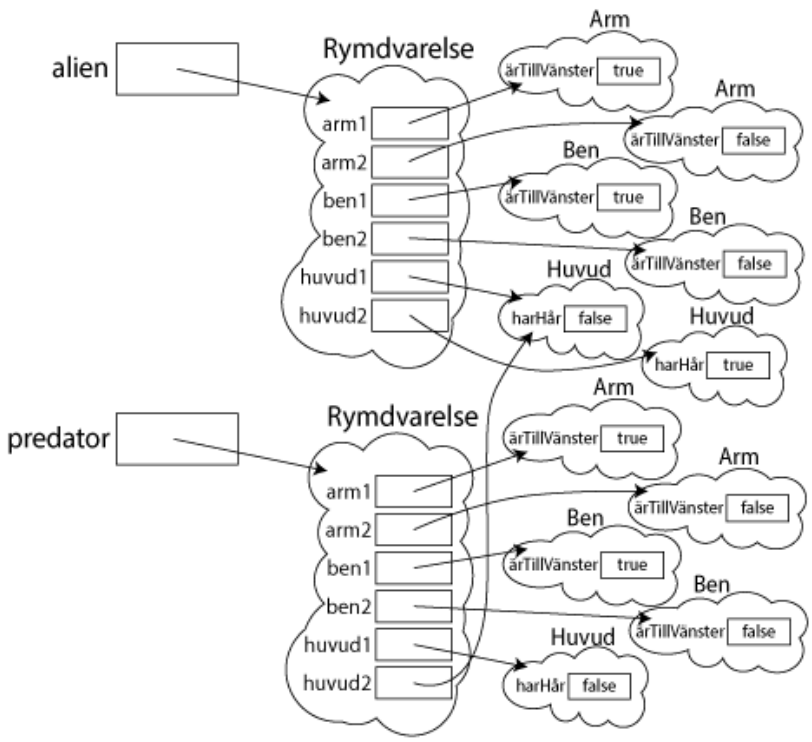
c) TODO!!!

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)

```
scala> val (s1, s2) = (Square(), Square(1, 10, 1))
val s1: Square = Square(0,0,1)
```

```
val s2: Square = Square(1,10,1)

scala> val s3 = s1 moved (1,-5)
val s3: Square = Square(1,-5,1)

scala> s1 isEqualSizeAs s3       // lika storlek
val res0: Boolean = true

scala> s2 isEqualSizeAs s1       // lika storlek
val res1: Boolean = true

scala> s1 isEqualSizeAs Square.unit   // s1 har sidan 1
val res2: Boolean = true

scala> s2.scale(math.Pi) isEqualSizeAs s2  // olika storlek
val res3: Boolean = false

scala> s2.scale(math.Pi) == s2.scale(math.Pi) // lika innehåll
val res4: Boolean = true

scala> s2.scale(math.Pi) eq s2.scale(math.Pi)  // olika objekt
val res5: Boolean = false

scala> Square.unit eq Square.unit   // samma objekt
val res6: Boolean = true
```

L.5.3 Fördjupningsuppgifter; utmaningar

Lösn. uppg. 11. *Innehållslikhet mellan olika typer.*

```
scala> 42 == "Fyrtiotvå"
1 |42 == "Fyrtiotvå"
  |^^^^^^^^^^^^^^^^^
  |Values of types Int and String cannot be compared with == or !=

scala> Gurka(50) == Bil("Sedan")
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 innehållslikhet. 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)
```

```
scala> Gurka(50) == Bil("Sedan")
1 |Gurka(50) == Bil("Sedan")
  |^^^^^^^^^^^^^^^^^^^^^^^^^
  |Values of types Gurka and Bil cannot be compared with == or !=
```

Lösn. uppg. 12. *Attributrepresentation. Privat konstruktor. Fabriksmetod.*

a) Det blir kompileringsfel eftersom konstruktorn är privat.

```
scala> 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()
     |
// defined class Point
// defined object Point

scala> new Point(0, 0)
1 |new Point(0, 0)
  |    ^^^^^
  |constructor Point cannot be accessed as a member of Point from module class
```

b)

- Genom att ha en privat konstruktor 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.

```
11 |  def kompisVikt = kompis.vikt
   |                   ^^^^^^^^^^^
   |value vikt cannot be accessed as a member of (Gurka5.this.kompis : Gurka5)
```

b)

```
scala> new Gurka1(42).vikt
1 |new Gurka1(42).vikt
  |^^^^^^^^^^^^^^^^^^^
  |value vikt cannot be accessed as a member of Gurka1 from module class

scala> new Gurka2(42).vikt
val res0: Int = 42

scala> new Gurka3(42).vikt
1 |new Gurka3(42).vikt
  |^^^^^^^^^^^^^^^^^^^
  |value vikt cannot be accessed as a member of Gurka3 from module class

scala> val ingenGurka: Gurka4 = null
val ingenGurka: Gurka4 = null

scala> new Gurka4(42, ingenGurka).kompisVikt
java.lang.NullPointerException: Cannot invoke "rs$line$1$Gurka4.vikt()" bec...
  at rs$line$1$Gurka4.kompisVikt(rs$line$1:8)
  ... 38 elided

scala> new Gurka4(42, new Gurka4(84, null)).kompisVikt
val res2: Int = 84

scala> new Gurka6(42)
1 |new Gurka6(42)
  |    ^^^^^^
  |constructor Gurka6 cannot be accessed as a member of Gurka6 from module...

scala> new Gurka7(-42)
1 |new Gurka7(-42)
  |    ^^^^^^
```

```
  |constructor Gurka7 cannot be accessed as a member of Gurka7 from module...

scala> Gurka7(-42)
java.lang.IllegalArgumentException: requirement failed: negativ vikt: -42

scala> val g = Gurka7(42)
val g: Gurka7 = Gurka7@51fd1c7c

scala> g.vikt
val res4: Int = 42

scala> g.vikt = -1

scala> g.vikt
val res5: Int = -1
```

Lösn. uppg. 14. *Egendefinierad setter kombinerat med privat konstruktor.*

a)

Rad 1:

```
  java.lang.IllegalArgumentException: requirement failed: negativ vikt: -42
```

`Gurka8.apply` kräver att `vikt >= 0` annars kastar `require` ett undantag.

Rad 5:

```
  java.lang.IllegalArgumentException: requirement failed: negativ vikt: -1
```

Settern `vikt_=` kräver att `vikt >= 0` annars kastar `require` ett undantag.

Rad 7:

```
  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)

```scala
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)

```
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 testar 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` testar 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)

```
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)

```
<console>:17: warning: match may not be exhaustive.
It would fail on the following input: Ruter
```

Varningen kommer redan vid kompilering.

e)

```
scala.MatchError: Ruter (of class Ruter)
  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` definerad. 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 definerad 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)

```
<console>:16: warning: patterns after a variable pattern cannot match (SLS 8.1
.1)
```

och

```
<console>:17: warning: unreachable code due to variable patter 'tomat' on line
16
```

Trots att en klass `tomat` existerar så tolkar Scalas **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.*

```
jeh
jed
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.
```
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.
Slutgiltligen 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 undantantag 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.

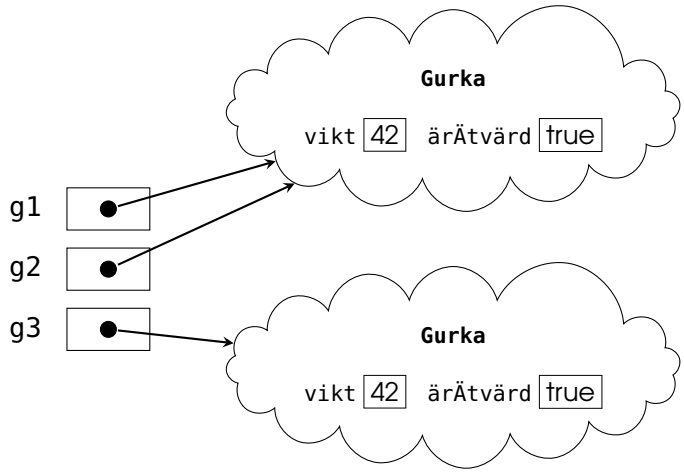
Lösn. uppg. 12. *Metoden* `equals`.

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\$1: 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 innehållslikhet, 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 innehållslikhet. Efter skuggningen ger dock detta uttryck `true` vilket påvisar jämförelse av innehållslikhet.

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 innehållslikhet 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
sekvensalgoritm	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	algoritm som ordnar element i en viss ordning
sökning	8	⇝	D	algoritm som letar upp element enligt sökkriterium
linjärsökning	9	⇝	F	sökalgoritm som letar i sekvens tills element hittas
registrering	10	⇝	H	algoritm 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)

`x +: xs`	1	⇝	G	`Vector(0, 1, 2, 3)`
`xs +: x`	2	⇝	L	`error: value +: is not a member of Int`
`xs :+ x`	3	⇝	J	`Vector(1, 2, 3, 0)`
`xs ++ xs`	4	⇝	M	`Vector(1, 2, 3, 1, 2, 3)`
`xs.indices`	5	⇝	E	`(0 until 3)`
`xs apply 0`	6	⇝	C	`1`
`xs(3)`	7	⇝	I	`java.lang.IndexOutOfBoundsException`
`xs.length`	8	⇝	O	`3`
`xs.take(4)`	9	⇝	F	`Vector(1, 2, 3)`
`xs.drop(2)`	10	⇝	K	`Vector(3)`
`xs.updated(0, 2)`	11	⇝	B	`Vector(2, 2, 3)`
`xs.tail.head`	12	⇝	N	`2`
`xs.head.tail`	13	⇝	D	`error: value tail is not a member of Int`
`xs.isEmpty`	14	⇝	H	`false`
`xs.nonEmpty`	15	⇝	A	`true`

b)

fel	*typ*	*förklaring*
`value +: is not a member of Int`	kompileringsfel	Operatorer som slutar med kolon är högerassociativa. Metodanropet `xs +: x` motsvarar med punktnotation `x.+:(xs)` och det finns ingen metod med namnet `+:` på heltal.
`IndexOutOfBoundsException`	körtidsfel	Det finns bara 3 element och index räknas från 0 i sekvenssamlingar.
`value tail is not a member of Int`	kompileringsfel	Metoden `head` ger första elementet och heltal saknar sekvenssamlingsmetoden `tail`.

Lösn. uppg. 4. *Kopiering av sekvenser.*

a)

`xs(0)`	`rs$line5$Mutant@66d766b9` nya instanser får nya hexkoder
`ys(0).int`	`0` eftersom `ys` innehåller samma instans som `xs`
`zs(0).int`	`5` eftersom `!(xs(0) eq zs(0))`
`xs(0) eq ys(0)`	`true` eftersom samma instans
`xs(0) eq zs(0)`	`false` eftersom olika instanser
`(ys.toBuffer :+ new Mutant).apply(0).int`	`0` eftersom den ej djupkopierade kopian av typen `ArrayBuffer` refererar samma instans på första platsen som både `ys` och `xs` och `x(0).int` 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) //kopia 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)

```
scala> class Mutant(var int: Int = 0)
// defined class Mutant

scala> def deepCopy(xs: Array[Mutant]): Array[Mutant] =
     |   val result = Array.ofDim[Mutant](xs.length)
     |   var i = 0
     |   while i < xs.length do
     |     result(i) = new Mutant(xs(i).int)
     |     i += 1
     |   result

scala> val xs = Array.fill(3)(new Mutant)
xs: Array[Mutant] = Array(rs$line$2$Mutant@46a123e4, rs$line$2$Mutant@44bc2449,
rs$line2$Mutant@3c28e5b6)

scala> val ys = deepCopy(xs)
ys: Array[Mutant] = Array(rs$line$2$Mutant@14b8a751, rs$line2$Mutant@7345f97d,
rs$line$2$Mutant@554566a8)

scala> xs(0).int = 5

scala> ys(0).int
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 säker 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
```

```
scala> insert(Array(1, 2), 0, pos = -1)
val res2: Array[Int] = Array(0, 1, 2)

scala> insert(Array(1, 2), 0, pos = 0)
val res3: Array[Int] = Array(0, 1, 2)

scala> insert(Array(1, 2), 0, pos = 1)
val res4: Array[Int] = Array(1, 0, 2)

scala> insert(Array(1, 2), 0, pos = 2)
val res5: Array[Int] = Array(1, 2, 0)

scala> insert(Array(1, 2), 0, pos = 42)
val res7: Array[Int] = Array(1, 2, 0)
```

Lösn. uppg. 6. *Jämföra strängar i Scala.*

a)

```
true
true
true
true
true
false
```

b) *s1* kommer först.

Lösn. uppg. 7. *Linjärsökning enligt olika sökkriterier.*

a)

`xs.indexOf(0)`	1	⇝	F	`5`
`xs.indexOf(6)`	2	⇝	B	`-1`
`xs.indexWhere(_ < 2)`	3	⇝	H	`4`
`xs.indexWhere(_ != 5)`	4	⇝	I	`1`
`xs.find(_ == 1)`	5	⇝	D	`Some(1)`
`xs.find(_ == 6)`	6	⇝	J	`None`
`xs.contains(0)`	7	⇝	C	**`true`**
`xs.filter(_ == 1)`	8	⇝	A	`Vector(1, 1)`
`xs.filterNot(_ > 1)`	9	⇝	E	`Vector(1, 0, 1)`
`xs.zipWithIndex.filter(_._1 == 1).map(_._2)`	10	⇝	G	`Vector(4, 6)`

b) Med en boolesk variabel `found`:

```scala
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`:

```scala
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:

```scala
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)

```
scala> registreraTärningskast(kasta(1000))
val res0: Vector[Int] = Vector(171, 163, 166, 152, 184, 164)

scala> registreraTärningskast(kasta(1000))
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`:

```
scala> ps.sorted
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 `sortWith`.

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`:

```
scala> import scala.util.Random

scala> val xs = Vector("Sten", "Sax", "Påse")
val xs: Vector[String] = Vector(Sten, Sax, Påse)

scala> (1 to 10).foreach(_ => println(Random.shuffle(xs).mkString(" ")))
Sax Påse Sten
Sten Påse Sax
Sten Sax Påse
Sten Sax Påse
Sten Påse Sax
Sten Påse Sax
Sax Sten Påse
Sten Påse Sax
Sax Påse Sten
Sax Påse Sten

scala> (1 to 5).map(_ => Random.shuffle(1 to 6))
val res1: IndexedSeq[IndexedSeq[Int]] =
  Vector(Vector(5, 2, 1, 4, 3, 6), Vector(6, 5, 4, 2, 1, 3),
  Vector(3, 1, 4, 6, 5, 2), Vector(3, 2, 6, 5, 1, 4),
  Vector(5, 3, 4, 6, 1, 2))

scala> (1 to 1000).map(_ => Random.shuffle(1 to 6).head).count(_ == 6)
val res2: Int = 168
```

Lösn. uppg. 11. *Repeterade parametrar.*

a)

```
scala> def stringSizes(xs: String*): Vector[Int] = xs.map(_.size).toVector
def stringSizes(xs: String*): Vector[Int]

scala> stringSizes("hej")
val res0: Vector[Int] = Vector(3)

scala> stringSizes("hej", "på", "dej", "")
val res1: Vector[Int] = Vector(3, 2, 3, 0)

scala> stringSizes()
val res2: Vector[Int] = Vector()
```

Anrop med tom argumentlista ger en tom heltalssekvens.

b)

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

```

```
scala> stringSizes(xs: _*)
val res0: Vector[Int] = Vector(3, 2, 3, 0)

scala> stringSizes(Vector(): _*)
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 uppräkningen 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 ← antalet element xs
2  ys ← en ny Array[Int] med plats för n − 1 element
3  for i ← 0 to pos − 1 do
4  |  ys(i) ← xs(i)
5  end
6  ys(pos) ← x
7  for i ← pos + 1 to n − 1 do
8  |  ys(i − 1) ← 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 ← antalet element xs
2  for i ← pos + 1 to n − 1 do
3  |  xs(i − 1) ← xs(i)
4  end
5  xs(n − 1) ← 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)

```
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)

```
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 $\quad xs(i) \leftarrow ys(i - 1)$
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 `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)

```
scala> fib(Int.MaxValue).size
val res0: Int = 46
```

c)

```
def fibBig(max: BigInt): List[BigInt] =
  val xs = scala.collection.mutable.ListBuffer.empty[BigInt]
  xs.prependAll(Vector(BigInt(1), BigInt(1)))
  while xs.head < max do xs.prepend(xs.take(2).sum)
  xs.reverse.drop(1).toList
```

```
scala> fibBig(Long.MaxValue).size
val res0: Int = 92

scala> fibBig(BigInt(Long.MaxValue).pow(64)).size
val res1: Int = 5809

scala> fibBig(BigInt(Long.MaxValue).pow(128)).last
val res2: BigInt = 46657280552835544919455361110286315395072000518604554717752

scala> fibBig(BigInt(Long.MaxValue).pow(128)).last.toString.size
val res3: Int = 2428

scala> fibBig(BigInt(Long.MaxValue).pow(256)).last.toString.size
val res4: Int = 4856

scala> fibBig(BigInt(Long.MaxValue).pow(1024)).last.toString.size
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.*

```
scala> val xs = Vector.tabulate(10)(i => math.pow(2, i).toInt)
xs: Vector[Int] = Vector(1, 2, 4, 8, 16, 32, 64, 128, 256, 512)

scala> xs.forall(_ < 1024)
val res0: Boolean = true

scala> xs.exists(_ == 3)
val res1: Boolean = false

scala> xs.count(_ > 64)
val res2: Int = 3

scala> xs.zipWithIndex.take(5)
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 innehållslikhet 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.

```
scala> val as1 = Array(1,2,3)
val as1: Array[Int] = Array(1, 2, 3)

scala> val as2 = Array(1,2,3)
val as2: Array[Int] = Array(1, 2, 3)


scala> val (xs1, xs2) = (as1.toSeq, as2.toSeq)
val xs1: Seq[Int] = ArraySeq(1, 2, 3)
val xs2: Seq[Int] = ArraySeq(1, 2, 3)

scala> as1 == as2
val res0: Boolean = false

```

```
scala> xs1 == xs2
val res1: Boolean = true

scala> as1(0) = 42

scala> xs1
val res2: Seq[Int] = ArraySeq(1, 2, 3)

scala> xs1(0) = 42
value update is not a member of Seq[Int]
```

b) Vid repeterade parametrar får man en `ArraySeq`.

```
scala> def f(xs: Int*) = xs
def f(xs: Int*): Seq[Int]

scala> println(f(1,2,3))
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. –