EDAA45 Programmering, grundkurs

Föreläsningar

Björn Regnell

Datavetenskap, LTH

Kompileringsdatum: 14 maj 2025

1 Introduktion

2 Program och kontrollstrukturer

3 Funktioner och abstraktion

4 Objekt och inkapsling

5 Klasser och datamodellering

6 Mönster och felhantering

7 Sekvenser och enumerationer

8 Nästlade och generiska strukturer

9 Mängder och tabeller

10 Arv och komposition

11 Varians och kontextparametrar

12 Fördjupning, Projekt

13 Repetition

14 MUNTLIGT PROV

1 Introduktion

- Om kursen
- Att lära denna läsvecka `w01`
- Om programmering
- De enklaste beståndsdelarna: litteraler, uttryck, variabler
- Funktioner
- Logik
- Satser
- Kontrollstrukturer
- Veckans uppgifter

Om kursen

Vem går pgk?

- D1|C1: Nybörjare på Datateknik|Infocom, kursen är obligatorisk men du ska ändå självregistrera dig i Ladok.
- Dx|Cx: Äldre LTH-studenter som är omregistrerade i senare årskurs eller bytt till Datateknik|Infocom

Viktiga länkar

- Kursens **öppna** hemsida: `http://cs.lth.se/pgk`

  Här finns det mesta, t.ex. dessa bilder, annat kursmaterial, hur du installerar verktyg på din egen dator etc.

  Lär dig hitta på hemsidan redan nu.

- Kursens **slutna** sida bakom inloggningsvägg: `https://canvas.education.lu.se/courses/31677`

  Här finns administrativ information om efterbeställning av bokpaket, gruppindelning, föreläsningsdeltagarlista, hemlig invit till Discord-server etc.

  Håll koll på ”anslag” i Canvas så du inte missar något.

  Du måste ha studentkonto, läs noga här: `https://www.student.lth.se/ny-student/`

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

Progression

Kursens koncept avancerar **steg för steg**:

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

Vi **itererar** över koncepten & **fördjupar** förståelsen **efter hand**.

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
TP	–	–	–
W08	Nästlade och generiska strukturer	matrices	life
W09	Mängder och tabeller	lookup	words
W10	Arv och komposition	inheritance	snake0
W11	Varians och kontextparametrar	context	snake1
W12	Fördjupning, Projekt	extra	Projekt0
W13	Repetition	examprep	Projekt1
W14	MUNTLIGT PROV	Munta	Munta
TP	VALFRI TENTAMEN	–	–

Kursutveckling och förnyelse

- **Scala** är förstaspråk på Datateknik (D) sedan **2016**.

  Den **största förnyelsen** av den inledande programmeringskursen sedan vi införde **Java 1997**.
- **Scala** är förstaspråk på InfoCom (C) sedan **2021**.
- **Scala 3** sedan 2021 med stora förenklingar för nybörjare + nya avancerade koncept för proffsen.
- **Ny examination** från 2021: muntligt prov + valfri tenta
- Kursmaterialet är **öppen källkod** och **fritt** tillgängligt.
- **Studentmedverkan** i kursutvecklingen:
  - Mer än 90 personer har bidragit på github.com/lunduniversity/introprog
- Se alla förbättringar sen förra året här: github.com/lunduniversity/introprog/commits
- Du är hjärtligt välkommen att bidra! Se instruktioner i kompendiet.

Historik förstaspråk på D & C vid LTH

Scala	**2016 (D), 2021 (C)**
Java	1997 (D), 2001 (C), InfoCom-programmet grundas
Simula	1990 (D)
Pascal	1982 (D), Datateknik-programmet grundas
Algol	(F, E, ...) förhistoria med hålkortsprogrammering

Historik förstaspråk på D & C vid LTH

Scala	**2016 (D), 2021 (C)**
Java	1997 (D), 2001 (C), InfoCom-programmet grundas
Simula	1990 (D)
Pascal	1982 (D), Datateknik-programmet grundas
Algol	(F, E, ...) förhistoria med hålkortsprogrammering

Scalas uppfinnare **Professor Martin Odersky** vid EPFL i Lausanne, Schweiz, har även skrivit stora delar av Java-kompilatorn, och var en gång i tiden doktorand för Prof. Niklaus Wirth, som låg bakom Algol, Pascal, Modula m.m. Första versionen av Scala kom 2004. `https://en.wikipedia.org/wiki/Scala_(programming_language)`

Varför Scala som förstaspråk?

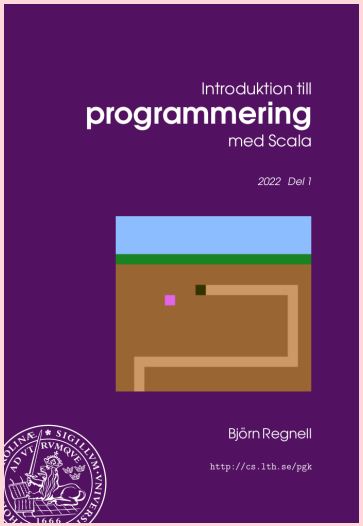
- Varför **Scala**?
  1. Enkel och enhetlig syntax => **lätt att skriva**
  2. Enkel och enhetlig semantik => **lätt att fatta**
  3. Kombinerar flera angreppssätt => **jämföra lösningar**
  4. Stark typning + statisk typning => **färre buggar**
  5. Typhärledning => **koncis kod**
  6. Scala Read-Evaluate-Print-Loop => **lätt att experimentera**
  7. Skalbart från lätt till avancerat => **nybörjare** + **fördjupning**
  8. Scala är **öppen källkod** + massor av fria kodbibliotek[1]
  9. Effektivitet: avancerad, mogen teknik => snabba program
  10. Stor industriell spridning: Netflix, LinkedIn, Twitter, Spotify, PayPal, Klarna, Sony, AirBNB, UBS, The Guardian, ...
  11. Scala och Java fungerar utmärkt tillsammans:
      - Java-bibliotek kan användas direkt i ditt Scala-program, och Java är det mest använda (?) språket på planeten Jorden

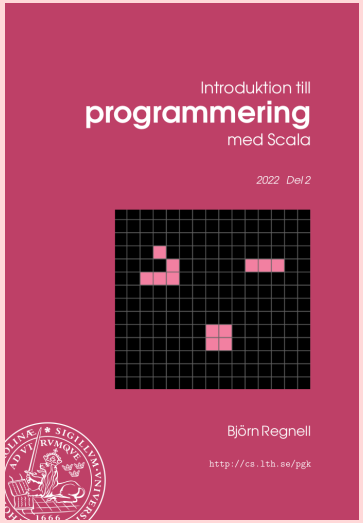
[1] `https://index.scala-lang.org/`

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!**

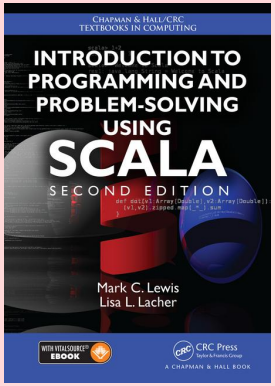
Kurslitteratur



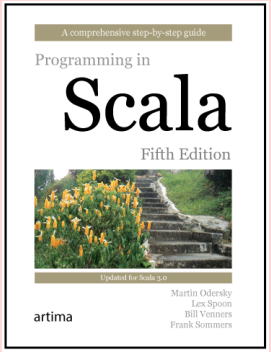


- **Kompendium**
  med övningar & laborationer,
  trycks till självkostnad enl. beställning,
  se info bakom Canvas-väggen.
- Föreläsningsbilder på kurshemsidan
  `http://cs.lth.se/pgk/`
- Nätresurser enl. länkar i kursmaterialet

Bra extra **överkursläsning**:
– för **nybörjare**:



– för de som **redan kan OO**:



Se `https://cs.lth.se/pgk/litteratur/`

Bokpaketet

UTHÄMTNING av beställda men ej uthämtade bokpaket sker på rasten i E:A i kl 14 i morgon tisdag, se info i Canvas.

- Kompendiet och snabbreferens trycks här i E-huset och säljs av institutionen till **självkostnadspris**. Se info i Canvas.
- **Kompendiet** finns i pdf för fri nedladdning enl. licensen CC-BY-SA, men det **rekommenderas starkt** att du även drar nytta av pappersversionen.
- För de flesta funkar det mycket bättre att ha övningar och labbar **på papper bredvid skärmen**, när du ska tänka, koda och plugga!
- **Snabbreferensen** finns också i pdf men du behöver ha en tryckt version eftersom det är **enda tillåtna hjälpmedlet** på skriftliga kontrollskrivningen och tentamen.
  Säljs separat, se info i Canvas ”Efterbeställning snabbreferens”.

Föreläsningsanteckningar

- Föreläsningbilder uppdateras under kursens gång.
- `http://cs.lth.se/pgk/foerelaesningar/`
- Fram till mån kl 13 aktuell vecka är de bilder som ligger ute under **pågående uppdatering** och kan således ändras.
- Latex-koden och ändringshistorik finns här: github.com/lunduniversity/introprog/tree/master/slides
- Kom gärna med förslag på innehåll och tips på förbättringar!

Personal 2024

Kursansvar: Prof. Björn Regnell, bjorn.regnell@cs.lth.se, E:2413

Handledare: **teknologer** anställda som s.k. **amanuenser**, 2024 (34 st) Adam Månsson, Alice Westerberg Zetterlund, André Philipsson Eriksson, Axel Langenskiöld, Axel Nilsson, Dag Hemberg, David Unelind, Edvin Norrman, Elias Åradsson, Erik Lundberg, Emil Sunnerdahl, Erik Heimdal, Esbjörn Stenberg, Hampus Edén, Hjalmar Rutberg, Johan Ekberg, Jon Mellerby, Johannes Nydahl, Karl Sellergren, Lovisa Löfgren, Marina Fridh-Cardoso, Maximilian Uggla, Moa Gassilewski, Naima Khan, Nils Klemming Nordenskiöld, Nils Melén, Noah Andreasson, Oscar Korpi, Rurik Hagman, Sander Holm, Sara Reimers, Tova Stengard, Tristan Farkas, Viggo Bergdahl

Kursadmin: Birger Swahn, rum E:2181, Ulrika Templing, rum E:2179
Karta till expeditionen och mer info här:
`http://cs.lth.se/kontakt/expedition/`

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`

Detta är bara början...

Exempel på efterföljande kurser som bygger vidare på denna:

- Programmeringsteknik – fördjupningskurs
- Objektorienterad modellering och design
- Programvaruutveckling i grupp
- Algoritmer, datastrukturer och komplexitet
- Funktionsprogrammering

Förkunskaper

- Förkunskaper $\neq$ Förmåga
- Varken kompetens eller personliga egenskaper är statiska
- ”Programmeringskompetens” är inte *en* enda enkel förmåga utan en komplex sammansättning av flera olika förmågor som **utvecklas** genom hela livet
- Ett innovativt utvecklar**team** behöver många olika kompetenser för att vara framgångsrikt

Stor spridning i förkunskaper

Har du programmerat tidigare?

Ja	130 respondenter	**74 %**	
Nej	45 respondenter	26 %	

Samarbetgrupper

- Ni delas in i **samarbetsgrupper** om `(4 to 6)` personer baserat på förkunskapsenkäten, så att olika förkunskapsnivåer sammanförs.
- En av laborationerna (`snake`) i lp2 är en mer omfattande **grupplabb** och kommer att göras i din samarbetsgrupp.
- Kontrollskrivningen i halvtid kan ge **samarbetsbonus** (max 5p) som adderas till valfria tentans poäng (max 100p) med medelvärdet av gruppmedlemmarnas individuella kontrollskrivningspoäng

  Bonus b för varje person i en grupp med n medlemmar med p_i poäng vardera på kontrollskrivningen:

$$b = \sum_{i=1}^{n} \frac{p_i}{n}$$

Varför studera i samarbetsgrupper?

Huvudsyfte: **Djupinriktat lärande!**

- Pedagogisk forskning stödjer tesen att lärandet blir mer djupinriktat om det sker i utbyte med andra
- Ett studiesammanhang med **höga ambitioner** och **respektfull gemenskap** gör att **alla lär sig mer**
- Varför ska du som redan kan mycket aktivt dela med dig av dina kunskaper?
  - Förstå bättre själv genom att förklara för andra
  - Träna din pedagogiska förmåga
  - Förbered dig för ditt kommande yrkesliv som mjukvaruutvecklare

Samarbetskontrakt

Gör ett skriftligt **samarbetskontrakt** gärna med dessa och ev. andra punkter som ni också tycker bör ingå:

1. Återkommande mötestider per vecka
2. Kom i tid till gruppmöten
3. Var väl förberedd genom självstudier inför gruppmöten
4. Hjälp varandra att förstå, men ta inte över och lös allt
5. Ha ett respektfullt bemötande även om ni har olika åsikter
6. Inkludera alla i gemenskapen

Diskutera hur ni ska uppfylla dessa innan alla skriver på.
Var och en tar med en egen **pappersutskrift** av samarbetskontraktet där **alla skrivit på** och visar för handledare på första labben.

Om arbetet i samarbetsgruppen inte fungerar ska ni mejla kursansvarig och boka mötestid för konsultation!

Dina frågor är viktiga!

- Det finns bättre och sämre frågor vad gäller hur mycket man kan lära sig av svaret, men **all undran är en chans** att i dialog utbyta erfarenheter och lärande.
- Den som frågar **vill veta** och berättar genom frågan något om nuvarande kunskapsläge.
- Den som svarar får chansen att **reflektera** över vad som kan vara svårt och olika vägar till djupare förståelse.
- I en hälsosam lärandemiljö är det **helt tryggt** att visa att man **ännu inte förstår**, att man gjort ”fel”, att man har mer att lära, etc.
- Det är viktigt att **våga försöka** även om det blir **”fel”**: **det är ju då man lär sig!**

Plagiatregler

- Läs dessa regler noga och diskutera i samarbetsgrupperna:
  - `http://cs.lth.se/utbildning/samarbete-eller-fusk/`
  - Föreskrifter angående obligatoriska moment
- Ni ska lära er genom **eget arbete** och **bra samarbete**.
- Samarbete gör att man lär sig bättre, men man lär sig **inte** av att kopiera andras lösningar.
- **Plagiering är förbjuden** och kan medföra **disciplinärende och avstängning**.
- Du får **INTE** lägga ut laborationslösningar öppet på **`github`** eller på annan plats där någon annan kan komma åt dem!
- Det är **INTE** tillåtet att använda **artificiell intelligens** i arbetet med laborationer och projekt i denna kurs.

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

Övningar

- **Programmering lär man sig bäst genom att programmera...**
- Genom veckans övningarna i kompendiet bearbetar du teorins olika delar via egna **undersökningar** med **Scala REPL** som viktigaste verktyg.
- Fokusera på att **förstå** vad som händer när du kör din kod. Om du bara rusar vidare utan reflektion lär du dig inte alls lika bra.
- Till de flesta övningar finns **facit**. Titta inte på facit förrän du själv gjort ett försök. Det finns ofta många olika sätt att åstadkomma samma sak, och din lösningsvariant kan vara lika bra som den i facit – använd REPL för att verifiera att din lösning fungerar. Diskutera med handledare om du är osäker på vad som är en mer eller mindre bra lösning.
- Skapa en miljö för **koncentration** och lärande **på djupet**. Stäng telefon, be kompisar i datorsalen att inte prata för högt, etc.

Laborationer

- På laborationerna **sammanför** du veckans koncept till en **helhet** i ett större program och kollar att du **kan grunderna** inför kommande veckor.
- Labbarna är **individuella** (utom en) och **obligatoriska**.
- Gör **övningarna** och **labbförberedelserna** noga **innan** själva labben – detta är ofta helt nödvändigt för att du ska hinna klart. Dina labbförberedelser kontrolleras av handledare under labben.
- Är du **sjuk?** Anmäl det **före** labben till `bjorn.regnell@cs.lth.se`, få hjälp på resurstid och redovisa på resurstid (eller labbtid, när handledaren har tid över)
- Hinner du inte med hela labben? Se till att handledaren noterar **”kompletteras”**, och fortsätt på resurstid och ev. uppsamlingstider.
- Läs noga kapitel ”**Anvisningar**” i kompendiet!
- Laborationstiderna är gruppindelade enligt schemat. Du ska gå till den tid och den sal som motsvarar din grupp som visas i TimeEdit.
- **Du hittar vilken grupp du tillhör i Canvas.**

Ge din **hjärna** rätt förutsättningar för programmering

Att lära denna läsvecka w01

Att lära denna läsvecka w01

Modul **Introduktion**: Övn `expressions` → Labb `kojo`

- ☐ 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
- ☐ dod: operativsystem

Om programmering

Att skapa koden som styr världen

I stort sett **alla** delar av samhället är beroende av programkod:

- kommunikation
- transport
- byggsektorn
- statsförvaltning
- finanssektorn
- media & underhållning
- sjukvård
- övervakning
- integritet
- upphovsrätt
- miljö & energi
- sociala relationer
- utbildning
- ...

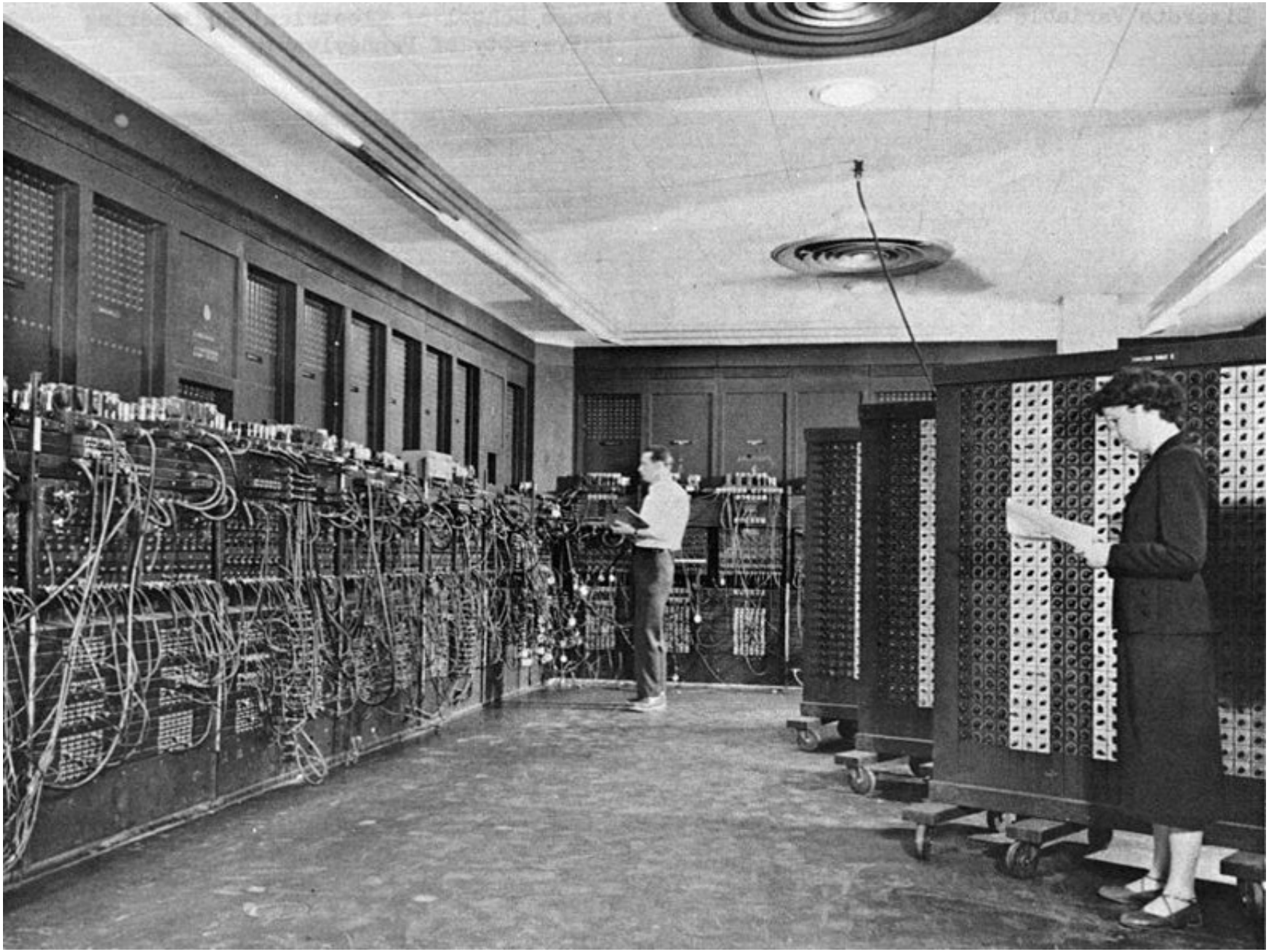
Hur blir ditt framtida yrkesliv som systemutvecklare?

- Det är sedan lång tid en **skriande brist** på utvecklare och det blir bara värre...
  CIO 2018-11-09
- Störst brist är det på **kvinnliga** utvecklare:
  SVT 2016-12-03
- Global kompetensmarknad
  CIO 2016-02-01
  CS 2015-06-14
  CS 2016-07-14

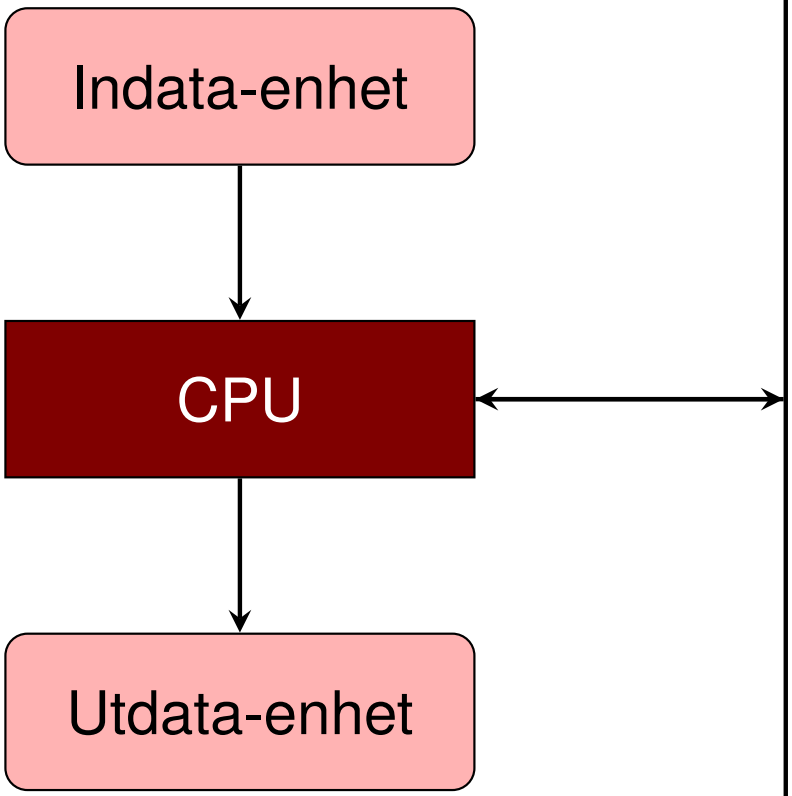
Utveckling av mjukvara i praktiken

- **Inte bara kodning:** kravbeslut, releaseplanering, design, test, versionshantering, kontinuerlig integration, driftsättning, återkoppling från dagens användare, ekonomi & investering, gissa om morgondagens användare, ...
- **Teamwork:** Inte ensamma hjältar utan autonoma team i decentraliserade organisationer med innovationsuppdrag
- **Snabbhet:** Att koda innebär att hela tiden uppfinna nya ”byggstenar” som ökar organisationens förmåga att snabbt skapa värde med hjälp av mjukvara. **Öppen källkod**. Skapa kraftfulla API:er. (API=application programming interface, byggstenar för att bygga appar)
- **Livslångt lärande:** Lär nytt och dela med dig hela tiden. Exempel på pedagogisk utmaning: hjälp andra förstå och använda ditt API $\Longrightarrow$ **Samarbetskultur**

Vad är en dator?



Hur fungerar en dator?



Minne med minnesceller

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

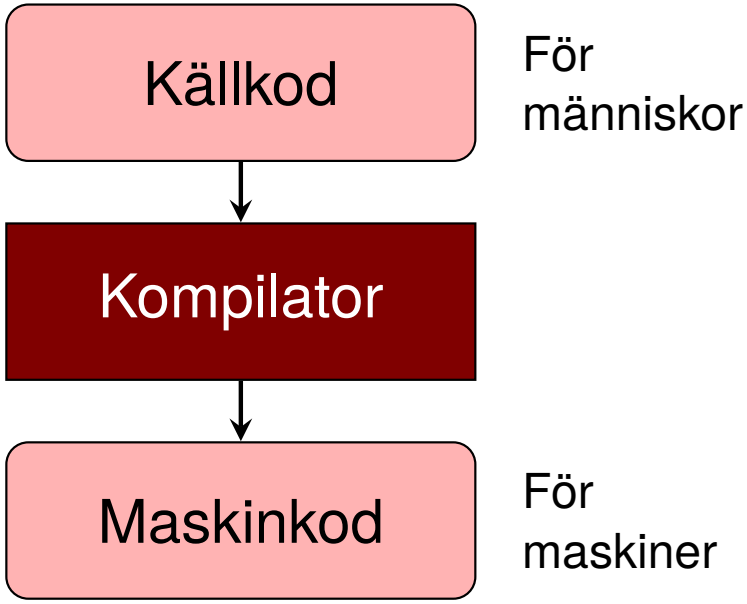
Minnet innehåller endast **heltal** som representerar **data och instruktioner**.

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!

Vad är en kompilator?

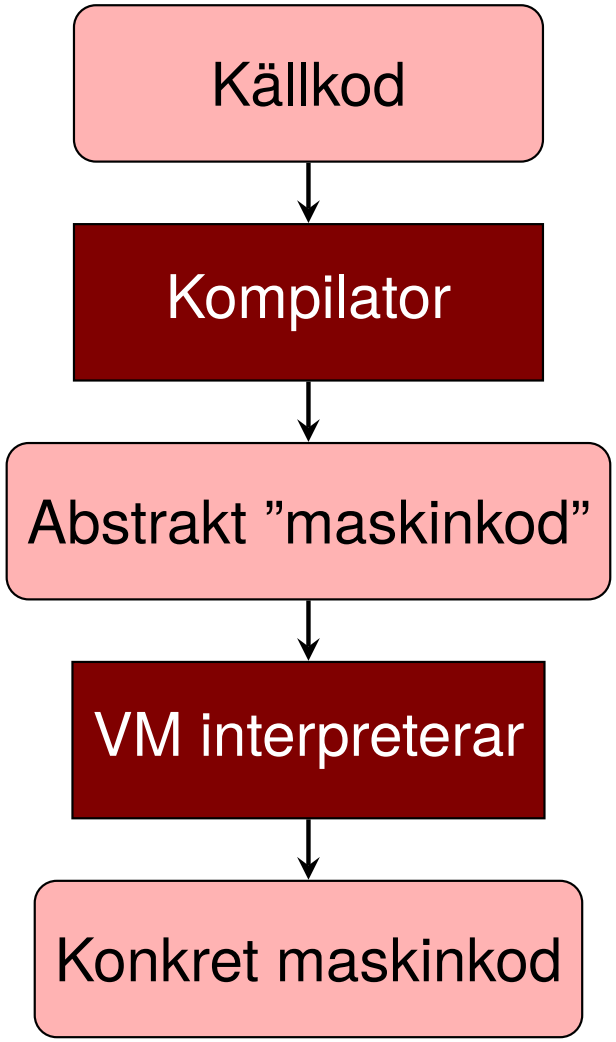


Grace Hopper uppfann kompilatorn 1952.

en.wikipedia.org/wiki/Grace_Hopper

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**



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

Exempel på programmeringsspråk

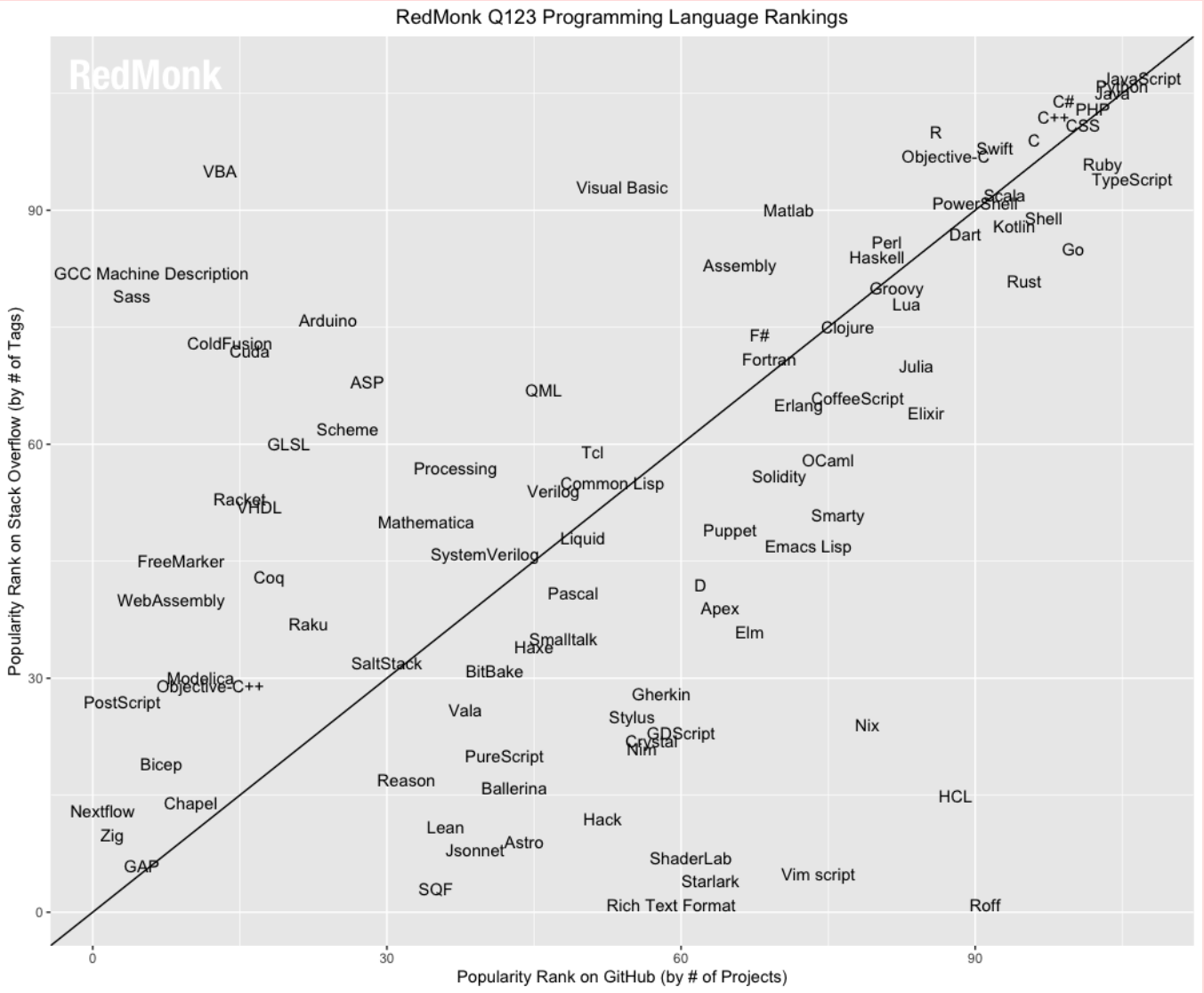
Det finns massor med olika språk och det kommer ständigt nya.

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

Några topplistor:

- Redmonk
- PYPL
- TIOBE

Redmonk Language Rankings: Github, Stackoverflow



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.

Hello world

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

```
> scala 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!
```

Hello world

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

```
> scala 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 compile hello.scala
```

Kör ditt program med `scala run` som kompilerar automatiskt vid behov.

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

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
}
```

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`: 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

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`
- 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. Drop-in: kl 12-13,
     4/9 E:3336, 5/9 E:3336, 6/9 E:2116, 9/9 E:2116,
     10/9 E:3336, 11/9 E:2116, 12/9 E:2116, 13/9 E:2116
  2. Pluggkvällar som SRD ordnar.
  3. `#frågor-och-svar` på vår Discord-server
  4. Fråga handledare på resurstid (i mån av tid).

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.
- I augusti 2024 blev **`scala-cli`** det nya **`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 help
```

`scala-cli help` är ”gamla” kommandot som också ingår i Scala-installationen.

[2] När kompendiet trycktes hade skiftet ännu inte skett. Nu kan du ersätta alla förekomster av `scala-cli` med det kortare `scala`.

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 repl .`
- Kör med punkt efter blanktecken så kompileras och exekveras alla scala-filer i **aktuell katalog och** eventuella **underkataloger**:
  `scala run .`

De enklaste beståndsdelarna: litteraler, uttryck, variabler

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.

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*)[3].
- Scala-kompilatorn gör **typhärledning**: man **slipper skriva typerna** om kompilatorn kan lista ut dem med hjälp av typerna hos deluttrycken.

[3] 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?

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`

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).

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**.

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.

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` ``

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}."
```

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
```

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`

Funktioner

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
```

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
```

Färdiga matte-funktioner i paketet `scala.math`

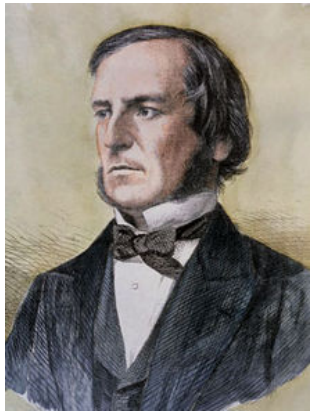
- I paketet **`scala.math`** finns många användbara funktioner: t.ex. `math.random()` ger slumptal mellan `0.0` och `0.99999999999999999`

```
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:
  `https://www.scala-lang.org/api/current/scala/math.html#`
- Paketet `scala.math` delegerar ofta till Java-klassen **`java.lang.Math`** som är dokumenterad här:
  `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Math.html`

Logik

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)
```

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)
```

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
```

Satser

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).

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.

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**.

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`

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
```

Övning: Tilldelningar i sekvens

Rita hur minnet ser ut efter varje rad nedan:

```
1 var u = 42
2 var x = 10
3 var y = 2 * x + 1
4 x = 20
5 var z = y + x + y - x
6 x += 1; y *= 2
```

En variabel som ännu inte **initierats** har ett **odefinierat** värde, anges nedan med frågetecken.

	rad 1	rad 2	rad 3	rad 4	rad 5	rad 6
u	42					
x	?					
y	?					
z	?					

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.

Kontrollstrukturer

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")
```

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).

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!")
```

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).

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.

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.

Veckans uppgifter

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 två 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, gammal syntax etc).
  3. Webbappen **`http://kojo.lu.se/`** direkt i webbläsare; rekommenderas ej – endast Scala 2, mer begränsad.

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

Köa med Sigrid

För att köa till handledare på plats i sal i pgk använd Sigrid.

(Se hemlig länk i Canvas, sprid ej länken på internet så vi slipper bottar).

- 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 använd Sigrid Monitor.

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*!

Om veckans övning: `expressions`

- 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.

Om veckans labb: `kojo`

- 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.

2 Program och kontrollstrukturer

- Om förra veckan och kommande två veckor
- Studieteknik
- Datastrukturer
- Kontrollstrukturer
- Fristående applikation
- Algoritmer: stegvisa lösningar
- Funktioner skapar struktur
- Katalogstruktur för kodfiler med paket
- Att göra denna vecka

Om förra veckan och kommande två veckor

Förra veckan

Viktiga övergripande mål:

- Förstå skillnaden mellan värde och typ
- Börja använda sekvens, alternativ, repetition, abstraktion
- Förstå variabel och tilldelning
- Reflektera över ditt lärande
- Träffas i samarbetsgrupper och lära känna varandra
- Komma in i kursens arbetsgång: förel. -> övn. -> labb

Om du inte hann klart labben, fortsätt på egen hand och på resurstid. **Avsätt mer tid till labbförberedelser nästa gång**. **Glöm inte** att du ska få labbhandledarens **underskrift** i protokollet i kompendiet på sidan iii när du är **närvarande** (godkänd eller kompletteras).

Fråga på resurstider och labbar

- På LTH skiljer vi tydligt på **undervisning** och **examination**.
- **Resurstider** är undervisning och du får fråga vad du vill!
- **Labbarna** är i huvudsak undervisning utom redovisningen på slutet som är examination. **Du får fråga vad du vill!**
- **Redovisningen** är en kontroll som hjälper dig avgöra om du har den **förståelse** som krävs för **fortsättningen**.
- Labbarna ger **godkänd**/**kompletteras** – ej graderat betyg.
- Labbarna påverkar inte slutbetyget men det krävs godkänt på **alla labbar** för att få göra muntligt prov och tentera.
- Att hjälpa varandra i samarbetsgrupperna med labbarna är **inte fusk**. Men att göra labben åt någon annan **är fusk**. Hjälp varandra att **förstå** begrepp och att komma vidare när någon kör fast. **Läs kapitel -1 Anvisningar**.

Kommande 2 veckor

Övergripande mål:

- Börja skriva dina **egna program**
- Träna på att dela upp din kod i **många små funktioner**

Läsvecka 2:

- Övning `programs`
- **OBS! Ingen labb denna vecka**, pga dod.

Läsvecka 3:

- Övning `functions`
- Labb `irritext`
  spela varandras textspel i samarbetsgrupper

OBS! Notera **schemaavvikelser** – veckorna är inte identiska:
`http://cs.lth.se/pgk/schema/timeedit/`

Studieteknik

Hur studerar du?

- Vad är bra **studieteknik**?
- **Aktivera dig!** Inte bara passivt läsa utan också aktivt göra.
- Hur skapa **struktur**? Du behöver ett sammanhang, ett **system av begrepp**, att **placera in** din nya kunskap i.
- Hur uppbåda **koncentration**? Steg 1: Stäng av mobilen!
- Hur vara **disciplinerad**? Studier först, nöje sen! Du måste inte vara med på allt under nollningen. Mitt råd: Hoppa över alla nollningsaktiviteter tills du är ikapp med i dina studier.
- Du måste **planera och omplanera** för att säkerställa **tillräckligt mycket egen pluggtid** då du är pigg och koncentrerad för att det ska funka!
- **Programmering kräver pigg && koncentrerad hjärna**!

Hur ska du studera programmering?

- När du gör **övningarna**:
  - Ta fram **föreläsningsbilderna** i pdf och kolla igen **innan** övningen.
  - Är det något i föreläsningsbilderna du inte förstår: ta upp det i samarbetsgrupperna eller på resurstiderna.
  - Om något är knepigt:
    - Hitta på egna **REPL-experiment**, undersök hur det funkar.
- Innan du gör **laborationerna**:
  - Gör **minst grundövningarna innan** du börjar med labben. Dubbelkolla att du har uppnått **övningsmålen**.
  - **Läs igenom hela labbinstruktionen innan** labben och gör en bedömning av din förberedelsetid.
  - Gör förberedelserna **i god tid innan** labben.
  - För varje labb behövs **allt större del** av koden **göras klart innan** ditt labbtillfälle. Mot slutet av kursen används nästan hela labbtiden till redovisning.

Det går inte att förstå allt på en gång!

- Vi **nosar** på ett visst begrepp **lite grand** i en vecka ...
- ... för att i senare vecka **återkomma** till det, men **djupare**.
- Förståelse kommer efter hand och kräver bearbetning.
- Vi måste **iterera begreppen** innan vi kan nå djup.
- Det är svårt för dig nu att se vad som är **detaljer** som du inte ska hänga upp dig på, och vad som är **det viktiga** i detta läget. **Men det kommer! Ha tålamod!**

Från förra veckan: `val` `var` `def`

Vad är det för skillnad på (och likhet mellan) **`val`**, **`var`** och **`def`**?

Från förra veckan: `val` `var` `def`

Vad är det för skillnad på (och likhet mellan) **`val`**, **`var`** och **`def`**?

- Med **`val`** deklareras en variabel som tilldelas ett värde vid initialisering och som sedan **aldrig ändras**.
- Med **`var`** deklareras en variabel som tilldelas ett värde vid initialisering och som sedan **kan uppdateras** hur många gånger som helst med hjälp av tilldelningssatser.
- Med **`def`** deklareras en funktion som körs vid **varje anrop**

Från förra veckan: `val` `var` `def`

Vad är det för skillnad på (och likhet mellan) **`val`**, **`var`** och **`def`**?

- Med **`val`** deklareras en variabel som tilldelas ett värde vid initialisering och som sedan **aldrig ändras**.
- Med **`var`** deklareras en variabel som tilldelas ett värde vid initialisering och som sedan **kan uppdateras** hur många gånger som helst med hjälp av tilldelningssatser.
- Med **`def`** deklareras en funktion som körs vid **varje anrop**

En **konstighet i REPL**:

- Man kan i REPL **deklarera** variabler och funktioner med **samma namn flera gånger** på samma nivå.
- Detta ger i vanliga, fristående program **kompileringsfel**.

Från förra veckan: `if then else`

Spelets regler: Singla slant. Om du får krona har du vunnit.

Förenkla dessa if-uttryck:

```
def singlaSlant: Boolean = if math.random() < 0.5 then true else false

def harVunnit(slantÄrKrona: Boolean): Boolean =
  if slantÄrKrona == true then true else false
```

Från förra veckan: `if then else`

Spelets regler: Singla slant. Om du får krona har du vunnit.

Förenkla dessa if-uttryck:

```
def singlaSlant: Boolean = if math.random() < 0.5 then true else false

def harVunnit(slantÄrKrona: Boolean): Boolean =
  if slantÄrKrona == true then true else false
```

förenkling av ”kaka-på-kaka”: **if uttryck then true else false**

```
def singlaSlant: Boolean = math.random() < 0.5
```

Från förra veckan: `if then else`

Spelets regler: Singla slant. Om du får krona har du vunnit.

Förenkla dessa if-uttryck:

```
def singlaSlant: Boolean = if math.random() < 0.5 then true else false

def harVunnit(slantÄrKrona: Boolean): Boolean =
  if slantÄrKrona == true then true else false
```

förenkling av ”kaka-på-kaka”: **if uttryck then true else false**

```
def singlaSlant: Boolean = math.random() < 0.5
```

förenkling av ”kaka-på-kaka”: **uttryck == true**

```
def harVunnit(slantÄrKrona: Boolean): Boolean = slantÄrKrona
```

Från förra veckan: `if then else`

Spelets regler: Singla slant. Om du får krona har du vunnit.

Förenkla dessa if-uttryck:

```
def singlaSlant: Boolean = if math.random() < 0.5 then true else false

def harVunnit(slantÄrKrona: Boolean): Boolean =
  if slantÄrKrona == true then true else false
```

förenkling av ”kaka-på-kaka”: **if uttryck then true else false**

```
def singlaSlant: Boolean = math.random() < 0.5
```

förenkling av ”kaka-på-kaka”: **uttryck == true**

```
def harVunnit(slantÄrKrona: Boolean): Boolean = slantÄrKrona
```

Hur ska vi ändra `harVunnit` om vi ändrar reglerna till att Klave ger vinst?

Från förra veckan: `if then else`

Spelets regler: Singla slant. Om du får krona har du vunnit.

Förenkla dessa if-uttryck:

```
def singlaSlant: Boolean = if math.random() < 0.5 then true else false

def harVunnit(slantÄrKrona: Boolean): Boolean =
  if slantÄrKrona == true then true else false
```

förenkling av ”kaka-på-kaka”: **if uttryck then true else false**

```
def singlaSlant: Boolean = math.random() < 0.5
```

förenkling av ”kaka-på-kaka”: **uttryck == true**

```
def harVunnit(slantÄrKrona: Boolean): Boolean = slantÄrKrona
```

Hur ska vi ändra `harVunnit` om vi ändrar reglerna till att Klave ger vinst?

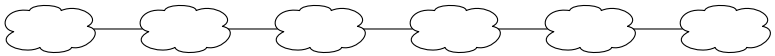
Repetera även **de Morgans lagar** från vecka 1 så du har koll på dem.

Datastrukturer

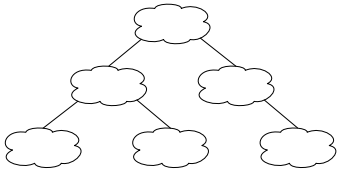
Vad är en datastruktur?

- 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:

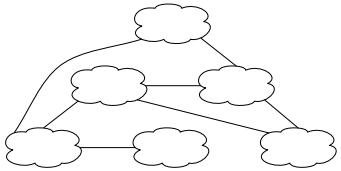
Sekvens



Träd



Graf



Mer om sekvenser & träd i EDAA01 pfk. Mer om träd, grafer i Diskreta strukturer.

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.

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

Vad är en vektor?

En **vektor**[4] (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)`:

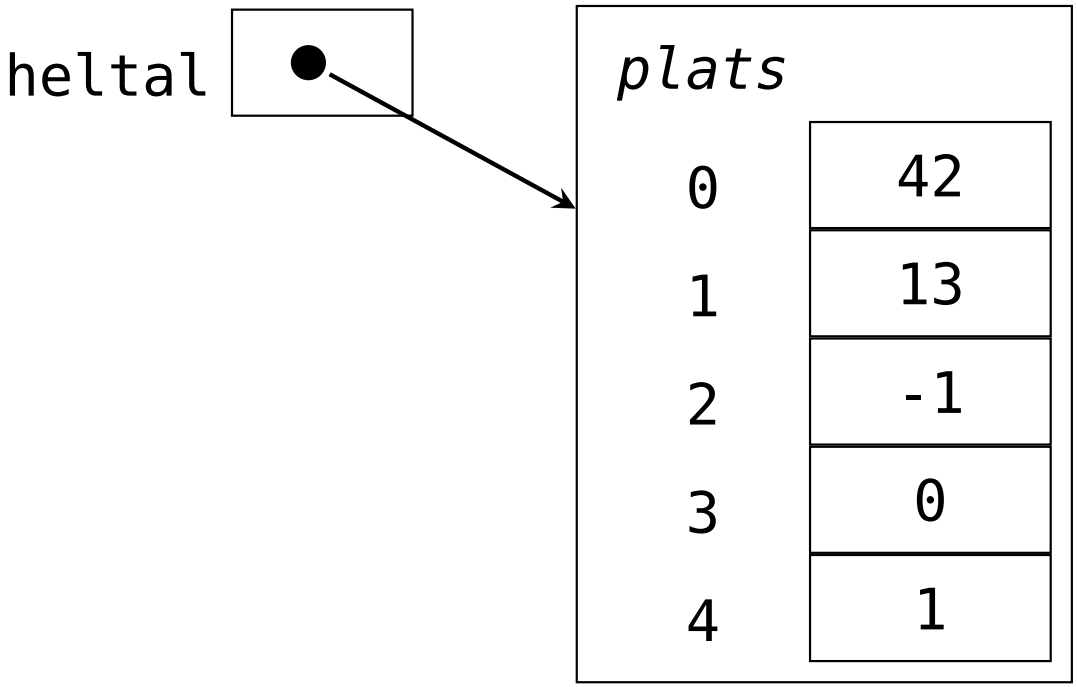
```
scala> val heltal = Vector(42, 13, -1, 0, 1)
val heltal: scala.collection.immutable.Vector[Int] = Vector(42, 13, -1, 0, 1)

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`.

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

En konceptuell bild av en vektor

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

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



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.)

Kontrollstrukturer

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}
  ```

Mitt första program: en oändlig loop på ABC80

```
10 print "hej"
20 goto 10
```

Mitt första program: en oändlig loop på ABC80

```
10 print "hej"
20 goto 10
```

```
hej
hej
hej
hej
hej
hej
hej
hej
hej
hej
hej
hej
<Ctrl+C>
```

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.

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)
```

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.Exclusive = Range 0 until 2147483647

scala> jättestortIntervall.end
val res1: Int = 2147483647

scala> jättestortIntervall.toVector
java.lang.OutOfMemoryError: Java heap space
```

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
```

Med metoden `indices` på kan du få en Range med alla index:

```
scala> val xs = Vector("gurka1","gurka2","tomat1")
val xs: Vector[String] = Vector(gurka1, gurka2, tomat1)

scala> xs.indices
val res0: Range = Range 0 until 3
```

Loopa med Range skapad med `to`

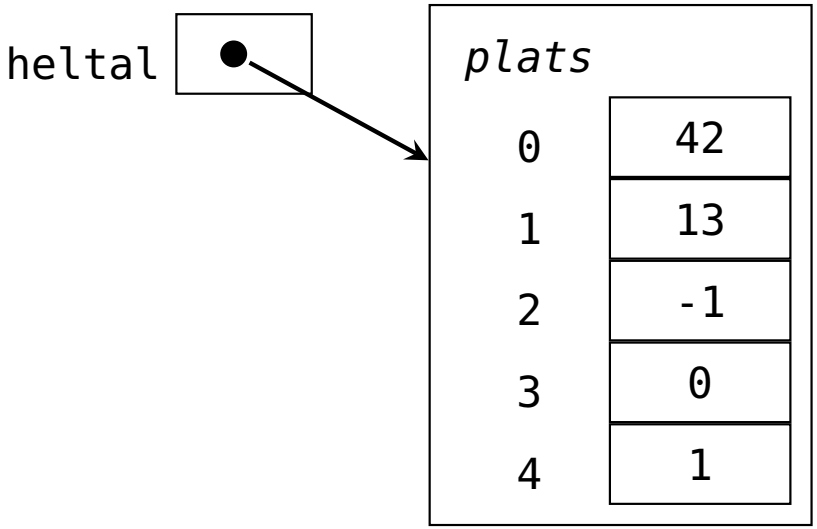
Med `to` efter ett heltal får du en Range till och **med** sista:

```
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
```

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)
```



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.
- Kan **ej** ändra storlek: tillägg eller borttagning mitt i kräver **långsam** kopiering av resten.

Fristående applikation

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 i terminalen: `scala` eller `scala-cli`
- Installera så här: `http://cs.lth.se/pgk/verktyg`
- Läs mer i Appendix C.
- Tips om du kör Windows: installera nya Windows Terminal

Få hjälp i kanalerna `#installationskrångel` och `#frågor-och-svar` på vår Discord-server eller fråga handledare på resurstid.

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.
- I augusti 2024 blev **`scala-cli`** det nya **`scala`**
- Du kan nu ersätta `scala-cli` med `scala`
- Läs mer i Appendix C och F, samt här: `https://scala-cli.virtuslab.org/`
- Se vad Scala CLI kan göra med underkommandot `help`

```
scala help
```

Ett minimalt fristående program i Scala

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

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

Kompilera och kör i terminalen:

```
> scala run hej.scala
Compiling project (Scala 3.5.0, JVM (21))
Compiled project (Scala 3.5.0, JVM (21))
Hej Scala!
```

Innan körning kompileras dina kodfiler automatiskt vid behov. Du kan se maskinkoden i en underkatalog i till katalogen `.scala-build`:

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

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
!!!
```

Strängargument till i ett program med primitiv main

Skriv och spara nedan kod i filen `helloargs1scala`

```
> code helloargs1.scala
```

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

Typsäkra argument till i ett program med @main

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

```
> code helloargs2.scala
```

```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 run helloargs2.scala -- 2 morot gurka tomat
morot
gurka
> scala 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.

För kännedom: Scala-**skript**

- Scala-kod kan köras som ett **skript**.
- 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 run myscript.sc -- ett två tre
Hej alla mina argument:
Hej: ett
Hej: två
Hej: tre
```

För kännedom: Scala-**skript**

- Scala-kod kan köras som ett **skript**.
- 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 run myscript.sc -- ett två tre
Hej alla mina argument:
Hej: ett
Hej: två
Hej: tre
```

Ett Scala-skript kan ej anropa andra skript utan speciella åtgärder, se detaljer här:

`https://scala-cli.virtuslab.org/docs/guides/scripting/scripts/`

Algoritmer: stegvisa lösningar

Vad är en algoritm?

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

Exempel:

- baka en kaka

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

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

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

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

Varför?

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

Varför? Så att de som spelar uppmuntras att spela mer :)

Algoritm:

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

Varför? Så att de som spelar uppmuntras att spela mer :)

Algoritm:

1. *points* ← poängen efter senaste spelet
2. *highscore* ← bästa resultatet innan senaste spelet
3. **om** *points* är större än *highscore* **så**
   - Skriv ”Försök igen!”

   annars
   - Skriv ”Grattis!”

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

Varför? Så att de som spelar uppmuntras att spela mer :)

Algoritm:

1. *points* ← poängen efter senaste spelet
2. *highscore* ← bästa resultatet innan senaste spelet
3. **om** *points* är större än *highscore* **så**
   - Skriv ”Försök igen!”

   annars
   - Skriv ”Grattis!”

Hittar du buggen?

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

Varför? Så att de som spelar uppmuntras att spela mer :)

Algoritm:

1. *points* ← poängen efter senaste spelet
2. *highscore* ← bästa resultatet innan senaste spelet
3. **om** *points* är större än *highscore* **så**
   - Skriv ”Försök igen!”

   annars
   - Skriv ”Grattis!”

Hittar du buggen?

Utskriften blir fel; vänd villkor eller byt plats på grenarna i if-satsen

HIGHSCORE implementerad i Scala

```
import scala.io.StdIn.readLine

@main
def run =
  val points = readLine("Hur många poäng fick du?").toInt
  val highscore = readLine("Vad var highscore före senaste spelet?").toInt
  val msg = if points > highscore then "GRATTIS!" else "Försök igen!"
  println(msg)
```

HIGHSCORE implementerad i Scala

```
import scala.io.StdIn.readLine

@main
def run =
  val points = readLine("Hur många poäng fick du?").toInt
  val highscore = readLine("Vad var highscore före senaste spelet?").toInt
  val msg = if points > highscore then "GRATTIS!" else "Försök igen!"
  println(msg)
```

Är det en bugg eller en feature att det står
`points > highscore`
och inte
`points >= highscore`
?

HIGHSCORE implementerad i Scala

```
import scala.io.StdIn.readLine

@main
def run =
  val points = readLine("Hur många poäng fick du?").toInt
  val highscore = readLine("Vad var highscore före senaste spelet?").toInt
  val msg = if points > highscore then "GRATTIS!" else "Försök igen!"
  println(msg)
```

Är det en bugg eller en feature att det står
`points > highscore`
och inte
`points >= highscore`

? Man får ej GRATTIS om poäng == highscore vilket är tråkigt :)

Algoritmexempel: N-FAKULTET

Indata : heltalet n
Utdata: produkten av de första n positiva heltalen

$prod \leftarrow 1$
$i \leftarrow 2$
while $i \leq n$ **do**
  $prod \leftarrow prod * i$
  $i \leftarrow i + 1$
end
$prod$

Algoritmexempel: N-FAKULTET

Indata : heltalet n
Utdata: produkten av de första n positiva heltalen

$prod \leftarrow 1$
$i \leftarrow 2$
while $i \leq n$ **do**
  $prod \leftarrow prod * i$
  $i \leftarrow i + 1$
end
$prod$

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

Algoritmexempel: MIN

Indata : Array *args* med strängar som alla innehåller heltal
Utdata: minsta heltalet

```
min ← det största heltalet som kan uppkomma
n ← antalet heltal
i ← 0
while i < n do
    x ← args(i).toInt
    if ( x < min) then
        min ← x
    end
    i ← i + 1
end
min
```

Algoritmexempel: MIN

Indata : Array *args* med strängar som alla innehåller heltal
Utdata: minsta heltalet

$min \leftarrow$ det största heltalet som kan uppkomma
$n \leftarrow$ antalet heltal
$i \leftarrow 0$
while $i < n$ **do**
  $x \leftarrow args(i).toInt$
  if $(x < min)$ **then**
    $min \leftarrow x$
  end
  $i \leftarrow i + 1$
end
min

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

Funktioner skapar struktur

Mall för funktionsdefinitioner

def funktionsnamn(parameterdeklarationer): returtyp = uttryck

Mall för funktionsdefinitioner

def funktionsnamn(parameterdeklarationer): returtyp = uttryck

Exempel:

```
def öka(i: Int): Int = i + 1
```

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!

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
}
```

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.

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(":(")
```

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))
```

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**.

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 }
```

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
```

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?
```

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.

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.

”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`

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.

Exempel på **funktionell nedbrytning**

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

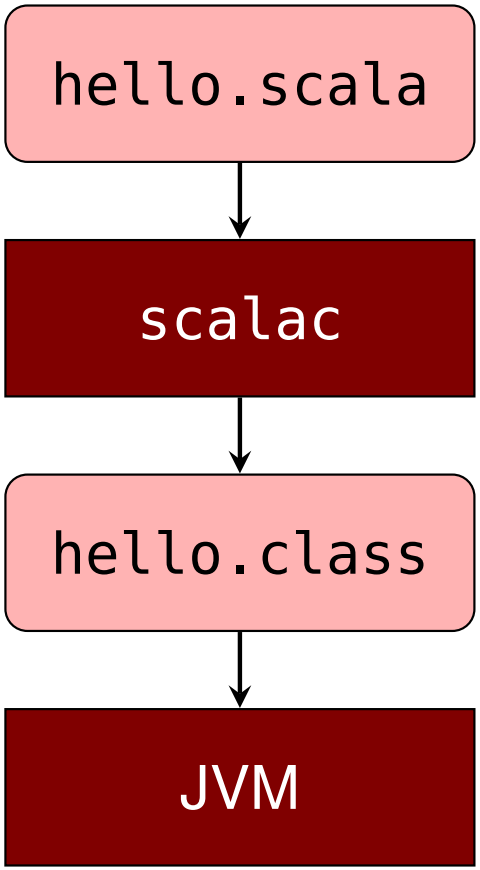
```
// skapa abstraktioner som bygger på varandra

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
```

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 skapa namnrymder och organisera maskinkod i en hierarkisk katalogstruktur.
    Nyckelord: `package`

Katalogstruktur för kodfiler med paket

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)

Paket

```
package greeting

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

- Paket (eng. *package*) skapar namnrymder och i en hierarkisk struktur.
- 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.[5]
  Är du nyfiken, kolla underkataloger i `.scala-build`:

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

[5] 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`**.

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 namnet syns **direkt**, och man slipper skriva hela vägen till namnet:

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

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

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 run . --jar introprog.jar
```

- Du kan skapa egna jar-filer med `scala package` där optionen `--library` gör så att endast den komilerade koden inkluderas. Utan optionen `--library` så görs jar-filen exekverbar. Med optionen `--assembly` tas allt med i jar-filen som behövs för att köra jar-filen helt fristående med ett dubbelklick eller `java -jar myapp.jar`

```
scala package . --library --output myapp.jar
scala run --jar myapp.jar
scala package . --assembly --output my-fat-jar-app.jar
java -jar my-fat-jar-app.jar
```

Optionen `--assembly` kräver power-läge enl. instruktioner i varning.
Läs mer om jar-filer i Appendix F.

Att göra denna vecka

1. Gör övning `programs`
2. OBS! Ingen lab denna vecka w02. Använd tiden att komma ikapp om du ligger efter!
3. Träffas i samarbetsgrupper och hjälp varandra att förstå.
4. Vi har nosat på flera koncept som vi kommer tillbaka till senare: du kommer förstå mer detaljer på djupet då.
5. Om ni inte redan gjort det:
   Visa samarbetskontrakt för handledare på resurstid.
6. **Koda på resurstiderna** och få hjälp och tips!

Veckans övning: `programs`

- 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.

Labb läsvecka 3: `irritext`

- Skapa ett **lagom** irriterande textspel som körs i terminalen:
  - ska vara **lagom** irriterande om man **först läser koden**
  - får gärna vara **orimligt** irriterande om man **inte läser koden**
  - koden ska vara **lättläst** och uppdelad i **många små funktioner** med bra, förklarande funktionsnamn, parameternamn och variablenamn
- Använd en editor, kompilera och kör i terminalen
- Mål: skapa **eget** program med **många små funktioner** och träna på **alla begrepp** vi använt hittills. Ju fler begrepp du kan använda på olika sätt desto bättre. Fokusera på det **du** behöver träna mest på.
- Spela varandras textspel inom din samarbetsgrupp
- Utveckla ditt spel **stegvis** och spela varandras halvfärdiga spel i flera omgångar. Ge varandra tips om förbättringar för att spelet ska bli mer lagom irriterande på ett kul sätt och för att koden ska bli mer lättläst.
- Skriv ner den återkoppling du fått av din grupp inför labbredovisningen.
- Läs igenom labbuppgiften redan nu och börja fundera. Ta dig inte ”vatten över huvudet”. Ta små steg i början och ha hela tiden körbar kod.

Alla kodar loss!

Vi ses nästa vecka!

3 Funktioner och abstraktion

- Abstraktion
- Vad är en funktion?
- Hur ser det ut i minnet när funktioner anropas?
- En funktion är ett värde
- Äkta funktioner
- Anonyma funktioner
- Skapa din egen kontrollstruktur
- Kort om rekursion
- Automatisk omkompilering
- Veckans uppgifter

Hämta beställda bokpaket, snabbref!

- Du som **ännu inte** hämtat beställd trycksak:

  Kom överens med Birger om tid via mejl till `birger.swahn@cs.lth.se`:

  Hämtas hos Birger vid Datavetenskaps **expedition** på andra våningen i E-huset trapphus A

Kursombud

- Om du är intresserad: fyll i **enkät** om **kursombud** i Canvas.
- Kursombud träffar vid behov kursansvarig på rast mellan föreläsningar eller chattar med varandra och kursansvarig på Discord **under kursens gång**.
- Kursombud träffar kursansvarig och programledning **efter kursen** och diskuterar **kursutvärderingen** CEQ.
- Det vore bra med minst 2st D:are och minst 2st C:are och gärna minst 1st W:are.
- Läs mer om studierådet här:
  `https://www.dsek.se/committees/srd`

Abstraktion

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`

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**!

Vad är en funktion?

Om veckans tema: Funktioner

- Funktioner är en av de viktigaste abstraktionsmekanismerna inom datavetenskapen
- Du kan redan massor om funktioner, bl.a. från matematiken.
- Denna vecka ska vi fördjupa förståelsen:
  - överlagring
  - enhetlig access
  - defaultargument
  - namngivna argument
  - lokala funktioner
  - funktioner som äkta värden
  - anonyma funktioner
  - klammerparentes vid ensam paramenter
  - multipla parameterlistor
  - egendefinierade kontrollstrukturer
  - fördröjd evaluering (”call-by-name”)
  - stegade funktioner (”Curry-funktioner”)
  - fångad variablelrymd (”closure”)

Om veckans tema: Funktioner

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 funktionens **returvärde**.

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 funktionens **returvärde**.

Exempel:

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

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 funktionens **returvärde**.

Exempel:

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

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

Deklarera funktioner, överlagring

- Överlagrade funktioner i samma namnrymd:

```
scala> object matte:
         def öka(a: Int): Int = a + 1
         def öka(a: Int, b: Int): Int = a + b

scala> matte.öka(1)
val res0: Int = 2

scala> matte.öka(1, 2)
val res1: Int = 3
```

- 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.

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.

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
```

Enhetlig access

- Om en funktion **deklareras med** tom parameterlista ( ) så *ska* den **anropas med** tom parameterlista. (Undantag: Java-metoder)

```
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
```

Enhetlig access

- Om en funktion **deklareras med** tom parameterlista `()` så *ska* den **anropas med** tom parameterlista. (Undantag: Java-metoder)

```
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.

Hur ser det ut i minnet när funktioner anropas?

Anropsstacken och objektheapen

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

- **Anropsstacken**:
  - På anropsstacken läggs en **aktiveringspost** (eng. *stack frame*[6], *activation record*) för varje funktionsanrop med plats för **parametrar** och **lokala variabler**.
  - Aktiveringsposten **raderas** när **returvärdet** har levererats.
  - Stacken **växer** vid **nästlade funktionsanrop**, då en funktion i sin tur anropar en annan funktion.
- **Objektheapen**: I objektheapen[7,8] 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.

---

[6] en.wikipedia.org/wiki/Call_stack
[7] en.wikipedia.org/wiki/Memory_management
[8] Ej att förväxlas med datastrukturen heap sv.wikipedia.org/wiki/Heap

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()
```

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...

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

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

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

Vad är en stack trace?

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

```
// 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
```

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.

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.[9]

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

En funktion är ett värde

Funktioner är äkta värden i Scala

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

Funktioner är äkta värden i Scala

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

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

scala> val f = add
val f: (Int, Int) => Int = Lambda7210/0x0000000841e4e040

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

- Ett funktionsvärde har en **typ** precis som alla värden:
  `f: (Int, Int) => Int`

Funktioner är äkta värden i Scala

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

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

scala> val f = add
val f: (Int, Int) => Int = Lambda7210/0x0000000841e4e040

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).

Funktionsvärden kan vara argument

Funktioner kan ha funktioner som parametrar:

```
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
```

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

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)
```

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?
```

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)
```

Metoden **`map`** är en smidig och ofta använd **högre ordningens funktion**.

Äkta funktioner

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*). Det innebär att **varje anrop kan bytas ut** mot **värdet av funktionskroppen** där parametrarna ersatts med motsvarande argument före evaluering.
- 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 värde. Alltså: *ej referentiellt transparent*.

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.

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.

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)      // något av heltalen 0, 1, 2, 3, 4, 5
```

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)      // något av heltalen 0, 1, 2, 3, 4, 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 vid varje programkörning
val r = rnd.nextInt(6)
```

Anonyma funktioner

Anonyma funktioner

- Man behöver inte ge funktioner namn. De kan i stället skapas med hjälp av **funktionslitteraler**.[10]
- En funktionslitteral har ...
  1. en parameterlista (utan funktionsnamn, utan returtyp),
  2. sedan den reserverade teckenkombinationen **=>**
  3. och sedan ett uttryck (eller ett block).

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

Anonyma funktioner

- Man behöver inte ge funktioner namn. De kan i stället skapas med hjälp av **funktionslitteraler**.[10]
- 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 har denna funktionslitteral?

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

Anonyma funktioner

- Man behöver inte ge funktioner namn. De kan i stället skapas med hjälp av **funktionslitteraler**.[10]
- 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 har denna funktionslitteral? `(Int, Int) => Int`

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

Anonyma funktioner

- Man behöver inte ge funktioner namn. De kan i stället skapas med hjälp av **funktionslitteraler**.[10]
- 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 har denna funktionslitteral? `(Int, Int) => Int`

- Om kompilatorn kan gissa typerna från sammanhanget så behöver typerna inte anges i själva funktionslitteralen:

```
val f: (Int, Int) => Int = (x, y) => x + y
```

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

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)
```

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 typen `Int` så behövs den inte:

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

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 typen `Int` så behövs den 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)
```

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
```

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.

```
_ + _
```

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
```

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
```

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`.

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`

Skapa din egen kontrollstruktur

Hur fungerar egentligen `upprepa` i Kojo?

```
upprepa(10) {
  println("hej")
}
```

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)

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

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

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 4
```

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 4
```

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 42 hej
```

Kluring: Varför skrivs ”hej” ut en extra gång i början?

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 4
```

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 42 hej
```

Kluring: Varför skrivs ”hej” ut en extra gång i början? ledtråd: `1 to` **`n`**

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?

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 yttre parentesparet för **bättre läsbarhet**:

```
scala> twice { print("hej"); print("san ") }
```

Skapa din egen kontrollstruktur

- Genom att **kombinera multipla parameterlistor** med **namnanrop** med **klammerparentes vid ensam parameter** kan vi skapa vår egen kontrollstruktur: `upprepa`

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?

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
```

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 to
```

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.[11]

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

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)
```

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**.

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@1bda2

scala> funkis(2)
val res0: Int = 45
```

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@1bda2

scala> funkis(2)
val res0: Int = 45
```

Ett funktionsobjekt med ”fångade” variabler kallas **closure**.
(Mer om funktioner som objekt senare.)

Översikt av begrepp vi gått igenom hittills

1. överlagring
2. utelämna tom parameterlista (enhetlig access)
3. defaultargument
4. namngivna argument
5. lokala funktioner
6. funktioner som äkta värden
7. anonyma funktioner
8. klammerparentes vid ensam paramenter
9. multipla parameterlistor
10. namnanrop (fördröjd evaluering)
11. egendefinierade kontrollstrukturer
12. stegade funktioner (”Curry-funktioner”)
13. fångad variablelrymd i funktionsobjekt (”closure”)

Kort om rekursion

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.

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 :)")
```

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.

Automatisk omkompilering

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.

Veckans uppgifter

Mål med övning functions

- Kunna skapa och använda funktioner med en eller flera parametrar, default-argument, och namngivna argument.
- Kunna förklara nästlade funktionsanrop med aktiveringsposter på stacken.
- Kunna förklara skillnaden mellan äkta och ”oäkta” funktioner.
- Kunna applicera en funktion på alla element i en samling.
- Kunna använda funktioner som äkta värden.
- Kunna skapa och använda anonyma funktioner (ä.k. lambda-funktioner).
- Känna till att funktioner kan ha uppdelad parameterlista.
- Känna till att det går att partiellt applicera argument på funktioner med uppdelad parameterlista för att skapa s.k. stegade funktioner (ä.k. curry-funktioner).
- Känna till rekursion och kunna beskriva vad som kännetecknar en rekursiv funktion.
- Känna till att det går att skapa egna kontrollstrukturer med hjälp av namnanrop.
- Känna till skillnaden mellan värdeanrop och namnanrop.
- Kunna tolka en stack trace.

Mål med laboration irritext

- 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.

Ni ska spela **varandras** textspel i din **samarbetsgrupp**.
Läs labbinstruktioner:`http://cs.lth.se/pgk/compendium/`

Tips till ditt textspel.

```
"Yes".toLowerCase.startsWith("y")    // true
"hejsan".contains("ejsa")            // true
"42".toInt                           // 42
"?".toInt                            // ger krasch (undantaget NumberFormatException)
"?".toIntOption.getOrElse(42)        // 42 (toIntOption kan inte krascha)
// ett annat sätt att förhindra krasch med try ... catch:
val x = try { "?".toInt } catch { case e: Exception => 42 }

val i = 42
s"Livets mening är $i!" // dollar $ före namn vid stränginterpolering med s""
s"Livets mening är inte ${i-1}!"  // klamrar ${} vid evaluering av uttryck

"""|en sträng som spänner över
   |flera rader där marginalen fram till vertikalstreck
   |är bortplockad med stripMargin (kan kombineras med s-interpolatorn)
""".stripMargin

math.random() < 0.8                  // true i 80% av fallen
scala.util.Random.nextInt(42)      // ger slumptal mellan 0 och 41
scala.io.StdIn.readLine("prompt>") // ger sträng som användaren skriver

Thread.sleep(1000)    // sova i 1 sekunder, en lagom irriterande fördröjning
```

Kolla **snabbreferensen** vad mer du kan göra med strängar!

Exempel på en början till ett textspel

Här finns en exempel på en enkel *början* på ett textspel som du stegvis kan ändra och bygga ut till något du själv vill göra: `https://github.com/lunduniversity/introprog/tree/master/workspace/w03_irritext`

- Vilka begrepp och principer ger koden träning i?

Jobba så här

- Skriv koden i editorn vs code som du startar med `code .`
- Dra igång 3 olika terminalfönster:
  1. `scala compile . --watch`
     så att din kod kompileras om automatiskt vid varje sparning med Ctrl+S.
  2. `scala repl .`
     så att du kan göra mindre undersökningar rad för rad, medan du tänker. Ctrl+D avslutar REPL, så du kan börja om efter kodändring.
  3. `scala run .`
     för att se om programmet funkar som det ska. Om programmet väntar på input kan du behöva avbryta för att köra om efter ändring.
- Börja enkelt och bygg vidare steg för steg.
- Bygg om koden allteftersom den växer genom att införa nya abstraktioner med väl valda namn (s.k. ”refaktorisering”) .
- Fixa **alla** kompileringsfel och **alla** körtidsfel **innan** du går vidare.
- Fokusera på kodens **läsbarhet**: Om en funktion blir stor så försök dela upp den i flera funktioner med bra namn.
- En åtgärd som förbättrar läsbarheten utan att ändra hur koden fungerar ur användarens synvinkel kallas **refaktorisering** (eng. *refactoring*).
  `https://sv.wikipedia.org/wiki/Omstrukturering_av_kod`
  `https://en.wikipedia.org/wiki/Code_refactoring`

4 Objekt och inkapsling

- Vad är ett objekt?
- Singelobjekt
- Paket
- Tupler
- Fördröjd initialisering
- Funktioner är objekt
- Använda färdiga klasser
- Extensionsmetoder
- Mer om import
- Använda färdiga kodbibliotek
- Veckans övning och laboration

Denna vecka: Objekt

- Övning `objects` innehåller bland annat:
  - hur man kan kapsla in funktioner och variabler i singelobjekt
  - hur man kan skapa namnrymder, hantera namnöverskuggning, och använda punktnotation
  - använda färdiga klasser, t.ex. `java.awt.Color`
  - händelsehantering i ett grafiskt fönster
- På laboration `blockmole` lär du dig bland annat:
  - att dela upp din kod i flera singelobjekt
  - att använda färdig klass: `introprog.PixelWindow`
  - att skapa ett större program i form av ett grafiskt spel
  - använda tidigare begrepp: uttryck, program och funktioner
- Senaste versionen av kursbiblioteket `introprog` är **1.4.0**
  - Jar-fil kan laddas ned här: `https://fileadmin.cs.lth.se/introprog.jar`
  - Eller låt `scala-cli` ladda ner den med

```
//> using scala 3.5.1
//> using dep se.lth.cs::introprog:1.4.0
```

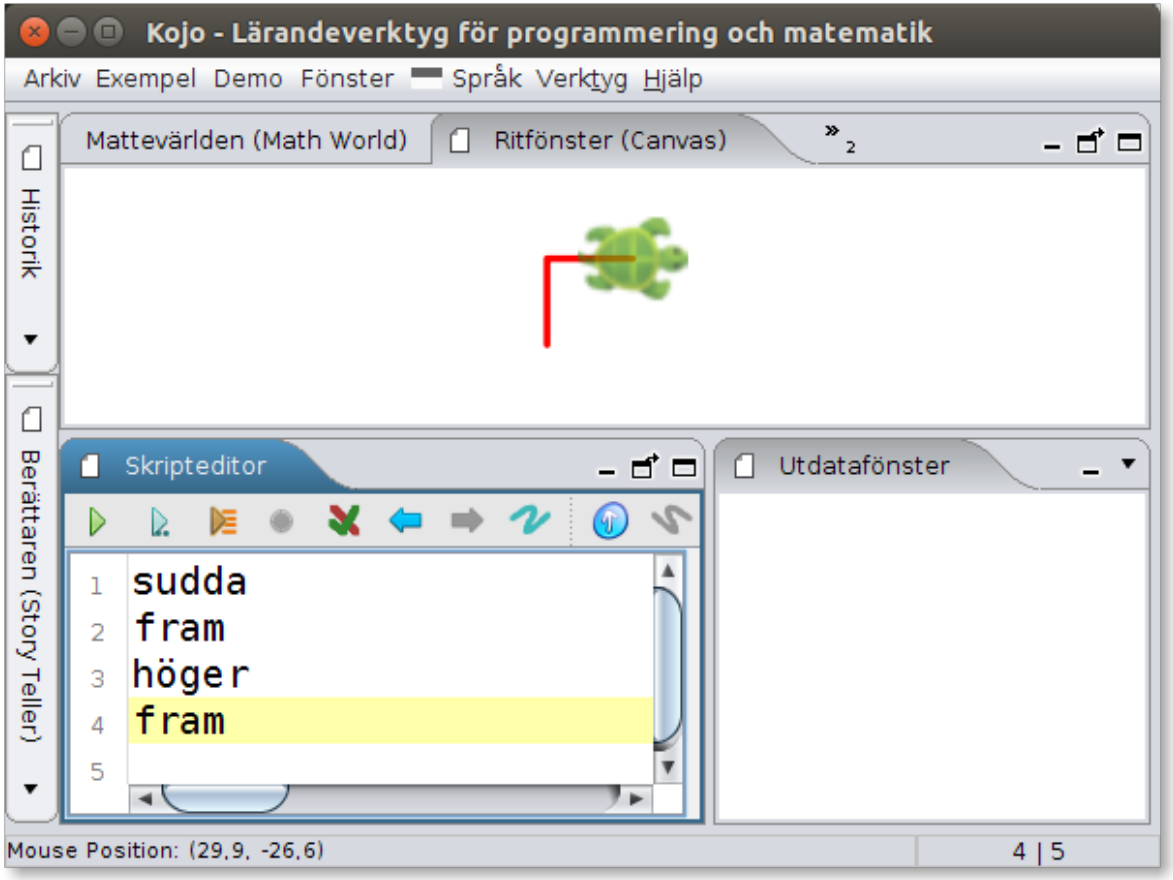
  - `https://github.com/lunduniversity/introprog-scalalib`

Vad är ett objekt?

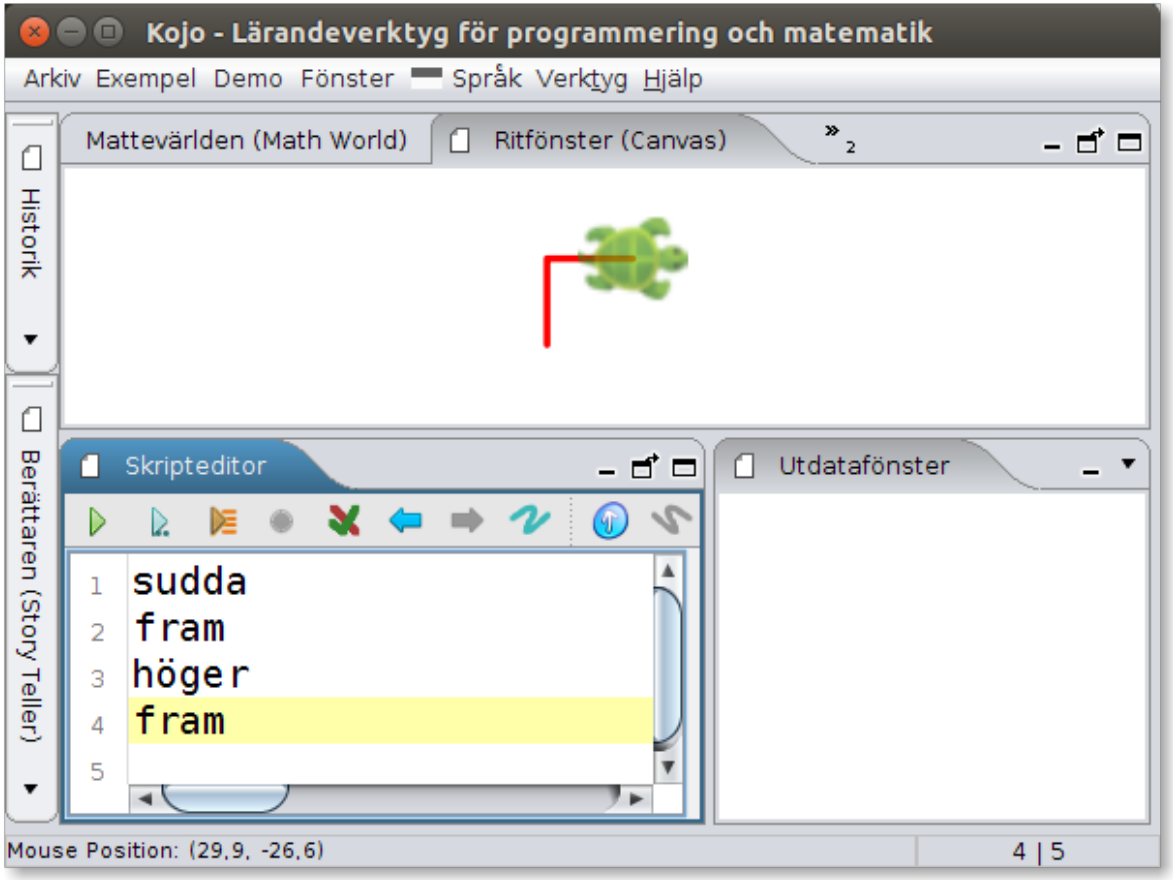
Objekt är ungefär som äggkartonger

Men de kan innehålla mer än bara ägg...

Vad rymmer sköldpaddan i Kojo i sitt tillstånd?



Vad rymmer sköldpaddan i Kojo i sitt tillstånd?



position, riktning, färg, bredd, penna uppe/nere, fyll-färg

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*

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?

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, ...

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 variablers **värden** utgör tillsammans objektets **tillstånd**
  - **operationerna** är funktioner i objektet och kallas **metoder**
  - objektets **delar** (attribut, metoder, etc.) kallas **medlemmar**

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`

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, Java, C#, m.fl tack vare **skräpsamlaren**, men i många andra språk, t.ex. C++, får man själv hålla reda på avallokering, vilket är knepigt och det blir lätt svåra buggar.

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`**.

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)`

Singelobjekt

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.

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
}
```

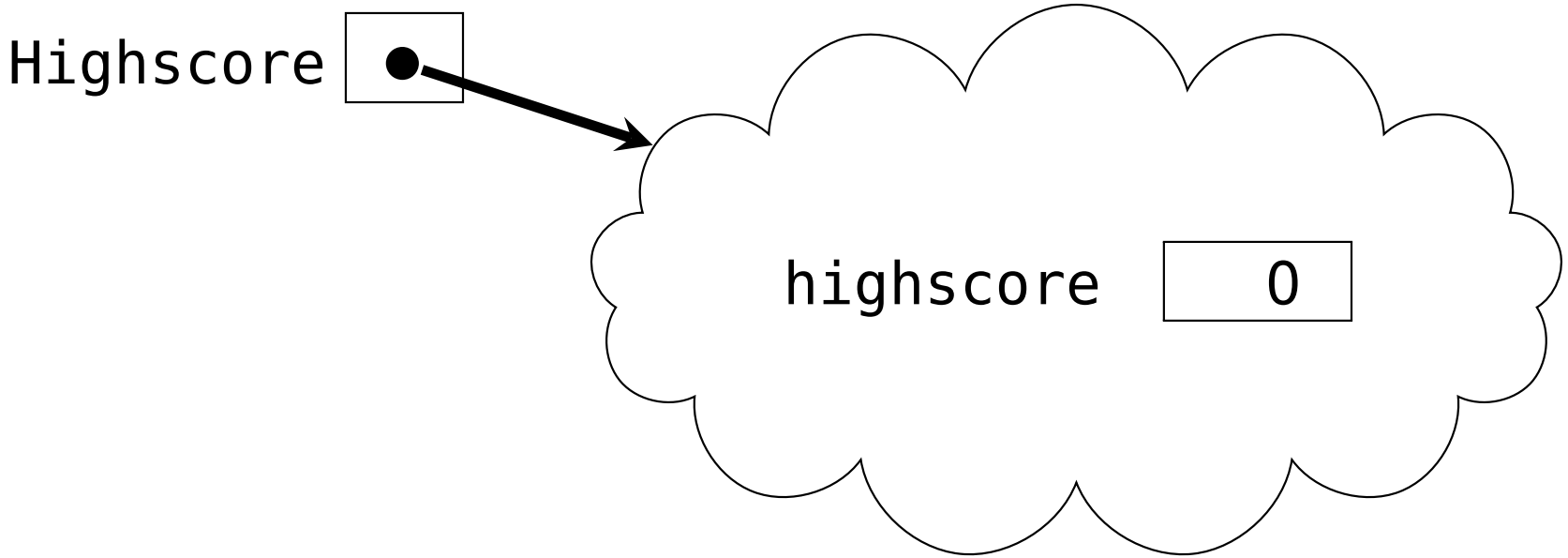
- Krullparenteser är valfria i Scala 3:
  du kan använda kolon och indentering i stället.
- Tanken är ofta att abstraktioner ska vara användbar i annan kod, för att underlätta när man bygger applikationer, och kallas då ett **API** (Application Programming Interface). Exempel: ett highscore-API.

Allokering: minne reserveras med plats för data

```
object Highscore:
  var highscore = 0
  def isHighscore(points: Int): Boolean = points > highscore
```

Allokering: minne reserveras med plats för data

```
object Highscore:
  var highscore = 0
  def isHighscore(points: Int): Boolean = points > highscore
```

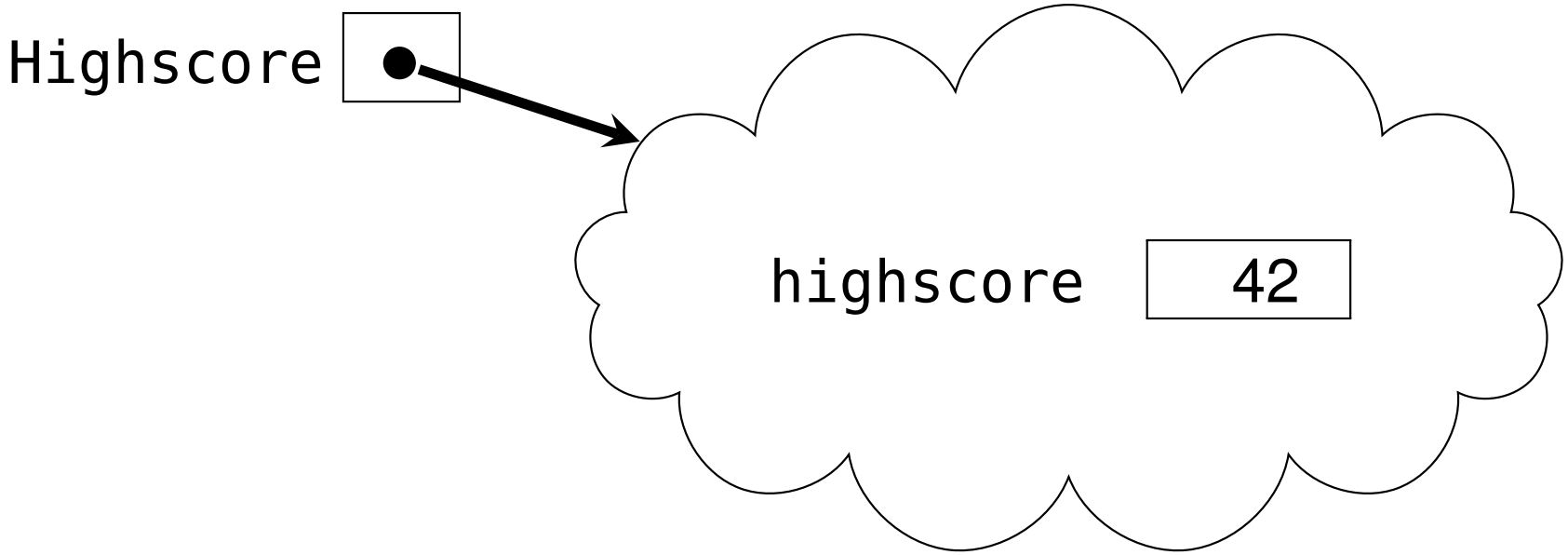


Punktnotation, tillståndsförändring med tilldelning

```
scala> Highscore.isHighscore(5)
res0: Boolean = true

scala> Highscore.highscore = 42
```

Punktnotation, tillståndsförändring med tilldelning

```
scala> Highscore.isHighscore(5)
res0: Boolean = true

scala> Highscore.highscore = 42
```



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
```

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
```

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.

Namnrymd och skuggning

- En **namnrymd**[12] (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
}
```

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

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?

Inkapsling: att dölja interna delar

Med nyckelordet **`private`** döljs interna delar för omvärlden. Privata medlemmar kan bara refereras *inifrån* objektet. Denna princip kallas **inkapsling** (eng. *encapsulation*).

```
object Highscore:
  private var myHighscore = 0        // namnet myHighscore syns ej utåt
  def highscore: Int = myHighscore   // en s.k. getter ger ett attributvärde
  def isHighscore(points: Int): Boolean = points > myHighscore
  def update(points: Int): Unit = if isHighscore(points) then myHighscore = points
```

Varför har man nytta av detta?

- Förhindra att man av misstag ändrar objekts tillstånd på fel sätt.
- Förhindra användning av kod som i framtiden kan komma att ändras.
- Erbjuder en enklare ”utsida” genom dölja komplexitet ”på insidan”.
- Inte ”skräpa ner” namnrymden med ”onödiga” namn.

Nackdelar?

Inkapsling: att dölja interna delar

Med nyckelordet **`private`** döljs interna delar för omvärlden. Privata medlemmar kan bara refereras *inifrån* objektet. Denna princip kallas **inkapsling** (eng. *encapsulation*).

```
object Highscore:
  private var myHighscore = 0        // namnet myHighscore syns ej utåt
  def highscore: Int = myHighscore   // en s.k. getter ger ett attributvärde
  def isHighscore(points: Int): Boolean = points > myHighscore
  def update(points: Int): Unit = if isHighscore(points) then myHighscore = points
```

Varför har man nytta av detta?

- Förhindra att man av misstag ändrar objekts tillstånd på fel sätt.
- Förhindra användning av kod som i framtiden kan komma att ändras.
- Erbjuder en enklare ”utsida” genom dölja komplexitet ”på insidan”.
- Inte ”skräpa ner” namnrymden med ”onödiga” namn.

Nackdelar?

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

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
```

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.

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.

Exempel: singelobjektet med förändringsbart tillstånd

```
object mittBankkonto:
  val kontonr: Long        = 123456789L
  var saldo: Int           = 1000
  def ärSkuldsatt: Boolean = saldo < 0
```

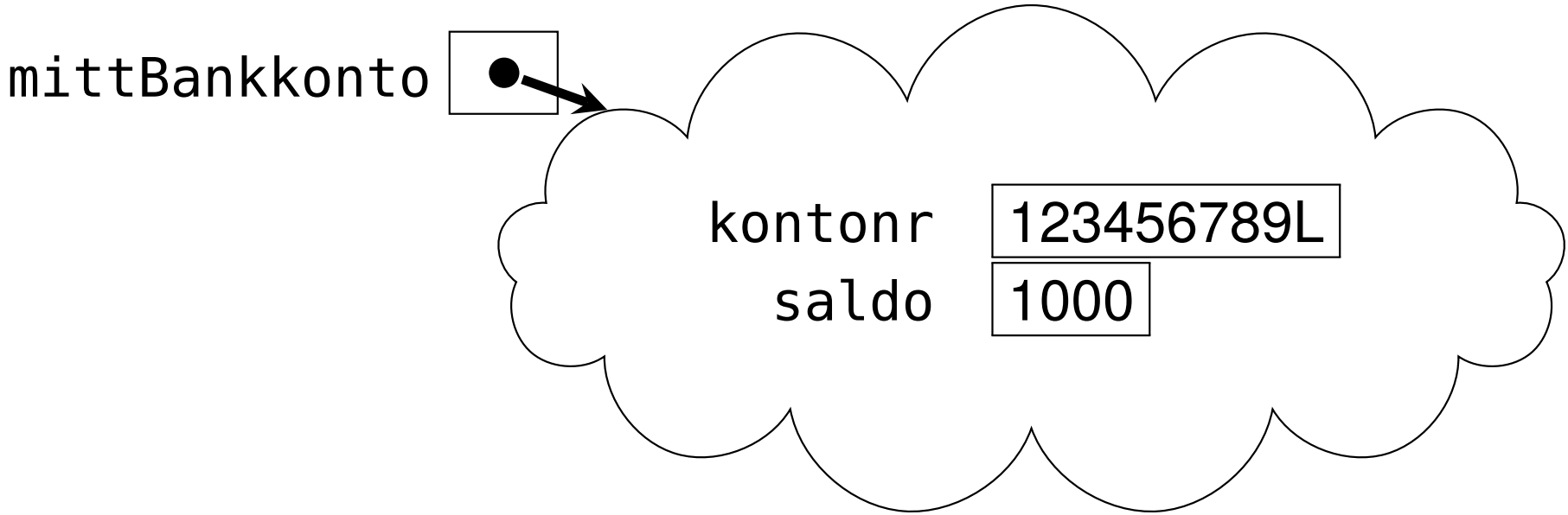
```
scala> mittBankkonto.saldo -= 25000

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.)

Exempel: tillstånd, attribut

Ett objekts **tillstånd** är den samlade uppsättningen av värden av alla de attribut som finns i objektet.

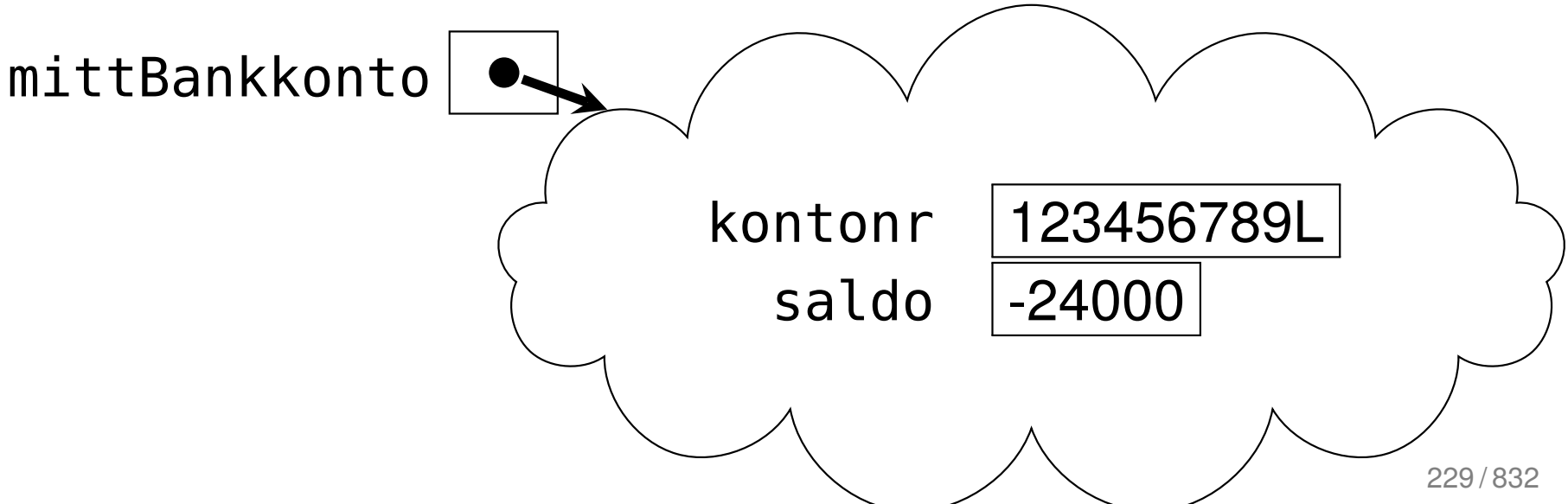
```
object mittBankkonto:
  val kontonr: Long        = 123456789L
  var saldo: Int           = 1000
  def ärSkuldsatt: Boolean = saldo < 0
```



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
```



Paket

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:[13]
  - **singelobjekt** med nyckelordet `object` och
  - **paket** med nyckelordet `package`

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

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:[13]
  - **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.

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

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:[13]
  - **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)

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

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`.

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
```

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 }}`

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
```

Tupler

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)`

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
```

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.)

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 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

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))
```

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).

Fördröjd initialisering

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, 0.156677181

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, 0.9025572
```

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).

Singelobjekt är lata

- Singelobjekt allokeras **inte** direkt vid deklaration; allokeringen sker först då objektet refereras första gången.

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?

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?

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

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
```

Med kompilator-optionen `-Wsafe-init` får du en välbehövlig varning.
Skriv såhär i din kod om du vill ha denna option påslagen:

```
//> using options -Wsafe-init

@main def run =
  val b = a
  val a = 42
  println(b)
```

```
>  scala run .
[error] a is a forward reference extending over the definition of b
[error]   val b = a
[error]           ^
```

Be kompilatorn ge fler bra varningar

Slå på mer utförliga meddelanden och varningar:

```
//> using options -unchecked -deprecation -Wunused:all -Wvalue-discard -Wsafe-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.
`-Wsafe-init`	Varna vid användning av ännu ej initialiserade variabler.

Be kompilatorn ge fler bra varningar

Slå på mer utförliga meddelanden och varningar:

```
//> using options -unchecked -deprecation -Wunused:all -Wvalue-discard -Wsafe-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.
`-Wsafe-init`	Varna vid användning av ännu ej initialiserade variabler.

Om du tycker vissa specifika varningar är irriterande kan du slå av dem med `@annotation.nowarn`

```
@annotation.nowarn
val b = a
val a = 42
```

Funktioner är objekt

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.

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.

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
```

Fördjupning: Äkta funktionsobjekt är av funktionstyp

Egentligen, mer precist:
En funktion är ett objekt av funktionstyp som har en `apply`-metod.

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...

Använda färdiga klasser

Vad är en klass?

Singelobjekt finns bara i exakt EN upplaga:

```
object mittBankkonto:
  val kontonr: Long        = 123456789L
  var saldo: Int           = 1000
  def ärSkuldsatt: Boolean = saldo < 0
```

Om vi vill ha flera bankkonton behöver vi en **klass** (eng. *class*).

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
```

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(123456789L, 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
```

Använda klassen `Color`

- I JDK (Java Development Kit) finns hundratals paket (moduler) och tusentals färdiga klasser. [14]
- 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
```

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

Använda klassen `Color`

- I JDK (Java Development Kit) finns hundratals paket (moduler) och tusentals färdiga klasser. [14]
- 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?

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

Använda klassen `Color`

- I JDK (Java Development Kit) finns hundratals paket (moduler) och tusentals färdiga klasser. [14]
- 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

[14] `https://stackoverflow.com/questions/3112882/`

Extensionsmetoder

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
```

Kollektiva extensionsmetoder

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

```
extension (s: String)
  def baklänges = s.reverse
  def skrik = s.toUpperCase
```

- Detta kallas **kollektiva extensionsmetoder** (eng. *collective extension methods*).
- Notera att det *inte* ska vara något kolon efter **`extension`**-deklarationens första rad.

Mer om import

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.

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]
```

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, *}
```

Lokal import-deklaration

- Man kan begränsa ”nedskräpningen” av namnrymden genom att göra import-deklarationer så lokalt som möjligt, till exempel i ett objekt eller i en funktionskropp.
- Exempel:

```
object A:
  def x =
    import java.awt.Color.RED
    /* ... namnet RED syns bara lokalt i denna funktion */
```

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)
```

Använda färdiga kodbibliotek

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.

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)`

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/`

Vad är *classpath*?

- Hur hittar kompilatorn färdiga moduler?

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.

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
```

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/`

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:1.4.0
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")
```

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 3.5.1
//> using dep se.lth.cs::introprog:1.4.0

@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`

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...

Veckans övning och laboration

Övning `objects`

- 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 färdiga kodbibliotek i jar-filer.
- 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.

Lab `blockmole`

- 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.

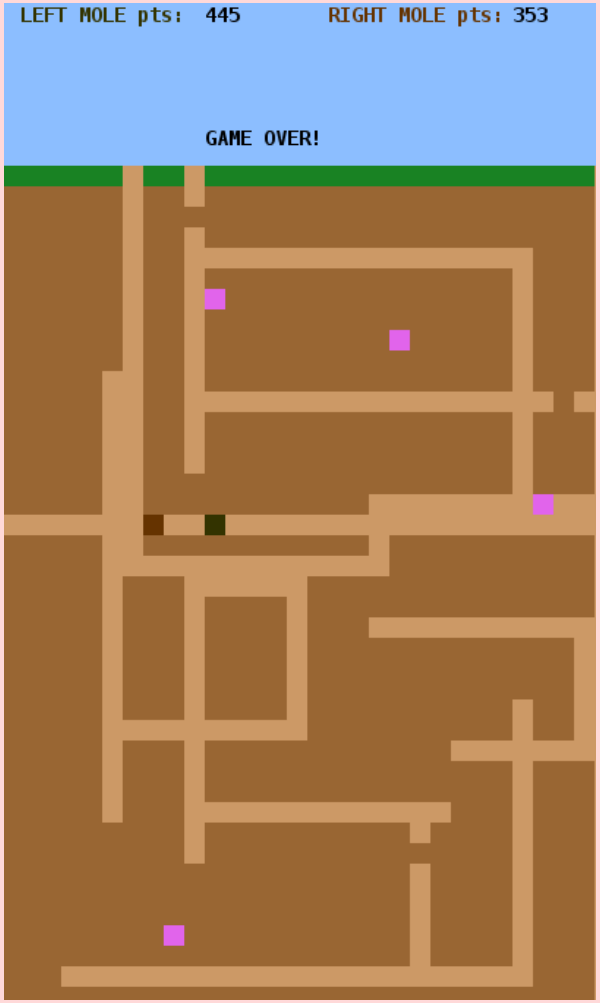
5 Klasser och datamodellering

- Vad är en klass?
- Olika sätt att skapa instanser
- Case-klasser och fördelen med oföränderlighet
- Konstruktor
- Referens saknas: `null`
- Initialisering av attribut
- Referensen `this`
- Getters och setters
- Likhet
- Implementation saknas: ???

Övning: `classes`

- 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.

Lab `blockbattle` redovisas NÄSTA läsvecka 6



- Ett arkadspel för TVÅ spelare.
- Nu med TVÅ blockmullvader!
- Programmet `blockbattle` är **mer omfattande** jmf m tidigare labbar.
- Labben omfattar TVÅ veckors förel. + övn.: klasser (w05) & mönster (w06).
- Det normala är registrering ”kompletteras” i SAM på `blockbattle0` under w05.
- Läs igenom labben innan du gör övningarna.
- Kod från övningarna behövs till labben.
- OBS! Labbtider är **schemalagda som vanligt** under läsvecka 5. Om du mot förmodan hinner redovisa redan denna vecka så ska du **ändå närvara** och t.ex. jobba med **extrauppgifter**.
- Dra nytta av undervisningen så att du lär dig så mycket som möjligt!

Vad är en klass?

En metafor för klass: Stämpel

En klass liknar en **stämpel**.

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**.

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**.

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 s.k. **`case`**-klasser och **oföränderliga** instanser.

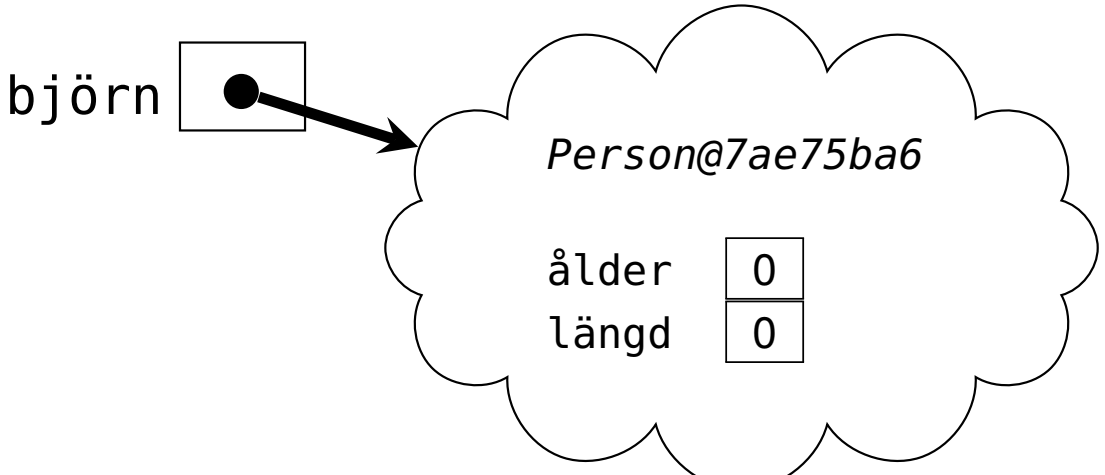
Singelobjekt jämfört med klass

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

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

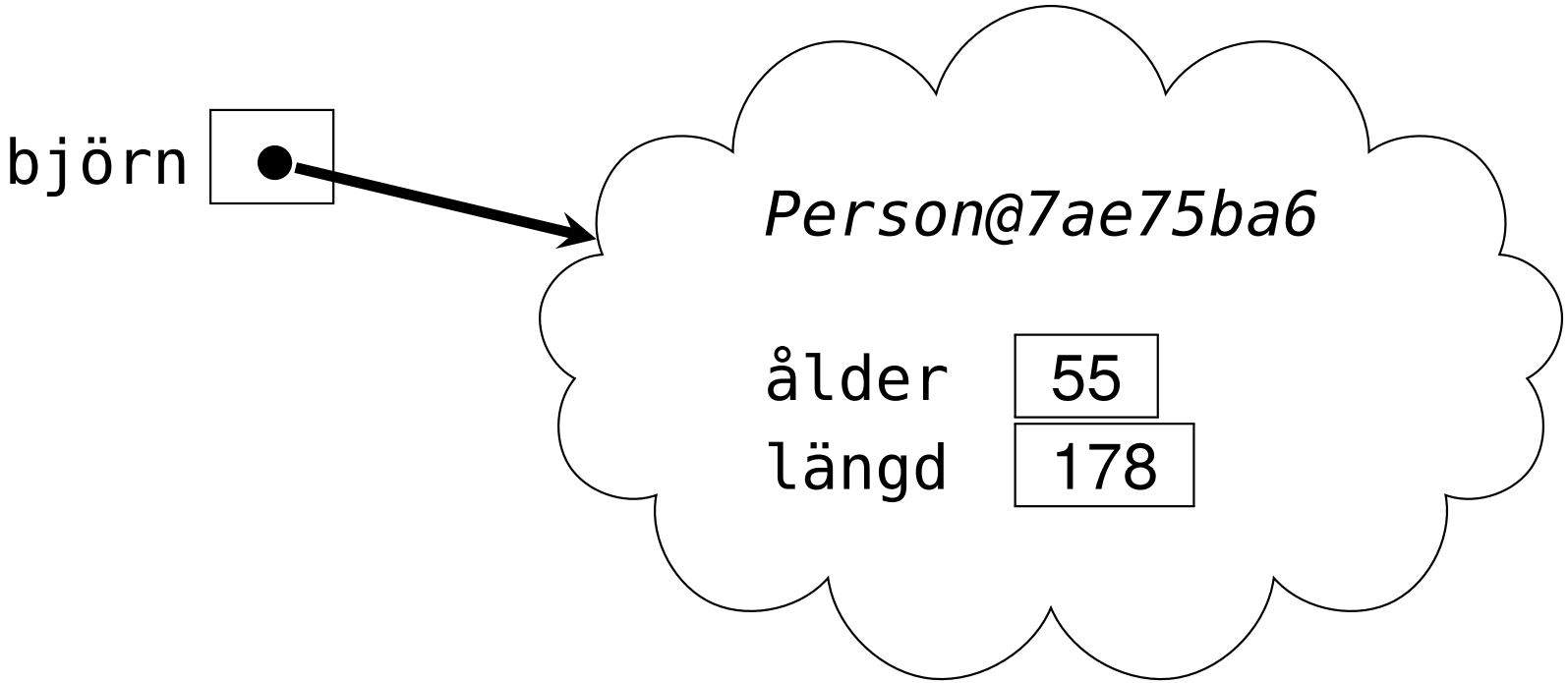
Med en **klass** kan man skapa **godtyckligt många instanser av klassen** med hjälp av nyckelordet **new** följt av klassens namn:

```
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
```



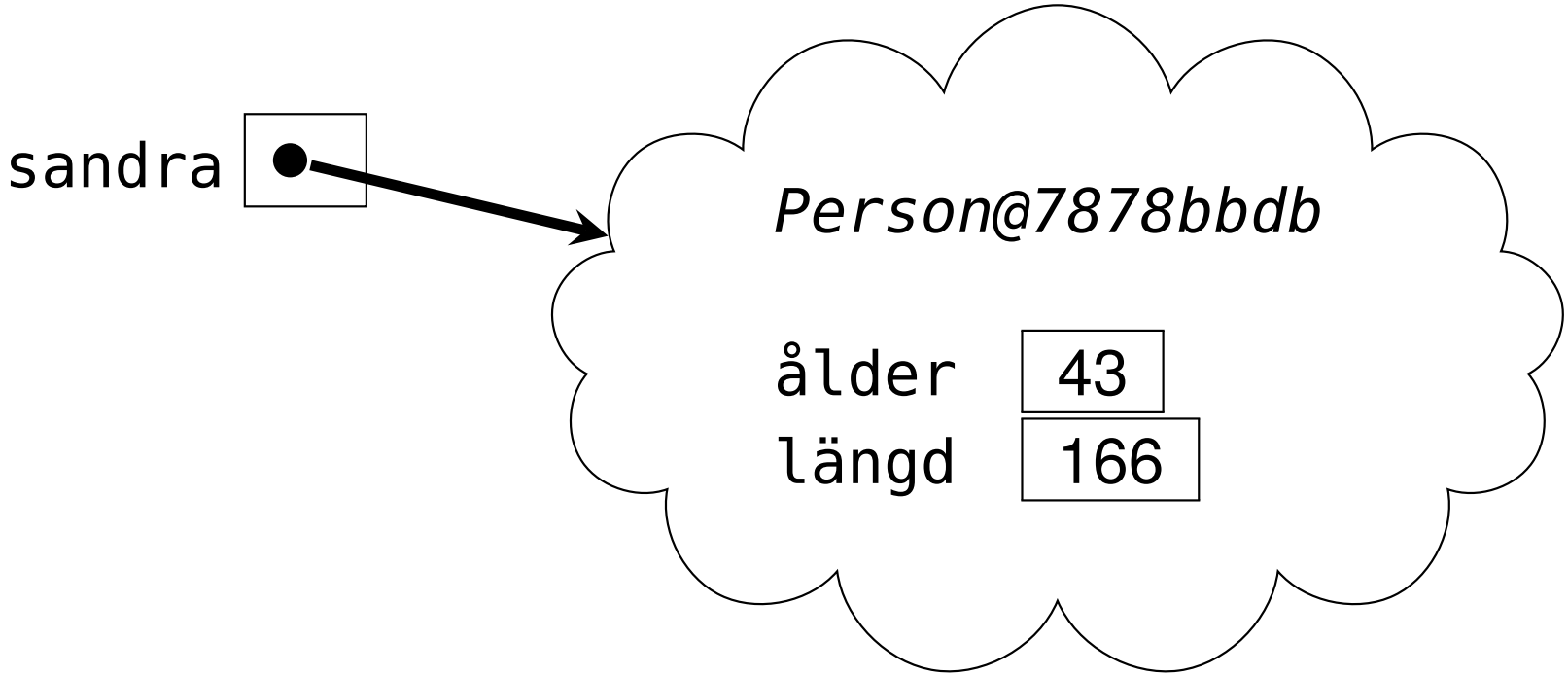
Förändring av objektets tillstånd

```
scala> björn.ålder = 55

scala> björn.längd = 178
```



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
```



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
```

- **Klass-parametrar** blir **attribut** som initialiseras med de argument som ges vid **new**. (Kompilatorn skapar **primärkonstruktor**: kod som allokerar & initialiserar alla attribut.)
- 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, de kallas då **instansprivata**.
- Attribut i klasskroppen är **publika** (alltså synliga utåt) om de inte deklareras **private** (eller **protected** som begränsar synlighet till subtyper som vi ska se senare).

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

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` ?

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`"?"`

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`).

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: det blir då **smidigt att skapa delvis förändrade kopior** där några attribut ändrats med namngivna argument och andra förblir som innan.

```
scala> p.copy(age = p.age + 1)
val res2: Person = Person(Björn,56)
```

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
```

Styra användningen av infixa alfanumeriska operatorer

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`.

Styra användningen av infixa alfanumeriska operatorer

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):
  def +(other: Box): Box = Box(x + other.x)    // utan varning
  def plus(other: Box) = Box(x + other.x)      // ger varning
  infix def add(other: Box) = Box(x + other.x) // utan varning
```

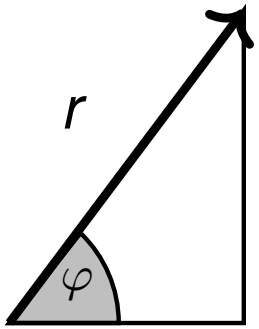
```
scala> Box(41) plus Box(1)
1 warning found
-- Warning: ----------------------------------------------------------------------
1 |Box(41) plus Box(1)
  |        ^^^^
  |Alphanumeric method plus is not declared infix; it should not be used as infix operator.
  |Instead, use method syntax .plus(...) or backticked identifier `plus`.
val res0: Box = Box(42)

scala> Box(41) add Box(1)
val res1: Box = Box(41)
```

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:
$z = 3 + 4i$



imaginär-delen är 4

real-delen är 3

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

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"
```

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.)

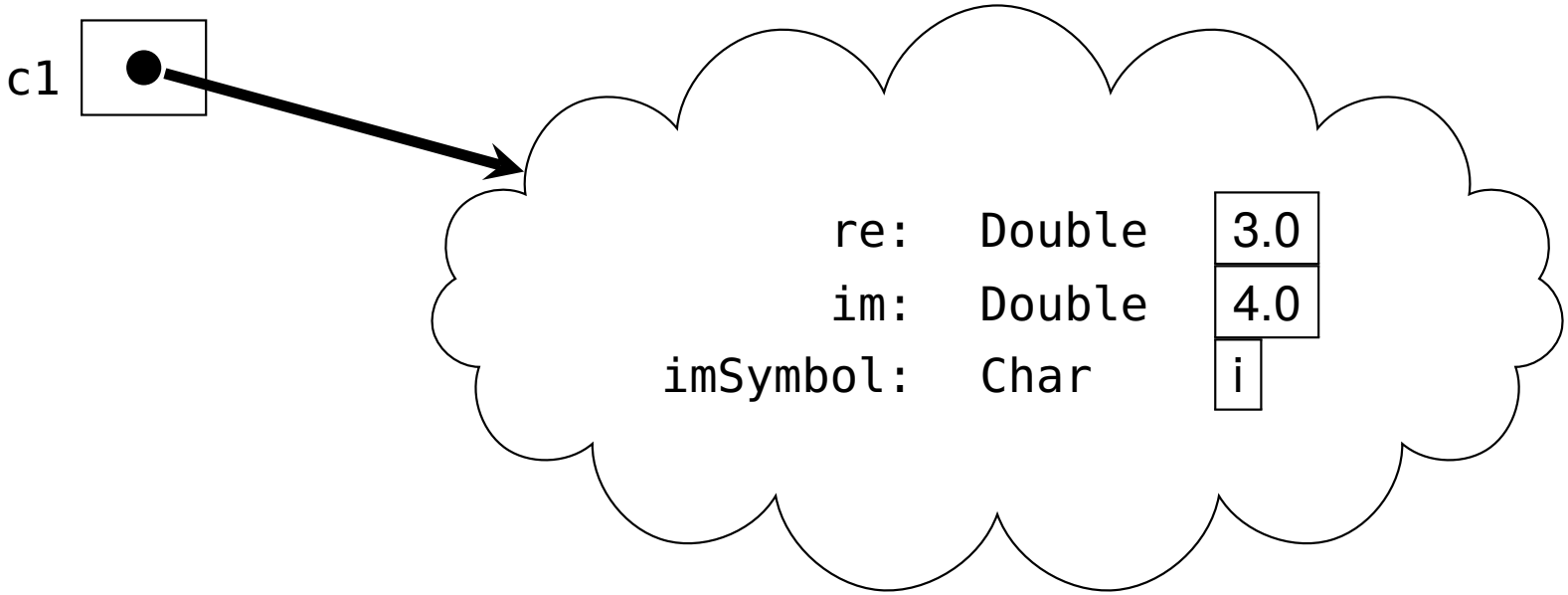
Olika sätt att skapa instanser

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)
```

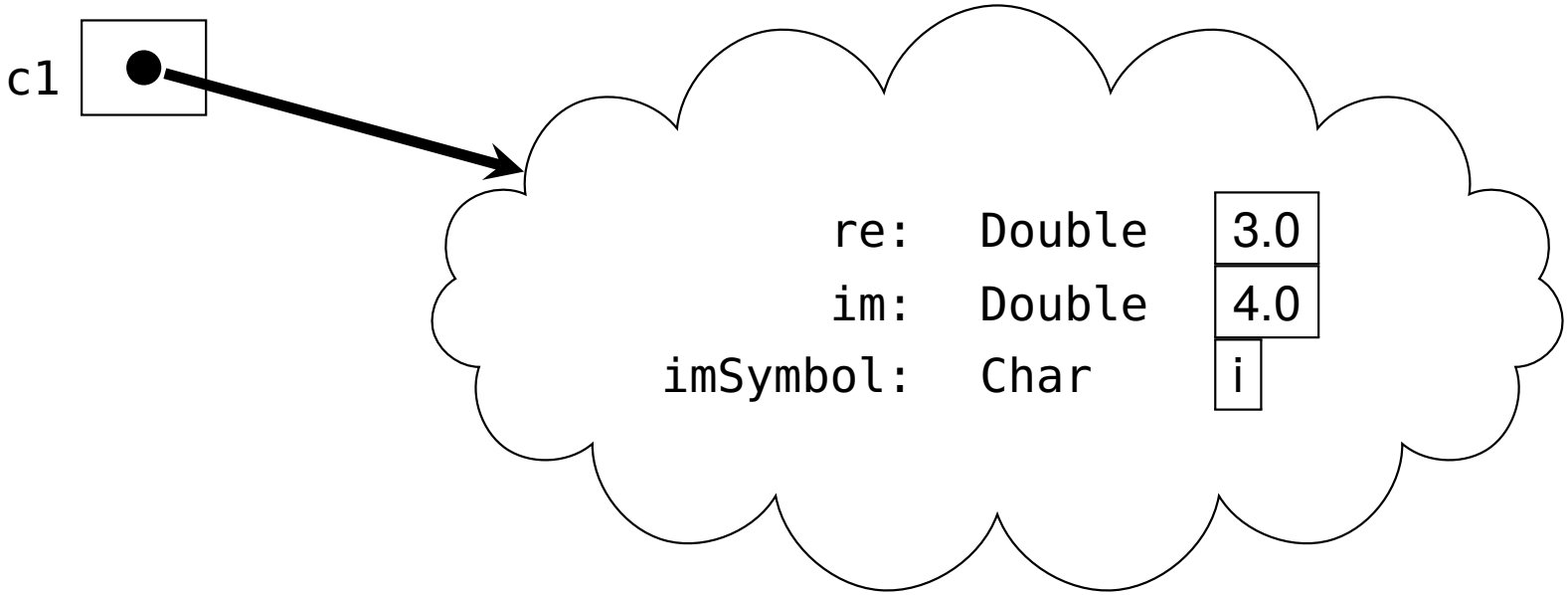


Instansiering med direkt användning av new

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.

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)
}
```

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
```

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
```

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

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
```

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
```

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()
```

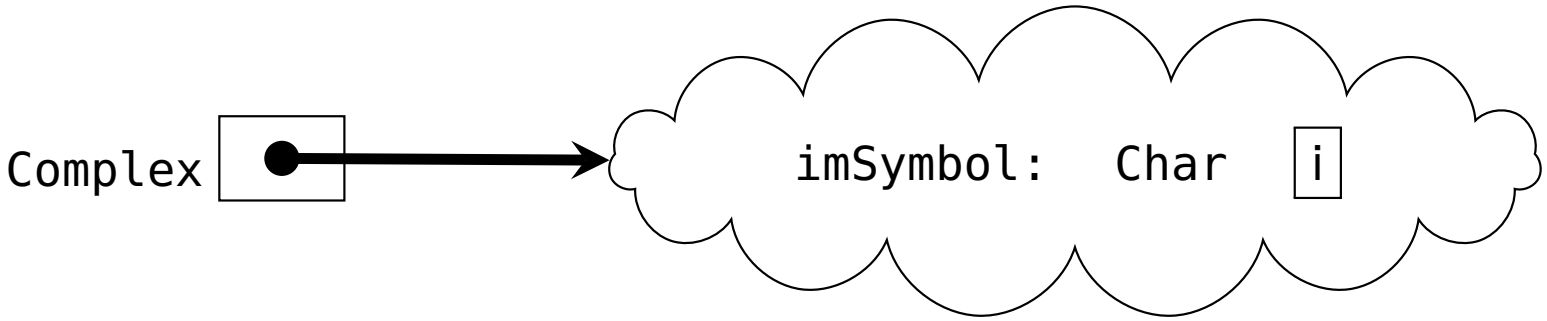
Medlemmar som bara behövs i en enda upplaga

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()
```

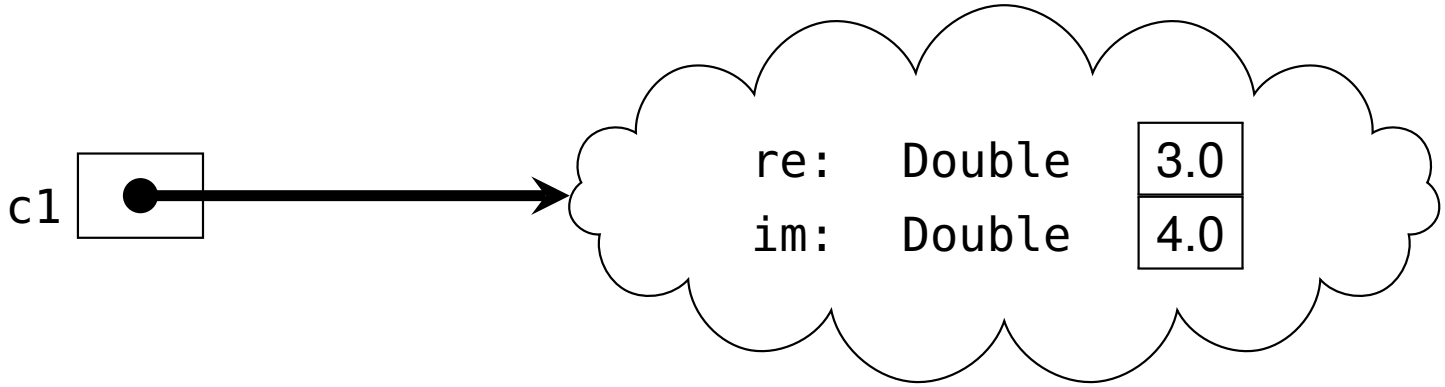
Medlemmar i singelobjekt är statiskt allokerade

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)
```

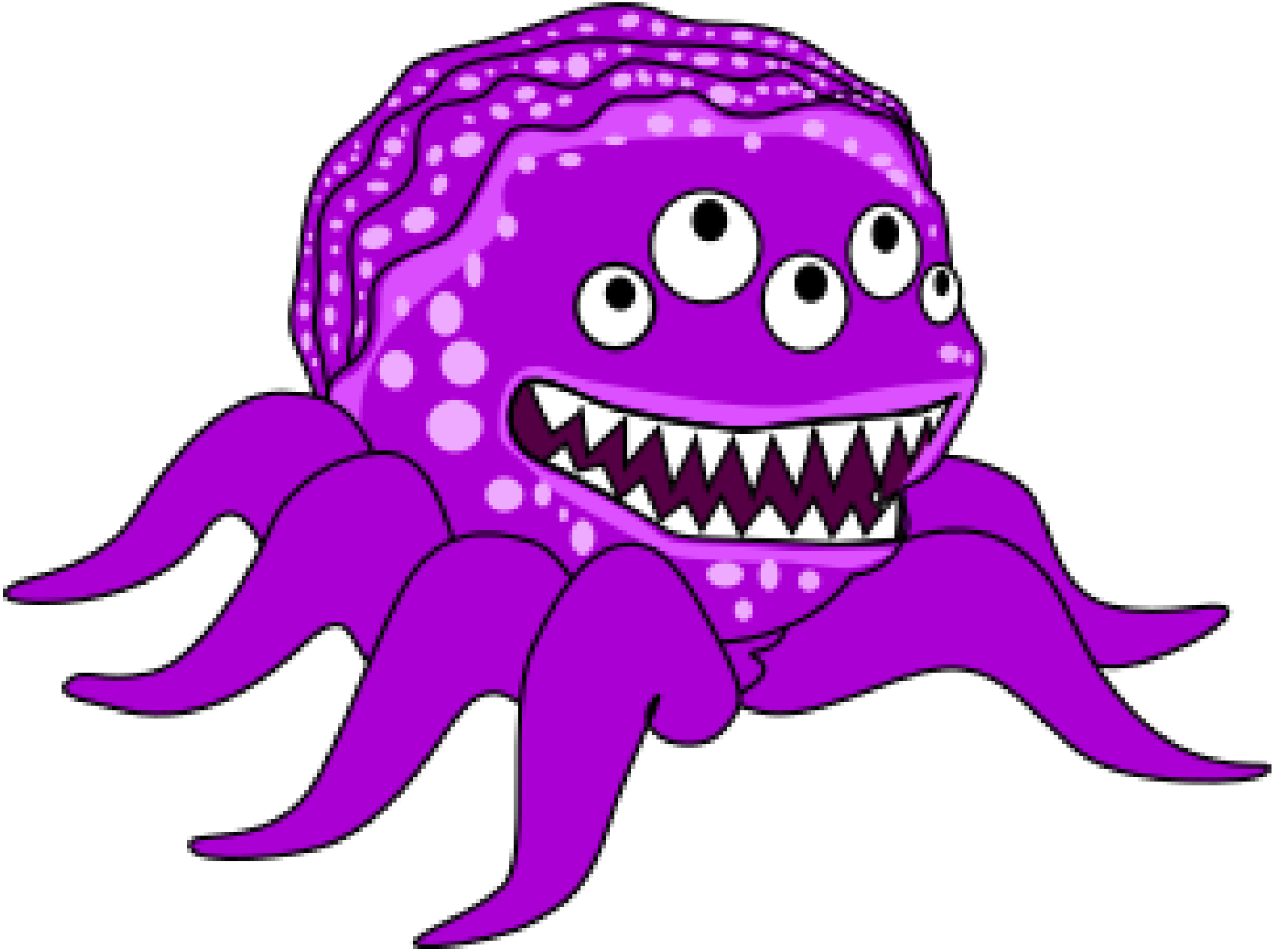


Attribut i kompanjonsobjekt användas för sådant som är gemensamt för alla instanser

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
```

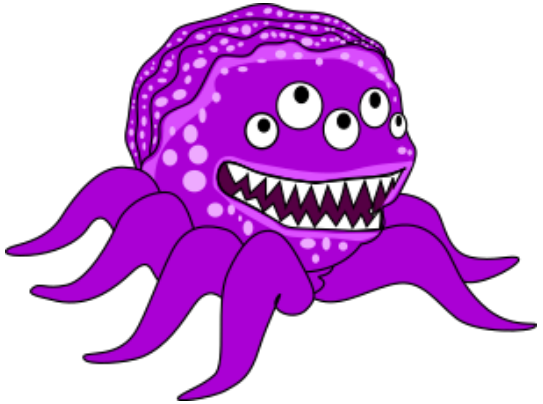
Case-klasser och fördelen med oföränderlighet



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?

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[15] 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.

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

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[15] 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...

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

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.)

Konstruktor

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.

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** (eng. *auxilliary constructor*), men det är **ovanligt**, eftersom man har möjligheten med fabriksmetoder i kompanjonsobjekt och default-argument.

Fördjupning: Hjälpkonstruktorer i Scala (ovanliga)

Fördjupning för kännedom:

- I Scala kan man skapa ett alternativ till primärkonstruktorn, en så kallad **hjälpkonstruktor** (eng. *auxilliary constructor*) genom att deklarera en metod med det speciella namnet `this`.
- Hjälpkonstruktorer **måste** börja med att anropa en **annan** konstruktor som står **före** i koden, till exempel primärkonstruktorn.

```
class Point(val x: Int, val y: Int, val z: Int): // primärkonstruktor
  def this(x: Int, y: Int) = this(x, y, 0) // anropa primärkonstruktorn
  def this(x: Int) = this(x, 0) // anropa hjälpkonstruktor
```

- Varför? Enklare att använda från **Java**-kod jämfört med apply i kompanjonsobjekt. (Men om din Scala-kod inte ska användas från Java så är detta onödigt: använd då hellre kompanjonsobjekt med fabriksmetod.)

Fördjupning: 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
```

Fördjupning: 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:

```scala
case class Point(x: Int, y: Int = 0, z: Int = 0)
```

Använd alltså hellre

- defaultargument i klassparametrar, eller
- fabriksmetoder i kompanjonsobjekt, antingen med default-argument eller överlagrade.

Referens saknas: `null`

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.

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 kompatibilitetsskäl, men det är brukligt att **endast** använda **`null`** om man anropar Java-kod.
- Scala erbjuder smidiga `Option`, `Some` och `None` för säker hantering av saknade värden; mer om detta kommande vecka.

Initialisering av attribut

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`**

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?

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?

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: `-Wsafe-init`

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: `-Wsafe-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...)

Referensen `this`

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.

Getters och setters

Getters och setters

- I många språk (t.ex. Java, Python) finns inget motsvarande nyckelord `val` som garanterar oföränderliga attributreferenser. [16]
- 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.

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

Java-exempel: Klassen JPerson

Indirekt access av **privata** attribut:

```java
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@7e774085

scala> p.getAge
val res0: Int = 0

scala> p.setAge(42)

scala> p.getAge
val res1: Int = 42

scala> p.age
-- Error:
p.age
^^^^^
value age is not a member of JPerson
```

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**.

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**?

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! :(

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** _=

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
```

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
```

Likhet

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**.

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.

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 `==`

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
```

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
```

Case-klasser ger innehållslikhet

Förutom annat ”godis på köpet” får du med `case class` även detta:

- Metoden == ger **innehållslikhet** (och inte referenslikhet).

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
```

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*)

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).

Implementation saknas: ???

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**.

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
```

Exempel: ofärdig kod

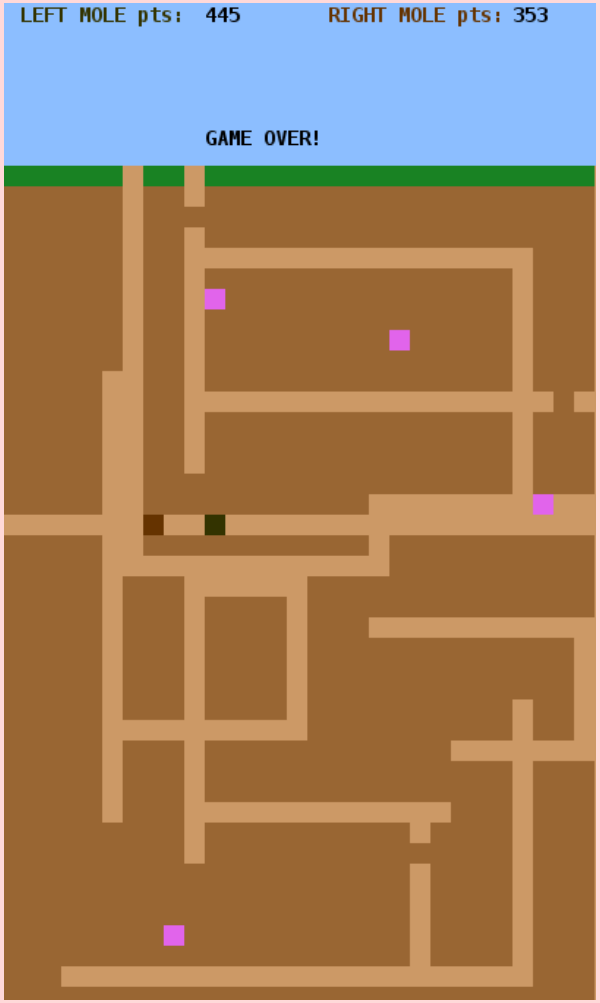
```
case class Person(name: String, age: Int):
  def ärTonåring = age >= 13 && age <= 19
  def ärUng = !ärGammal
  def ärGammal: Boolean = ???   //implementation ännu ej klar
```

```
scala> Person("Björn", 51).ärTonåring
res23: Boolean = false

scala> Person("Sandra", 39).ärUng
scala.NotImplementedError: an implementation is missing
```

Övning: `classes`

- 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.

Lab `blockbattle` redovisas NÄSTA läsvecka 6



- Ett arkadspel för TVÅ spelare.
- Nu med TVÅ blockmullvader!
- Programmet `blockbattle` är **mer omfattande** jmf m tidigare labbar.
- Labben omfattar TVÅ veckors förel. + övn.: klasser (w05) & mönster (w06).
- Det normala är registrering ”kompletteras” i SAM på `blockbattle0` under w05.
- Läs igenom labben innan du gör övningarna.
- Kod från övningarna behövs till labben.
- OBS! Labbtider är **schemalagda som vanligt** under läsvecka 5. Om du mot förmodan hinner redovisa redan denna vecka så ska du **ändå närvara** och t.ex. jobba med **extrauppgifter**.
- Dra nytta av undervisningen så att du lär dig så mycket som möjligt!

6 Mönster och felhantering

- Typhierarkier i Scala
- Mönstermatchning
- Hantera speciella värden med inkapsling
- Hantera saknade värden med `Option`
- Undantag
- Hantera undantag med `Try`
- Hantera undantag med `try-catch`
- För- och nackdelar med undantag
- Fördjupning: Implementera `equals`
- Veckans uppgifter

Typhierarkier i Scala

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 `==`

Bastypen för alla typer: Any

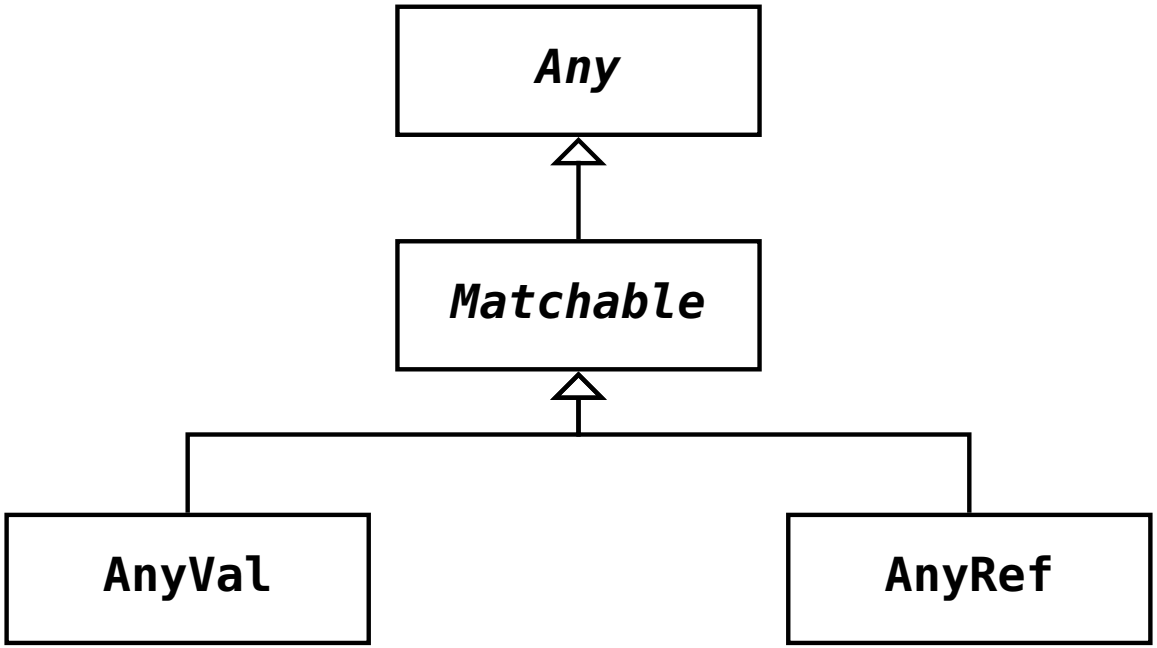
Scalas typsystem är **fullständigt**:

- 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: String    // en strängrepresentation
  def equals(other: Any): Boolean = eq(other)
  def hashCode: Int       // ska vara samma som andras om equals true

  // finala metoder får ej överskuggas:
  final def eq(other: Any): Boolean  // ger alltid referenslikhet
  final def ne(other: Any): Boolean = !eq(other)
  final def ==(other: Any) = equals(other)
  final def !=(other: Any) = !equals(other)
  final def isInstanceOf[T]: Boolean  // typtest vid körtid
  final def asInstanceOf[T]: T  // osäker typkonvertering vid körtid
  final def ## = hashCode  // specialbehandlas för värdetyper i JVM
```

(Mer om bastyper, traits, equals, hashCode, ... senare.)

Alla typer är subtyper till Any

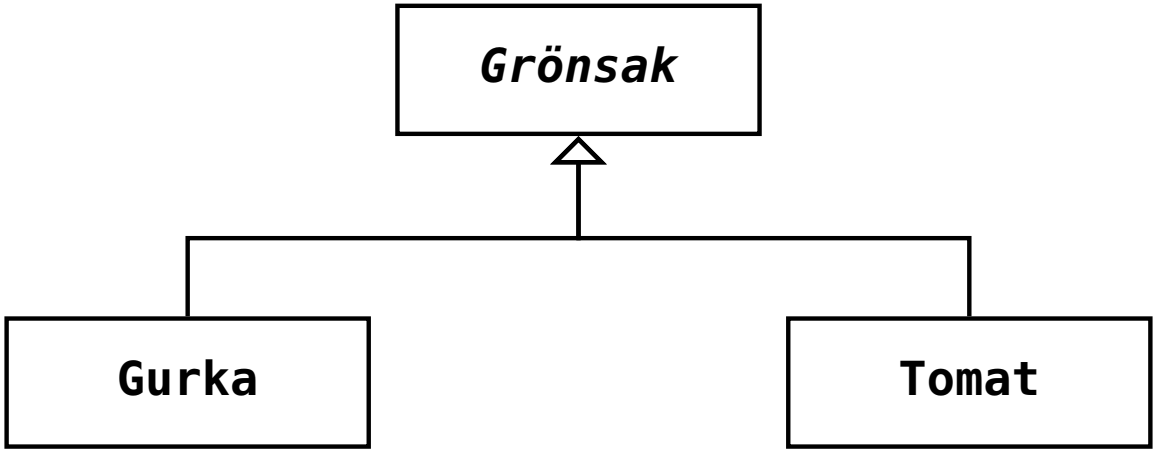


- Alla **värdetyper**, t.ex. `Int`, `Double`, `Boolean`, är subtyper till `AnyVal`
- Alla **referenstyper** t.ex. `String` är subtyper till `AnyRef`
- Värden av typen `Matchable` kan användas vid s.k mönstermatchning.
- (Det går att skapa s.k. opaka typer som inte kan mönstermatchas.)

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.

Mönstermatchning

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.

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)
```

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
```

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.

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)
```

Matchning är ungefär som att passa klossar i en låda

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..."
```

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.

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

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.

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"`.

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"
```

Matchning med typade mönster

Antag att vi har nedan oäkta funktion `f` som vi vill matcha på:

```
def f() =   // Vilken returtyp härleds av kompilatorn?
  if math.random() < 0.5 then 42 + math.random()
  else s"gurka ${math.random()}"
```

Matchning med typade mönster

Antag att vi har nedan oäkta funktion `f` som vi vill matcha på:

```
def f() =   // Vilken returtyp härleds av kompilatorn?
  if math.random() < 0.5 then 42 + math.random()
  else s"gurka ${math.random()}"
```

Med en typannotering efter en variabel får man ett **typat mönster** (eng. *typed pattern*). Vid lyckad matchning **omvandlas** värdet till den specifika typen och binds till variabeln.

```
val i: Int = f() match
  case x: Double => x.round.toInt     // typat mönster som kollar om Double
  case s: String => s.length          // typat mönster som kollar om String
```

Matchning med typade mönster

Antag att vi har nedan oäkta funktion `f` som vi vill matcha på:

```
def f() =   // Vilken returtyp härleds av kompilatorn?
  if math.random() < 0.5 then 42 + math.random()
  else s"gurka ${math.random()}"
```

Med en typannotering efter en variabel får man ett **typat mönster** (eng. *typed pattern*). Vid lyckad matchning **omvandlas** värdet till den specifika typen och binds till variabeln.

```
val i: Int = f() match
  case x: Double => x.round.toInt     // typat mönster som kollar om Double
  case s: String => s.length          // typat mönster som kollar om String
```

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)
```

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`

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
```

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.

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"
```

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?

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`

Matchning på tupler

Det går fint att plocka isär tupler med mönstermatchning:[17]

```
var pair = ("hej", 42)

pair match
  case (a, b) if b == 42 => s"livets mening är funnen: $a"
  case (_, b)            => s"fattas mening: $b"
```

Understreck betyder att vi inte är intresserade av att binda ett variabelnamn till värdet.

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

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 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?

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 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 : (

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

Varning vid kompilering :) Tack kompilatorn!

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
```

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
```

```
1 warning found
  |  f match
  |  ^
  |  match may not be exhaustive.
  |
  |  It would fail on pattern case: Ruter
```

Även här får vi hjälpsam varning vid kompilering :)

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
```

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
```

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: ???
val y: ???

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

Mönster i **for-uttryck**:

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

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)
```

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

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 **`case`** utan **`match`**.

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?

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`

Hantera speciella värden med inkapsling

Inkapsling av speciella värden så att krasch kan undvikas

Hantera saknade värden med `Option`

Hur hantera saknade värden?

Olika sätt att hantera saknade värden:

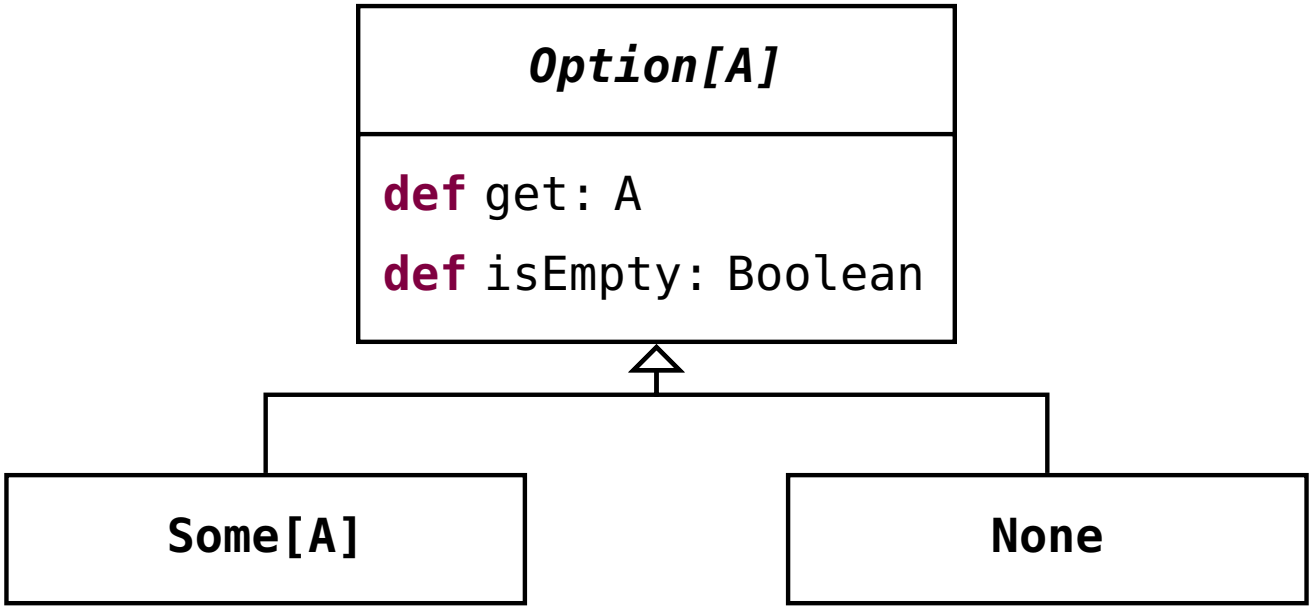
- Hitta på ett specialvärde: exempel -1 för saknat värde
- `null` om värde saknas (vanligt i Java m.fl. språk, mkt ovanligt i Scala)
- Använd en samling och låt tom samling representera saknat värde:
  ```
  val sums = Vector(Vector(42),Vector(32),Vector(),Vector(21))
  ```

Hur hantera saknade värden?

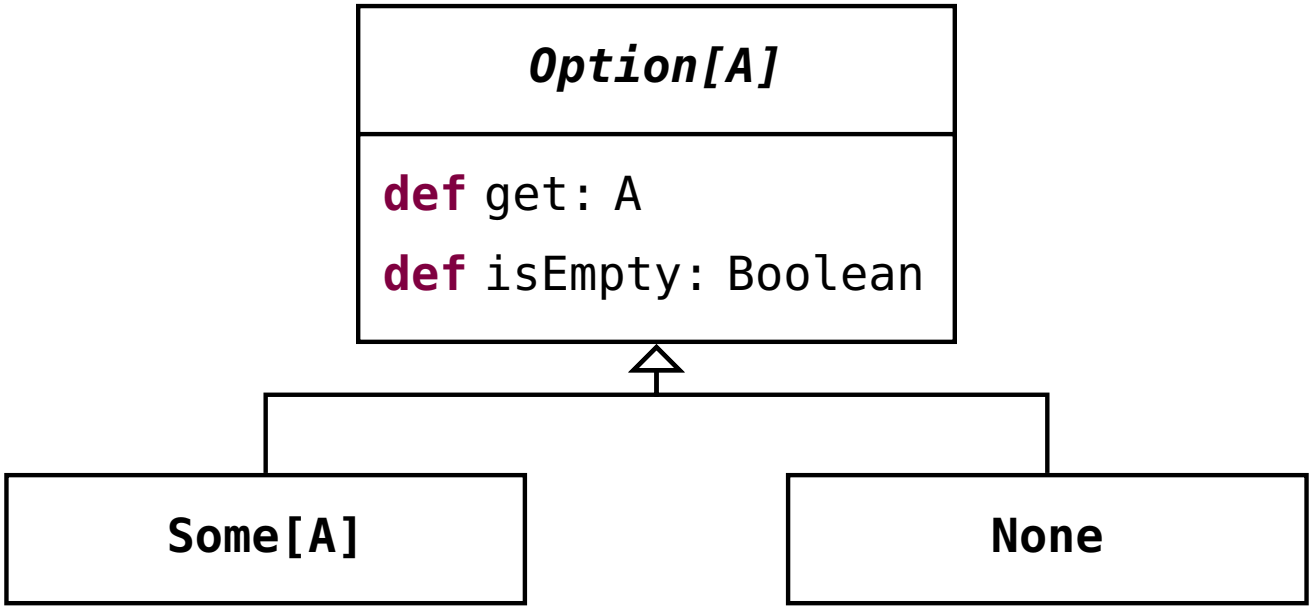
Olika sätt att hantera saknade värden:

- 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**

En gemensam bastyp för ett värde som kanske saknas



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
```

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])
```

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
```

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
```

Några samlingsmetoder som ger en `Option`, övning

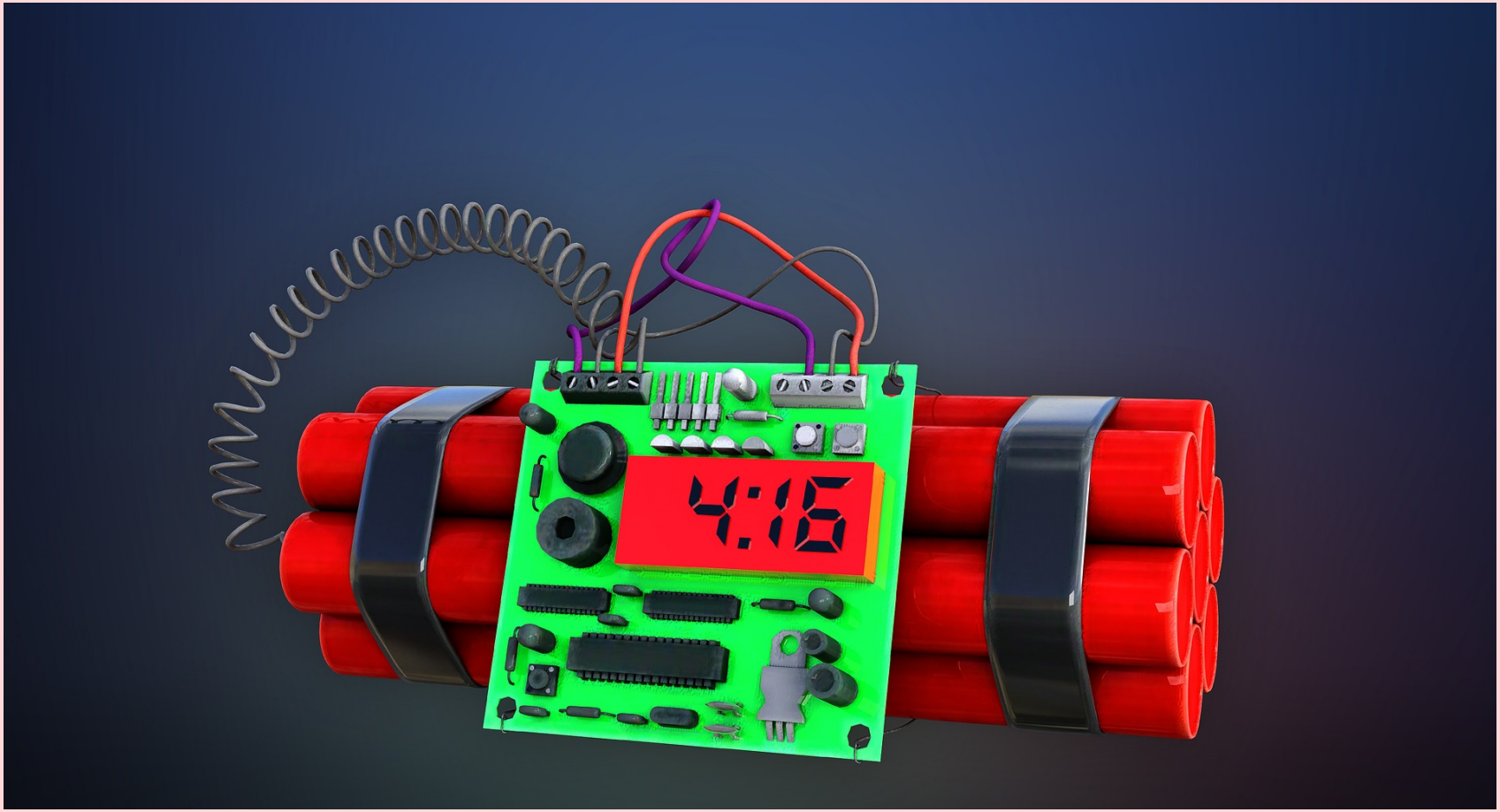
```
scala> val (xs, ys) = (Vector(1,2,3), Vector())

scala> xs.headOption
???

scala> ys.headOption
???

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

scala> xs.find(_ > 5)
???

scala> (xs.lift(0), ys.lift(0))
???

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

scala> huvudstad.get("Skåne")
???

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

Några samlingsmetoder som ger en `Option`, svar

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

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

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

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

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

scala> (xs.lift(0), ys.lift(0))
val res4: (Option[Int], Option[Nothing]) = (Some(1),None)

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

scala> huvudstad.get("Skåne")
val res5: Option[String] = Some(Malmö)

scala> huvudstad.get("Danmark")
val res6: Option[String] = None
```

Undantag

Undantag kan orsaka krasch...



Undantag orsakar ingen krasch om inkapslad i en Try

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:

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`

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`

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!
```

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!
```

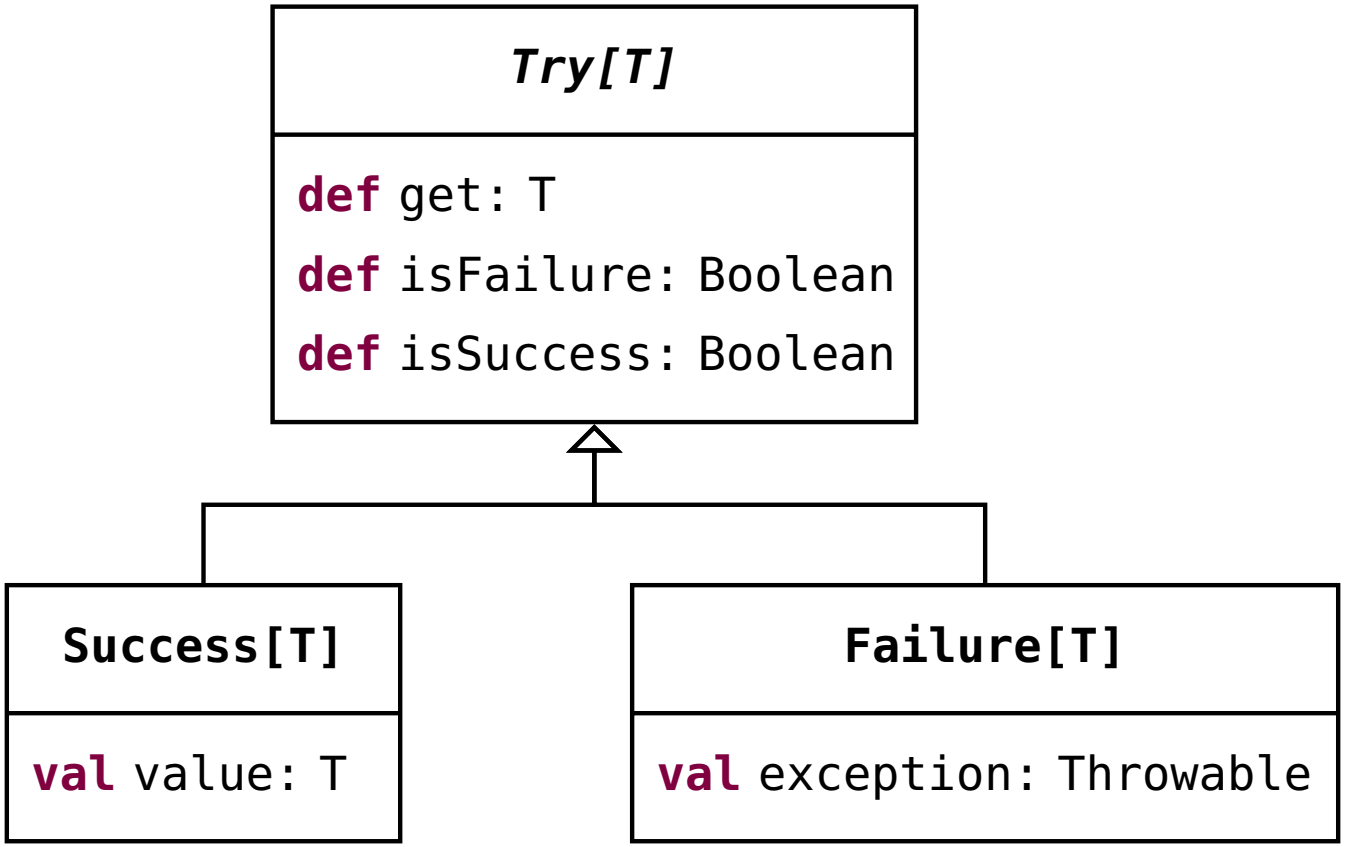
Olika sätt att hantera undantag och förhindra att exekveringen avbryts:

- **`try catch`**-uttryck omvandlar undantag till ngt lämpligt värde.
- `scala.util.Try` **kapslar in** kod som kan ge undantag.

Hantera undantag med `Try`

En gemensam bastyp för något som kan misslyckas

```
import scala.util.{Try, Success, Failure}
```



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
```

Hantera undantag med `try-catch`

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
```

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)
```

För- och nackdelar med undantag

Undvik 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.

Undvik undantag om det går

Fördelar med undantag:

- Vid allvarliga fel då det inte är mycket att göra än att starta om, t.ex. `OutOfMemoryException`, är det bra att få veta vad som är fel.
- Onormala fall som uppkommer sällan kan hanteras separat (t.ex. i huvudprogrammet) utan att koden för normalfallet blir tillkrånglad.

Nackdelar med undantag:

- Ett slags ”goto” som gör exekveringsflödet svårt att följa.
- Skapa stack-trace tar tid; undantag som sker ofta påverkar prestanda.

Exempel: undantagslösa `toIntOption` är både säker och snabb!

```
scala> def time(op: => Unit): Long = {val t0 = System.nanoTime; op; System.nanoTime - t0}

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

scala> min(util.Try("hello".toInt))
val res0: Long = 3549

scala> min(try "hello".toInt catch (_: Throwable) => ())
val res1: Long = 3046

scala> min("hello".toIntOption)
val res2: Long = 157
```

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`

Fördjupning: Implementera `equals`

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...

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?).

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
```

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

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`

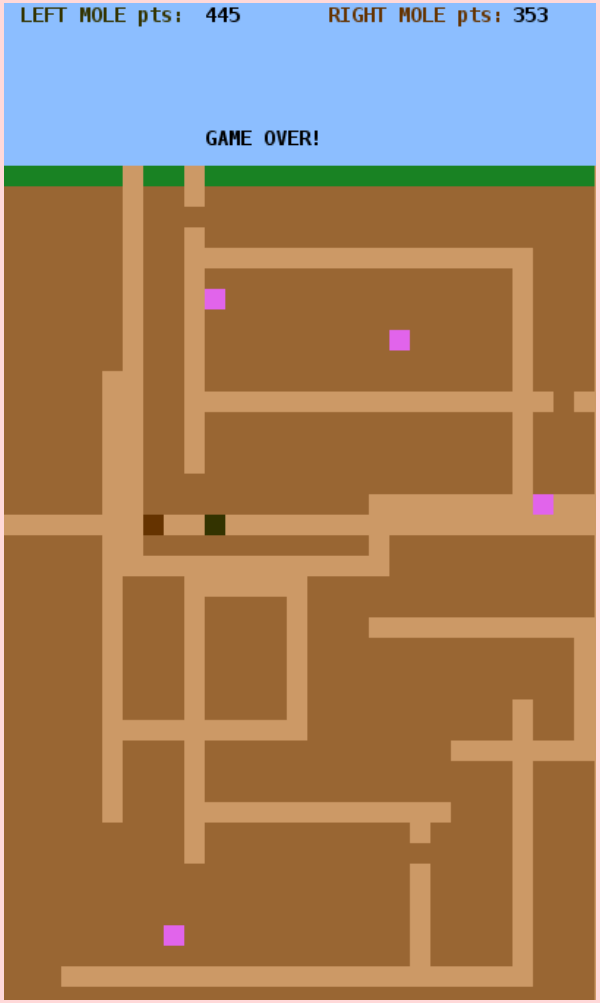
Veckans uppgifter

Veckans övning: `patterns`

Mål: Träna på matchning (och undantag)

- Uppg. 1–7: Matchning, `Option`
- Uppg. 8–10: Undantag, `Try`
- Fördjupn. 11: Matchning eller dynamisk bindning?
- Fördjupn. 12: avgöra likhet med matchning, `equals` utan arv
- Fördjupn. 13–23: diverse fördjupningsuppgifter om matchning, undantag, hash-koder, likhet vid arvshierarki

Lab `blockbattle` läsvecka 5–6



Tips på hur du kan träna på att använda mönstermatchning:

- Händelsehantering med **`match`** i stället för nästlade **`if`**
- Ändra så att `waitForKeyNonBlocking` returnerar `Option[String]`
- Låt metoderna `setBlock` och `getBlock` genererar undantag om position är utanför fönstret. Använd `require` för att åstadkomma detta.
- Använd `Try` när du anropar metoder som kan ge undantag och undvik att din app kraschar, men ge bra felmeddelande som underlättar avlusning.

7 Sekvenser och enumerationer

- Vad är en sekvens?
- Vad är en sekvensalgoritm?
- Använda färdiga sekvenssamlingsmetoder
- Repeterade parametrar
- Enumerationer
- Registrering
- Skapa lösningar på sekvensproblem från grunden
- Implementera insert, remove, append
- Exempel: Polygon
- Jämföra strängar
- Sökning och sortering
- Uppgifter denna vecka
- Kontrollskrivning

Vad är en sekvens?

ORDNINGEN SPELAR ROLL

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).

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)
```

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
```

Exempel: En sträng är en sekvens av tecken

```
scala> "haj po daj"
```

Längd? Vad ligger på första platsen? Elementtyp? Dubbletter?

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
```

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)
```

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> xs :+ 42          // ger ny Vector(1, 2, 3, 42)
scala> 42 +: xs          // ger ny Vector(42, 1, 2, 3)
```

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> xs :+ 42          // ger ny Vector(1, 2, 3, 42)
scala> 42 +: xs          // ger ny Vector(42, 1, 2, 3)
```

- 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)
```

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> xs :+ 42          // ger ny Vector(1, 2, 3, 42)
scala> 42 +: xs          // ger ny Vector(42, 1, 2, 3)
```

- 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`

Egenskaper hos några sekvenssamlingar i Scala

- `Vector`
  - **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
  - Allsidig prestanda: **bra till det mesta**.
- `List`
  - **Oföränderlig**. Snabbt att skapa kopior med uppdatering **i början**.
  - Snabbt jobba **i början**, men **långsamt** jobba i **slutet** av listan.
  - 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.

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 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)`

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
```

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**.

Vad är en sekvensalgoritm?

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** eller **lösningen**.
- Exempelproblem: sortera en sekvens av personer efter deras ålder.

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** 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

ORDNINGEN
SPELAR
ROLL
KUT FSSR

Använda färdiga sekvenssamlingsmetoder

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 **quickref** för att se vad man kan göra med färdiga samlingar.
- Det är **lärorikt** att ”**uppfinna hjulet**” och implementera några grundläggande sekvensalgoritmer **själv** för bättre förståelse, även om de redan finns färdiga i Scalas samlingsbibliotek.
- Fördjupning: En översikt av samlingarna i Scalas standardbibliotek: `https://docs.scala-lang.org/overviews/collections-2.13/introduction.html`
- (Det kommer mer om implementation av samlingar och algoritmer i fördjupningskursen pfk.)

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

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)
```

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
```

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
```

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)`

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)
```

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)
```

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
```

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.

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 eller färdiga samlingsmetoder?

Tips: För `sort&freq` se `sorted`, `distinct`, `count` i quickref

Implementation av sekvensproblem med `for`-uttryck eller färdiga samlingsmetoder

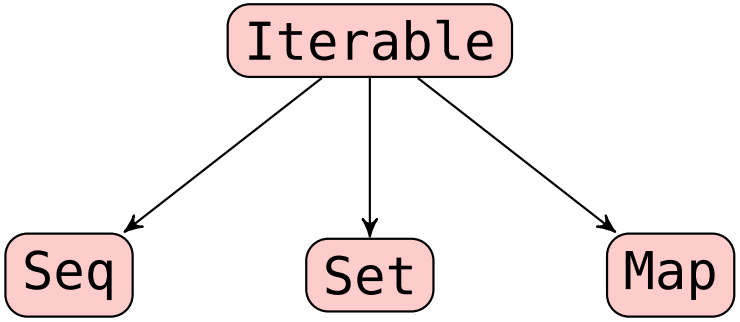
```
def copy(xs: Vector[Int]): Vector[Int] = for x <- xs yield x

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` eller andra färdiga samlingsmetoder?

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))
```

Hierarki av samlingstyper i `scala.collection` v2.13



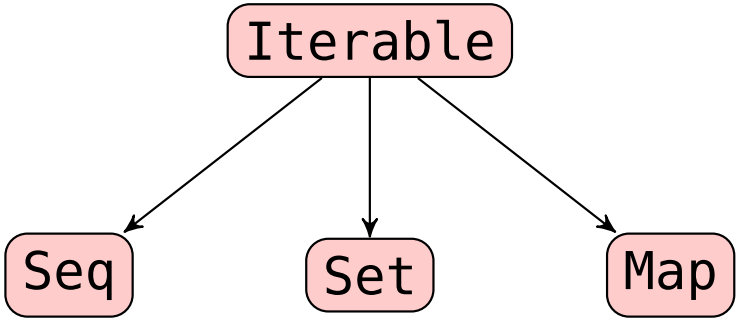
`Iterable` har metoder som är implementerade med hjälp av:

```
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`.

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]
```

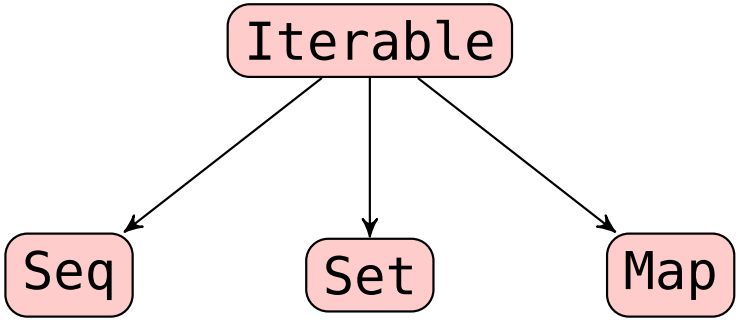
- `Seq`: ordnade i sekvens
- `Set`: unika element
- `Map`: par av (nyckel, värde)

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

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`)

Lämna det öppet: använd Seq

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

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
```

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)
```

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(x => x -> xs.count(_ == x))
```

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(x => x -> 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`.

Fördjupning: 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`

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, h
```

- Kompilatorn skapar då maskinkod som automatiskt gör typkonverteringarna.

Repeterade parametrar

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`

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
```

Enumerationer

Enumerationer har en ordning

En uppräkning av färger i en kortlek med **enum**:

```
enum Suit:
  case Spade, Heart, Club, Diamond
```

Viktiga enum-metoder för att hantera elementens **ordning**:
`ordinal fromOrdinal values valueOf`

```
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
```

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
```

Enum kan motsvara 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
```

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.)

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**.

Registrering

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	$\rightarrow$	178
2	$\rightarrow$	187
3	$\rightarrow$	167
4	$\rightarrow$	148
5	$\rightarrow$	155
6	$\rightarrow$	165

Registrering av tärningskast i `Array`

Vi låter plats 0 representera antalet ettor, plats 1 representerar antalet tvåor etc. **Övning**: implementera ???

```
scala> def rollDice(): Int = 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 = ??? //kasta tärning, räkna ut rätt index
         ??? //registrera kast i reg på rätt plats

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
```

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
```

Skapa lösningar på sekvensproblem från grunden

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

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**).

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

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.

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*.

result $\leftarrow$ en ny array med plats för *xs.length* element
$i \leftarrow 0$
while $i < xs.length$ **do**
  $result(i) \leftarrow xs(i)$
  $i \leftarrow i + 1$
end
result

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)
```

`xs.sameElements(ys)` behövs då == på en `Array` ger referenslikhet.

Implementera insert, remove, append

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).

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] = ???
```

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

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

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

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)
```

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.

Exempel: Polygon

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)
```

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)
```

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:

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.

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!

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(",",",")")
```

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(",",",")")
```

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(", "," ,")")
```

Jämföra strängar

Att sortera och jämföra strängar lexikografiskt

Teckenstandard UTF-8: Alla stora bokstäver är ”mindre” än alla små:

```
scala> Array("hej","Hej","gurka").sorted
```

Att sortera och 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)
```

Att sortera och 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
- Följdfrågor:
  - Vad betyder det att två strängar är ”lika”?
  - Vad betyder det att en sträng är ”mindre” än en annan?
- För att sortera en strängsekvens behöver vi lösa dessa delproblemen:
  - **att jämföra strängar**
  - **sökning i sekvenser**
  - **SWAP** (om på-plats-sortering i förändringsbar sekvens)

Att sortera och 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
- Följdfrågor:
  - Vad betyder det att två strängar är ”lika”?
  - Vad betyder det att en sträng är ”mindre” än en annan?
- För att sortera en strängsekvens behöver vi lösa dessa delproblemen:
  - **att jämföra strängar**
  - **sökning i sekvenser**
  - **SWAP** (om på-plats-sortering i förändringsbar sekvens)

Vi använder här strängjämförelse, sökning och sortering för att illustrera typiska **imperativa algoritmer**. **Normalt** använder man **färdiga lösningar** på dessa problem!

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*.**

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 = ???`

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
```

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).

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?

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
```

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 = ???
   ```

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) */
```

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
```

Sökning och sortering

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.

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

- **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

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 ...

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`

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.)

Sortering

Problem: Vi har en osorterad sekvens med heltal. Vi vill ordna denna osorterade sekvens i en sorterad sekvens från minst till störst.

Sortering

Problem: Vi har en osorterad sekvens med heltal. Vi vill ordna denna osorterade sekvens i en sorterad sekvens från minst till störst.

En *generalisering* av problemet:

Vi har många element av godtycklig typ och en **ordningsrelation** som säger vad vi menar med att ett element är *mindre än* eller *större än eller lika med* ett annat element.

Vi vill lösa problemet att ordna elementen i sekvens så att för varje element på plats i så är efterföljande element på plats $i + 1$ större eller lika med elementet på plats i.

- Insättningssortering **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.

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.

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?

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

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
```

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
```

Sortera 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
```

Sortera 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)
```

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
```

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!

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
```

Uppgifter denna vecka

Denna veckas övning: `sequences`

- 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.

Denna veckas laboration: `shuffle`

- 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.

Kontrollskrivning

Kontrollskrivning

Ta med **legitimation** och **snabbreferens**, blyertspenna, suddgummi, **röd** penna till rättningen, förtäring. Ingen mobil. Jackor och väskor vid väggen.

- **Diagnostisk**: vad har du lärt dig hittills?
- **Kamraträttad**: träna på att läsa och bedöma kod
- **Obligatorisk**: sjukdom **måste** meddelas i förväg
- Kan ge **samarbetsbonus om man har visat samarbetskontrakt**

  Max 5p i samarbetsbonus på **första ordinarie tentamen**, som adderas i slutet av kursen till din tentamenspoäng (max 100) och baseras på **medelvärdet** av resultaten på kontrollskrivningen i din samarbetsgrupp (se kap ”Anvisningar” i kompendiet).
- Detta är ingen ”riktig” tenta och du ska inte anmäla dig i LTH:s tentaanmälningssystem.

OBS! Du ska gå på kontrollskrivningen **även** om du inte har alla labbar godkända. Om du har mer än en icke godkänd labb efter dig så kontakta `mailto:bjorn.regnell@cs.lth.se`

Kontrollskriving – upplägg

Pågår i 5 timmar, se Timeedit för tid och plats.

- **Moment 0**: Genomgång, legitimationskontroll, utdelning
- **Moment 1**: ca 2 h 15 min: Du löser uppgifterna individuellt
- **Moment 2**: ca 1 h 15 min: Parvis kamratbedömning
- **Moment 3**: ca 30 min: Inspektera din egen skrivning
- **Moment 4**: Insamling, avslutning

- KOM I GOD TID: sitt redo vid anvisad plats en kvart **före** hel timme
- Läs **noga** igenom instruktionerna på gammal kontrollskrivning här: `https://cs.lth.se/pgk/examination/`

Plugga själv OCH i din samarbetsgrupp

- Träffas och prata i din samarbetsgrupp om hur ni bäst pluggar **individuellt** och **tillsammans** inför kontrollskrivningen för att maximera lärandet i gruppen!
- Repetera teorin i läsperiod 1.
- Repetera lösningar till övningarna och labbarna.
- Träna på att koda med papper och penna!
- Använd tidigare års kontrollskrivningar: `http://cs.lth.se/pgk/examination/`

Lycka till på kontrollskrivningen!

8 Nästlade och generiska strukturer

- Kontrollskrivning
- Veckans labb: `life`
- Matriser
- Typparametrar
- Upptäcka och åtgärda buggar

Kontrollskrivning

Resultat på kontrollskrivning

Poäng korr.	2024 antal	%	2023 antal	%
5	12	7%	10	6%
4	23	14%	18	10%
3	29	18%	29	17%
2	37	22%	55	31%
1	37	22%	52	30%
0	27	16%	11	6%
totalt	165		182	

Poäng korr.	2022 antal	%	2021 antal	%	2020 antal	%	2019 antal	%	2018 antal	%	2017 antal	%	2016 antal	%
5	4	2%	17	10%	6	5 %	6	5 %	15	12 %	22	19%	15	13%
4	14	8%	26	16%	32	28%	14	12 %	37	30 %	18	16%	18	16%
3	14	8%	25	15%	21	18%	21	18 %	20	16 %	21	18%	21	19%
2	33	20%	32	20%	34	29%	45	38 %	21	17 %	19	17%	15	13%
1	82	49%	55	34%	19	16%	28	24 %	27	22 %	31	27%	29	26%
0	22	13%	8	5%	4	3 %	4	3 %	5	4 %	4	3%	15	13%
totalt	169		163		116		118		125		115		113	

Omkontrollskrivning

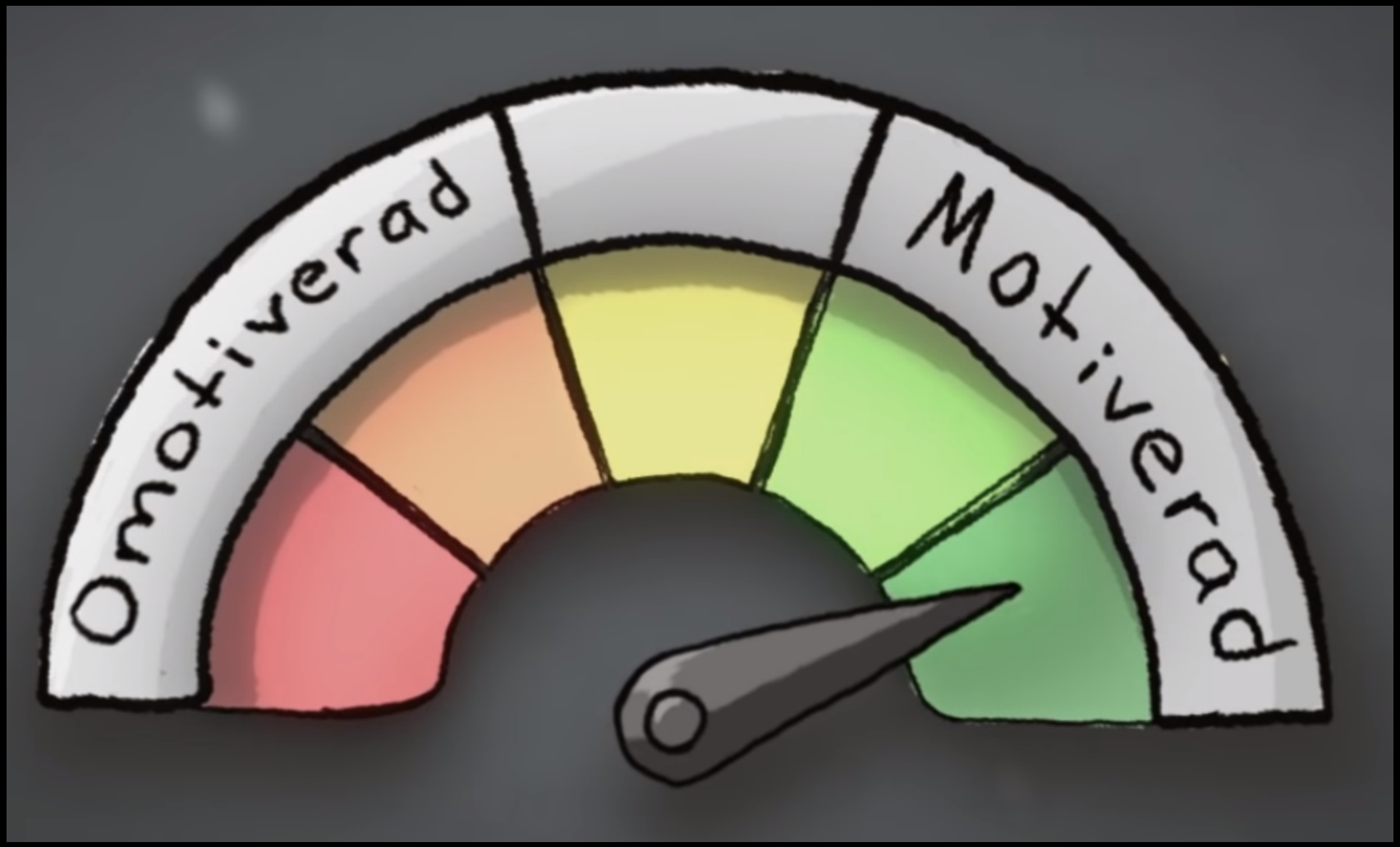
- **OMKONTROLLSKRIVNING** för de som **inte** deltog på ordinarie skrivning är **obligatorisk**:
  - **Torsd. 14/11 kl 13:00-18:00, E:1147**
  - **Frigör och boka denna tid om du ej deltog!**
  - Mer information kommer senare, håll koll på mejl.
- Frånvarande som ej fått bekräftat uppskov ska **omgående** kontakta `bjorn.regnell@cs.lth.se` och förklara varför du inte kom till kontrollskrivningen.

Repetera och fyll i kunskapsluckor

- Träffas i din samarbetsgrupp och organisera repetition och träning baserat på kontrollskrivningsresultatet.
- Använd bedömningsmallen som underlag.
  - Hur förberedde du dig?
  - Hur använde du tiden?
  - Hur påverkar det du nu vet om bedömningen hur kunde använt tiden bättre?
  - Vilka luckor har du och vad behöver du träna mer på?
- Alla rekommenderas ta med sin kontrollskrivning till resurstid och diskutera med handledare vad som är viktigast att repetera och hur pluggtekniken kan förbättras.
- Alla som hade låga poäng på kontrollskrivningen rekommenderas att gå på **extra resurstider**.

Självdiagnostik och planering

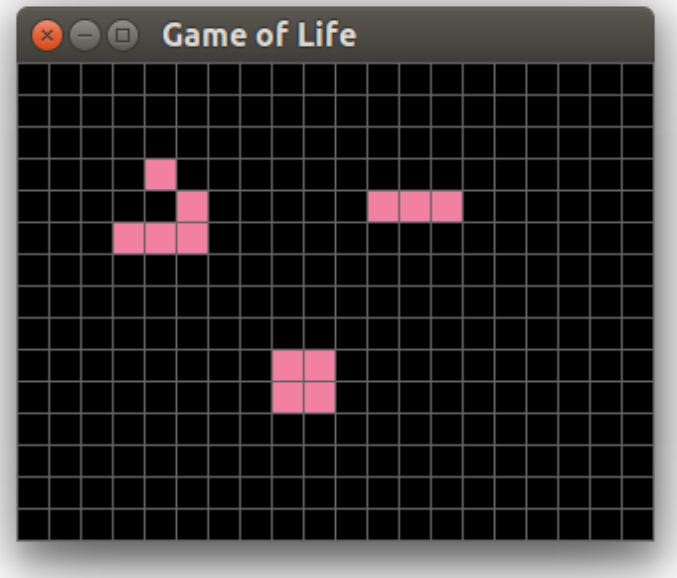
- Hur ska jag plugga?
  - Identifiera dina **behov** (trösklar, luckor, intressen...)
  - Gör ett **schema** dag för dag.
  - Dela upp varje pass i delar: teori, övningar, labbar.
  - Övningarnas syfte: att utöka din verktygslåda.
  - Identifiera övningar från lp1 om det du behöver träna på.
- Delta i all undervisning; dra nytta av höga lärartätheten!
- Alla får gå på extra resurtider i mån av plats!
  OBS! **Utöver** din **ordinarie** tid – hoppa inte över den!
  Prioritet om fullt i salen:
  1. De som deltar på sin ordinarie tid.
  2. De som varit sjuka och har labbar efter sig.
  3. De som fick 0–2 på kontrollskrivningen.
  4. De som behöver extra hjälp med svårigheter.
  5. Alla andra.


Omotiverad
Motiverad

Veckans labb: `life`

Veckans labb: `life`

- Universum är en binär matris av **celler** där **levande** celler representeras med `true` och **döda** med `false`.
- Följande regler gäller för **nästa generation** celler i universum:
  - **Fortlevnad**: en levande cell med 2 eller 3 grannar **lever vidare**
  - **Död**: en levande cell med färre än 2 eller fler än 3 grannar **dör**
  - **Födelse**: en död cell med exakt tre grannar föds
- Övning `matrices` uppgift 5: skapa en generisk `case class Matrix[T]`
- På labben: använd `Matrix[Boolean]`



- Du ska simulera *Game of Life* i ett `introprog.PixelWindow`
- Fördjupning: `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

Matriser

Vad är en matris?

- En **matris** inom **matematiken** innehåller **rader** och **kolumner**[18] med tal.
- I en **matematisk** matris har alla rader **lika många** element och
- även alla kolumner har **lika många** element.
- En matris av dimension 2×5 har $2 \cdot 5 = 10$ stycken element.
- Exempel på en matematisk matris av dimension 2×5:

$$M_{2,5} = \begin{pmatrix} 5 & 2 & 42 & 4 & 5 \\ 3 & 4 & 18 & 6 & 7 \end{pmatrix}$$

[18] även kallade *kolonner*

Indexering i en matris

- En matris av dimension $m \times n$ har $m \cdot n$ stycken element.
- En matris $A_{m,n}$ av dimension $m \times n$ ritas inom matematiken ofta så här:

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

- Matrisindexering inom matematiken sker ofta från 1, men ofta från 0 i datorprogram.
- Vad har talet 42 för index i matrisen $M_{2,5}$ nedan?
  - Inom matematiken?
  - I Scala och Java och många andra språk?

$$M_{2,5} = \begin{pmatrix} 5 & 2 & 42 & 4 & 5 \\ 3 & 4 & 18 & 6 & 7 \end{pmatrix}$$

Hur skapa matriser?

- Inom programmering används ordet **matris** ofta för att beteckna en **nästlad struktur** i två dimensioner. Exempel:
  - **Oföränderliga** sekvenser, t.ex. `Vector[Vector[Int]]`
    ```
    val xss = Vector(Vector(0, 0, 0), Vector(0, 0, 0))
    ```
    eller enklare:
    ```
    val xss = Vector.fill(2,3)(0)
    ```
  - **Föränderliga** sekvens, t.ex. `Array[Array[Int]]`
    ```
    val yss = Array(Array(0, 0, 0), Array(0, 0, 0))
    ```
    eller enklare:
    ```
    val yss = Array.fill(2,3)(0)
    ```

Hur indexera i matriser?

En matris med array av arrayer:

```
scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
xss: Array[Array[Int]] = Array(Array(5, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
```

Hur indexera i matriser?

En matris med array av arrayer:

```
scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
xss: Array[Array[Int]] = Array(Array(5, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
```

Man indexerar i en nästlad sekvens med upprepad `apply`:

```
scala> xss(0)(2)
res0: ???

scala> xss.apply(0).apply(2)
res1: ???

scala> xss(0)
res2: ???
```

Övning: Vad är typ och värde vid `???` ovan?

Hur indexera i matriser?

En matris med array av arrayer:

```
scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
xss: Array[Array[Int]] = Array(Array(5, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
```

Man indexerar i en nästlad sekvens med upprepad `apply`:

```
scala> xss(0)(2)
res0: Int = 42

scala> xss.apply(0).apply(2)
res1: Int = 42

scala> xss(0)
res2: Array[Int] = Array(5, 2, 42, 4, 5)
```

Övning: Rita en bild av minnet som referensen `xss` refererar till.

Uppdatering av en förändringsbar nästlad struktur

Man kan förändra en array av arrayer ”på plats” med tilldelning:

```
scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))

scala> xss(0)(0) = 100

scala> xss
res0: ???

scala> xss(0)(2) = xss(0)(2) - 1

scala> xss
res1: ???

scala> xss(1) = Array.fill(5)(-1)

scala> xss
res2: ???
```

Uppdatering av en förändringsbar nästlad struktur

Man kan förändra en array av arrayer ”på plats” med tilldelning:

```
scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))

scala> xss(0)(0) = 100

scala> xss
res0: Array[Array[Int]]=Array(Array(100, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))

scala> xss(0)(2) = xss(0)(2) - 1

scala> xss
res1: Array[Array[Int]]=Array(Array(100, 2, 41, 4, 5), Array(3, 4, 18, 6, 7))

scala> xss(1) = Array.fill(5)(-1)

scala> xss
res2: Array[Array[Int]]=Array(Array(100, 2, 41, 4, 5), Array(-1,-1,-1,-1,-1))
```

Några olika sätt att skapa förändringsbara matriser

Det jobbiga, primitiva sättet:

```
scala> val xss = new Array[Array[Int]](2)
xss: Array[Array[Int]] = Array(null, null)

scala> for (i <- xss.indices) {xss(i) = new Array[Int](5)}

scala> xss
res0: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))

scala> println(xss)
[[I@196a99d0
```

Enklare sätt:

```
scala> val xss = Array.ofDim[Int](2,5)
xss: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))
```

Enklare och tydligare sätt, där initialvärdet anges explicit:

```
scala> val xss = Array.fill(2,5)(0)
xss: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))
```

Exempel på skapande av oföränderlig nästlad struktur

Om du kan beräkna initialvärde direkt, använd `Vector.fill`:

```
def fill[A](n1: Int, n2: Int)(elem: => A): Vector[Vector[A]]
```

```
scala> Vector.fill(2,5)(scala.util.Random.nextInt(6) + 1)
res0:
  typ???
  värde???
```

Om du kan beräkna initialvärde ur index, använd `Vector.tabulate`:

```
def tabulate[A](n1: Int, n2: Int)(f: (Int, Int) => A): Vector[Vector[A]]
```

```
scala> Vector.tabulate(5,2)((x,y) => x + y + 1)
res1:
  typ???
  värde???
```

Exempel på skapande av oföränderlig nästlad struktur

Om du kan beräkna initialvärde direkt, använd `Vector.fill`:

```
def fill[A](n1: Int, n2: Int)(elem: => A): Vector[Vector[A]]
```

```
scala> Vector.fill(2,5)(scala.util.Random.nextInt(6) + 1)
res0: Vector[Vector[Int]] =
  Vector(Vector(1, 2, 6, 2, 1), Vector(1, 4, 3, 3, 2))
```

Om du kan beräkna initialvärde ur index, använd `Vector.tabulate`:

```
def tabulate[A](n1: Int, n2: Int)(f: (Int, Int) => A): Vector[Vector[A]]
```

```
scala> Vector.tabulate(5,2)((x,y) => x + y + 1)
res1: Vector[Vector[Int]] =
  Vector(Vector(1,2), Vector(2,3), Vector(3,4), Vector(4,5), Vector(5,  6))
```

Uppdatering av en oföränderlig nästlad struktur

Uppdatering av endimensionell struktur med `xs.updated`:

```
def updated[A](index: Int, elem: A): Vector[A]
```

```
scala> var xs = Vector.tabulate(5)(x => x + 1)
xs: typ??? = värde???

scala> xs = xs.updated(1, 42)
xs: typ??? = värde???
```

Uppdatering av nästlad struktur i två dimensioner:

```
scala> var xss = Vector.tabulate(2, 5)((x,y) => x + y + 1)
xss:
  typ??? =
  värde???

scala> xss = xss.updated(0, xss(0).updated(1, 42))
xss:
  typ??? =
  värde???
```

Uppdatering av en oföränderlig nästlad struktur

Uppdatering av endimensionell struktur med `xs.updated`:

```
def updated[A](index: Int, elem: A): Vector[A]
```

```
scala> var xs = Vector.tabulate(5)(x => x + 1)
xs: Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> xs = xs.updated(1, 42)
xs: Vector[Int] = Vector(1, 42, 3, 4, 5)
```

Uppdatering av nästlad struktur i två dimensioner:

```
scala> var xss = Vector.tabulate(2, 5)((x,y) => x + y + 1)
xss: Vector[Vector[Int]] =
  Vector(Vector(1, 2, 3, 4, 5), Vector(2, 3, 4, 5, 6))

scala> xss = xss.updated(0, xss(0).updated(1, 42))
xss:
  Vector[Vector[Int]] =
  Vector(Vector(1, 42, 3, 4, 5), Vector(2, 3, 4, 5, 6))
```

Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.
Exempel: iterera med nästlad **for**-sats för utskrift av denna matris

```
val xss = Vector.tabulate(2,5)((x,y) => x + y + 1)
```

Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.
Exempel: iterera med nästlad **for**-sats för utskrift av denna matris

```
val xss = Vector.tabulate(2,5)((x,y) => x + y + 1)
```

```
scala> for ??? do
         for ??? do
           print(xss(i)(j))
           print(" ")
         println

1 2 3 4 5
2 3 4 5 6
```

Övning:
Vad ska det stå vid ??? för att alla element ska skrivas ut?

Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.
Exempel: iterera med nästlad **for**-sats för utskrift av denna matris

```
val xss = Vector.tabulate(2,5)((x,y) => x + y + 1)
```

```
scala> for xs <- xss do
         for x <- xs do
           print(x)
           print(" ")
         end for
         println()
       end for

1 2 3 4 5
2 3 4 5 6
```

Övning: skriv ut matrisen med nästlad `foreach`

Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.
Exempel: iterera med nästlad **for**-sats för utskrift av denna matris

```
val xss = Vector.tabulate(2,5)((x,y) => x + y + 1)
```

```
scala> for xs <- xss do
         for x <- xs do
           print(x)
           print(" ")
         end for
         println()
       end for

1 2 3 4 5
2 3 4 5 6
```

Övning: skriv ut matrisen med nästlad `foreach`

```
xss.foreach { xs =>
  xs.foreach { x => print(x); print(" ") }
  println()
}
```

Övningsexempel: Yatzy

Skapa en funktion `roll` som ger utfallet av n st tärningskast:

```
scala> import scala.util.Random

scala> def roll(n: Int): Vector[Int] = ???
```

Skapa en funktion `isYatzy` som ger **`true`** om alla utfall är lika:

```
scala> def isYatzy(xs: Vector[Int]): Boolean = ???
```

Du kan anta att xs.length > 0
Tips: använd metoden xs.forall:
`def forall[A](p: A => Boolean): Boolean`

Övningsexempel: Yatzy

Skapa en funktion `roll` som ger utfallet av n st tärningskast:

```
scala> import scala.util.Random

scala> def roll(n: Int): Vector[Int] = Vector.fill(n)(Random.nextInt(6) + 1)
```

Skapa en funktion `isYatzy` som ger **`true`** om alla utfall är lika:

```
scala> def isYatzy(xs: Vector[Int]): Boolean = xs.forall(x => x == xs(0))
```

Du kan anta att xs.length > 0
Tips: använd metoden xs.forall:

```
def forall[A](p: A => Boolean): Boolean
```

Iterera över nästlad struktur: for-sats

Iterera med nästlad for-sats: (vad har xss för typ?)

```
scala> val xss = Vector.fill(100)(roll(5))

scala> for i <- ??? do
         for j <- ??? do
           print(s"($i)($j): ${xss(i)(j)} ")
         println(s" YATZY: ${isYatzy(xss(i))}")

(0)(0): 3 (0)(1): 6 (0)(2): 4 (0)(3): 4 (0)(4): 6  YATZY: false
(1)(0): 4 (1)(1): 1 (1)(2): 5 (1)(3): 2 (1)(4): 6  YATZY: false
(2)(0): 1 (2)(1): 3 (2)(2): 5 (2)(3): 6 (2)(4): 2  YATZY: false
(3)(0): 2 (3)(1): 1 (3)(2): 1 (3)(3): 5 (3)(4): 4  YATZY: false
(4)(0): 4 (4)(1): 4 (4)(2): 1 (4)(3): 6 (4)(4): 5  YATZY: false
(5)(0): 3 (5)(1): 3 (5)(2): 2 (5)(3): 3 (5)(4): 6  YATZY: false
(6)(0): 3 (6)(1): 6 (6)(2): 1 (6)(3): 1 (6)(4): 4  YATZY: false
(7)(0): 6 (7)(1): 2 (7)(2): 4 (7)(3): 4 (7)(4): 3  YATZY: false
(8)(0): 1 (8)(1): 5 (8)(2): 4 (8)(3): 2 (8)(4): 4  YATZY: false
(9)(0): 1 (9)(1): 1 (9)(2): 3 (9)(3): 6 (9)(4): 6  YATZY: false
(10)(0): 2 (10)(1): 5 (10)(2): 2 (10)(3): 4 (10)(4): 5  YATZY: false
(11)(0): 3 (11)(1): 4 (11)(2): 2 (11)(3): 5 (11)(4): 6  YATZY: false
...
```

Iterera över nästlad struktur: for-sats

Iterera med nästlad for-sats: (xss är en `Vector[Vector[Int]]`)

```
scala> val xss = Vector.fill(100)(roll(5))

scala> for i <- xss.indices do
         for j <- xss(i).indices do
           print(s"($i)($j): ${xss(i)(j)} ")
         println(s" YATZY: ${isYatzy(xss(i))}")

(0)(0): 3 (0)(1): 6 (0)(2): 4 (0)(3): 4 (0)(4): 6  YATZY: false
(1)(0): 4 (1)(1): 1 (1)(2): 5 (1)(3): 2 (1)(4): 6  YATZY: false
(2)(0): 1 (2)(1): 3 (2)(2): 5 (2)(3): 6 (2)(4): 2  YATZY: false
(3)(0): 2 (3)(1): 1 (3)(2): 1 (3)(3): 5 (3)(4): 4  YATZY: false
(4)(0): 4 (4)(1): 4 (4)(2): 1 (4)(3): 6 (4)(4): 5  YATZY: false
(5)(0): 3 (5)(1): 3 (5)(2): 2 (5)(3): 3 (5)(4): 6  YATZY: false
(6)(0): 3 (6)(1): 6 (6)(2): 1 (6)(3): 1 (6)(4): 4  YATZY: false
(7)(0): 6 (7)(1): 2 (7)(2): 4 (7)(3): 4 (7)(4): 3  YATZY: false
(8)(0): 1 (8)(1): 5 (8)(2): 4 (8)(3): 2 (8)(4): 4  YATZY: false
(9)(0): 1 (9)(1): 1 (9)(2): 3 (9)(3): 6 (9)(4): 6  YATZY: false
(10)(0): 2 (10)(1): 5 (10)(2): 2 (10)(3): 4 (10)(4): 5  YATZY: false
(11)(0): 3 (11)(1): 4 (11)(2): 2 (11)(3): 5 (11)(4): 6  YATZY: false
...
```

Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```
scala> val xss = for i <- 1 to 2 yield
                   for j <- 1 to 5 yield i + j + 1

val xss: IndexedSeq[IndexedSeq[Int]] =
      ???
```

Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```
scala> val xss = for i <- 1 to 2 yield
                   for j <- 1 to 5 yield i + j + 1

val xss: IndexedSeq[IndexedSeq[Int]] =
      ???
```

Om man skriver så här får man en endimensionell struktur:

```
scala> val xs = for { i <- 1 to 2; j <- 1 to 5 } yield i + j + 1
val xs: IndexedSeq[Int] =
    ???
```

Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```
scala> val xss = for i <- 1 to 2 yield
                  for j <- 1 to 5 yield i + j + 1

val xss: IndexedSeq[IndexedSeq[Int]] =
    Vector(Vector(3, 4, 5, 6, 7), Vector(4, 5, 6, 7, 8))
```

Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```
scala> val xss = for i <- 1 to 2 yield
                   for j <- 1 to 5 yield i + j + 1

val xss: IndexedSeq[IndexedSeq[Int]] =
    Vector(Vector(3, 4, 5, 6, 7), Vector(4, 5, 6, 7, 8))
```

Om man skriver så här får man en endimensionell struktur:

```
scala> val xs = for { i <- 1 to 2; j <- 1 to 5 } yield i + j + 1
val xs: IndexedSeq[Int] =
    Vector(3, 4, 5, 6, 7, 4, 5, 6, 7, 8)
```

Nästlade map-uttryck

Iterera med **nästlade map-uttryck**:

```
scala> val xss = (1 to 2).map(i => (1 to 5).map(j => i + j + 1))
xss: IndexedSeq[IndexedSeq[Int]] =
      ???
```

Nästlade map-uttryck

Iterera med **nästlade map-uttryck**:

```
scala> val xss = (1 to 2).map(i => (1 to 5).map(j => i + j + 1))
xss: IndexedSeq[IndexedSeq[Int]] =
      Vector(Vector(3, 4, 5, 6, 7), Vector(4, 5, 6, 7, 8))
```

Fallgrop: likhet av array

```
scala> Vector.fill(5, 2)(42) == Vector.fill(5, 2)(42)
val res0: ???

scala> Array.fill(5, 2)(42) == Array.fill(5, 2)(42)
val res1: ???
```

Fallgrop: likhet av array

```
scala> Vector.fill(5, 2)(42) == Vector.fill(5, 2)(42)
val res0: Boolean = true

scala> Array.fill(5, 2)(42) == Array.fill(5, 2)(42)
val res1: Boolean = false  // AAAARRGH!!! :(
```

Primitiva arrayer har en equals-metod som ger referenslikhet, **inte** innehållslikhet. Och det fungerar följaktligen ej heller på nästlade strukturer.

Kolla likhet mellan två heltalsmatriser (uppfinner hjulet)

```
def isEqual(xss: Array[Array[Int]], yss: Array[Array[Int]]) =
  if xss.length != yss.length then false else
    var i = 0
    var foundUnequal = false
    while ??? do
      if xss(i).length != yss(i).length then
        ???
      else
        var j = 0
        while ??? do
          if xss(i)(j) != yss(i)(j) then ???
          j += 1
        end while
      end if
      i += 1
    end while
    !foundUnequal
  end if
end isEqual
```

Kolla likhet mellan två heltalsmatriser (uppfinner hjulet)

```
def isEqual(xss: Array[Array[Int]], yss: Array[Array[Int]]) =
  if xss.length != yss.length then false else
    var i = 0
    var foundUnequal = false
    while i < xss.length && !foundUnequal do
      if xss(i).length != yss(i).length then
        foundUnequal = true
      else
        var j = 0
        while j < xss(i).length && !foundUnequal do
          if xss(i)(j) != yss(i)(j) then foundUnequal = true
          j += 1
        end while
      end if
      i += 1
    end while
    !foundUnequal
  end if
end isEqual
```

Använd INTE `sameElements` på nästlade arrayer

I Scala kan du använda metoden `sameElements` för att kolla innehållslikhet mellan två arrayer, men det funkar **INTE** på djupet i nästlade strukturer.

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

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

scala> xs.sameElements(ys)                          // xs, ys ej nästlade
res0: Boolean = true                                // innehåll lika!

scala> Array(Array(1)) sameElements Array(Array(1))
res1: Boolean = false                               //AAAARGH!
```

Använd i stället:
`java.util.Objects.deepEquals` eller `java.util.Arrays.deepEquals`
Den senare kräver typkonvertering av argumenten med:
`asInstanceOf[Array[Object]]`

Kontroll av innehållslikhet mellan nästlade arrayer

`java.util.Objects.deepEquals` fungerar **på djupet** för godtyckliga referenstyper:

```
scala> java.util.Objects.deepEquals(
          Array(Array("a", Array("b"), 42)),
          Array(Array("a", Array("b"), 42)))
val res0: Boolean = true

scala> java.util.Objects.deepEquals(
          Array(Array("a", Array("b"), 42)),
          Array(Array("a", Array("b"), 43)))
val res1: Boolean = false
```

`java.util.Objects.deepEquals` kontrollerar om argumenten är arrayer och anropar då i sin tur `java.util.Arrays.deepEquals` efter typkonvertering:

```
scala> java.util.Arrays.deepEquals(
          Array(Array("a", Array("b"), 42)).asInstanceOf[Array[Object]],
          Array(Array("a", Array("b"), 42)).asInstanceOf[Array[Object]])
val res3: Boolean = true
```

`https://stackoverflow.com/questions/63686721/best-replacement-of-deep-method-in-scala-2-13`

Om veckans övningar

- Träna på att iterera över nästlade strukurer
- Fortsätt jobba med Yatzy-exemplet
- träna på att skapa **imperativa** algoritmer:
  lös `isYatzy` med **`while`**-sats
- Extrauppgift där du ska bygga ett enkelt yatzy-spel i terminalen (kunde varit en tentauppgift...)

Typparametrar

Exempel: Icke-generisk case-klass med heltalsmatris

En *icke-generisk* datastruktur har inga obundna typparametrar; alla typer är **konkreta** (alltså specifika).

En icke-generisk case-class med en `Vector[Vector[Int]]`:

```
case class Matrix(data: Vector[Vector[Int]]):
  def apply(x: Int, y: Int): Int = data(x)(y)
```

```
scala> Matrix(Vector(Vector(5, 2, 42, 4, 5),Vector(3, 4, 18, 6, 7)))
res0: Matrix =
  Matrix(Vector(Vector(5, 2, 42, 4, 5), Vector(3, 4, 18, 6, 7)))

```

Exempel: Generisk case-klass med generell matris

En *generisk* datastruktur har minst en **obunden typparameter** som vid användning ska bindas till ett **konkret typargument**.

```
case class Matrix[T](data: Vector[Vector[T]]):
  def apply(x: Int, y: Int): T = data(x)(y)
```

`Matrix` i exemplet ovan är en **generisk** case-class där T är obunden, eftersom T är en typparameter deklarerad inom `[]` **efter** klassens namn men **före** klassparameterlistan.

Användning där T binds till `Int` via kompilatorns typhärledning:

```
scala> Matrix(Vector(Vector(5, 2, 42, 4, 5),Vector(3, 4, 18, 6, 7)))
res1: Matrix[Int] =
  Matrix(Vector(Vector(5, 2, 42, 4, 5), Vector(3, 4, 18, 6, 7)))

```

Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *paramteric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *paramteric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

- En **fri** typparameter kan bindas till vilken typ som helst.
- Bindingen av typargument till typparametrar sker vid **kompileringstid**.
- En typparameter är **fri** om den **inte** fått något värde, annars **bunden**.

Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *paramteric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

- En **fri** typparameter kan bindas till vilken typ som helst.
- Bindingen av typargument till typparametrar sker vid **kompileringstid**.
- En typparameter är **fri** om den **inte** fått något värde, annars **bunden**.
- Exempel: **generisk klass** med **generiska metoder**:

```
class Cell[A](   // A är här fri men måste bindas vid användning
    var value: A):                // A är bunden vid användning av Cell
  def update(a: A): Unit = value = a    // A är även här bunden vid anv.
  def replaced[B](b: B): Cell[B] = new Cell(b)  // första [B] är fri
```

Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *paramteric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

- En **fri** typparameter kan bindas till vilken typ som helst.
- Bindingen av typargument till typparametrar sker vid **kompileringstid**.
- En typparameter är **fri** om den **inte** fått något värde, annars **bunden**.
- Exempel: **generisk klass** med **generiska metoder**:

```
class Cell[A](   // A är här fri men måste bindas vid användning
    var value: A):                // A är bunden vid användning av Cell
  def update(a: A): Unit = value = a    // A är även här bunden vid anv.
  def replaced[B](b: B): Cell[B] = new Cell(b)  // första [B] är fri
```

- **Skuggning kan förekomma**: Om `replaced` i `Cell` hade använt namnet A på sin typparameter hade den **skuggat** klassens typparameter och tolkats som en fri typparameter, alltså en godtycklig typ och **inte** klassens typparameter. (jämför namnöverskuggning vid **lokala** namn i nästlade block)

Exempel: Generisk funktion

Vad händer här?

```

scala> def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
???



scala> def skrikBaklänges[T](x: T): String = x.toString.toUpperCase.reverse

scala> skrikBaklänges("gurka är gott")
val res0: ???
```

Exempel: Generisk funktion

Vad händer här?

```

scala> def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
1 |def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
  |                       ^
  |                       Not found: type T
                              ^

scala> def skrikBaklänges[T](x: T): String = x.toString.toUpperCase.reverse

scala> skrikBaklänges("gurka är gott")
val res0: ???
```

Exempel: Generisk funktion

Vad händer här?

```

scala> def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
1 |def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
  |                       ^
  |                       Not found: type T
                              ^

scala> def skrikBaklänges[T](x: T): String = x.toString.toUpperCase.reverse

scala> skrikBaklänges("gurka är gott")
val res0: String = TTOG RÄ AKRUG
```

Om ingen typparameter deklareras inom hakparenteser efter funktionens namns så vet inte kompilatorn vad T är för en typ. Men med en typparameter `[T]` efter funktionsnamnet tolkar kompilatorn funktionen som **generisk** och typen T bestäms av argumentets typ **vid anrop** och T kan bindas till godtycklig typ.

Exempel: Generisk case-klass

- En generisk klass har en eller flera typparametrar efter klassnamnet:

```
case class Box[A](value: A)
```

- Kompilatorn härleder typparameterarnas typ utifrån givna värden.

```
scala> Box("gurka")
val res1: Box[String] = Box(gurka)
```

- Du kan också ge typpparametern en typ explicit:

```
scala> Box[Int](42)
val res2: Box[Int] = Box(42)
```

- Om typen inte stämmer får du hjälp av kompilatorn att hitta felet:

```
scala> Box[String](42)
-- Error:
1 |Box[String](42)
  |            ^^ Found:    (42 : Int)  Required: String
```

- En generisk klass, här `Box`, kallas också **typkonstruktor** (eng. *type constructor*) då den ”färdiga” typen `Box[Int]` ”konstrueras” på platsen där den används.

Fallgrop: Typradering (eng. *type erasure*)

Informationen om typerna i typparametrar raderas innan kodgenerering för JVM av prestandaskäl och **typparametrar saknas vid runtime** i bytekoden.

```
scala> def isIntVector[T](xs: Vector[T]) = xs.isInstanceOf[Vector[Int]]
-- Warning:
1 |def isIntVector[T](xs: Vector[T]) = xs.isInstanceOf[Vector[Int]]
  |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |                the type test for Vector[Int] cannot be checked at runtime
def isIntVector[T](xs: Vector[T]): Boolean

scala> isIntVector(Vector("hej"))
res42: Boolean = true  // AAAARGHH!! :(
```

Måste ”packa upp” samlingen och typtesta alla element:

```
scala> def isIntVector[T](xs: Vector[T]) = xs.forall(_.isInstanceOf[Int])

scala> isIntVector(Vector("hej"))
res43: Boolean = false  // FUNKAR :)
```

Typkontroll vid körtid görs oftast hellre med `match`.

Upptäcka och åtgärda buggar

Testning och avlusning

- Läs om testning och avlusning (eng. *debugging*) i Appendix D: ”Fixa buggar”
- Träna på println-debugging
- Prova debuggern i VS code

9 Mängder och tabeller

- Veckans uppgifter
- Kodgranskning
- `scala.collection`
- Repetition: sekvens
- Mängd
- Nyckel-värde-tabell
- Tips inför veckans uppgifter
- Serialisering och deserialisering

Omkontroll

- De som ej deltog i kontrollskrivningen ska delat på **omkontrollskrivning Torsd. 14/11 kl 13:00-18:00, E:1147**
- Detta obligatoriska moment krävs för godkänt på kursen.
- Du som inte deltog vid ordinarie skrivning ska ha fått mejl från `bjorn.regnell@cs.lth.se` med instruktioner.
  Om du ej fått mejl men ska delta: mejla Björn snarast!

Veckans uppgifter

Övning: `lookup`

- Uppgifterna innehåller delar som är **nödvändiga** för laborationen.
- På övningen tränar du på **mängder** och **nyckel-värde-tabeller**.
- Du ska skapa en klass `FreqMapBuilder` som bygger upp en tabell med ordfrekvenser, som behövs på labben.

Laboration: `words`

- Denna uppgift handlar om analys av naurligt språk (eng. *Natural Language Processing, NLP*).
- Svara på frågorna:
  - Hur vanligt är ett visst ord i en given text?
  - Vilket är det vanligaste ordet som följer efter ett visst ord?
  - Hur kan man generera ordsekvenser som liknar ordföljden i en given text?
- Använda mängd för unika ord.
- Använda nyckel-värde-tabell för att för varje ord i en lång text räkna antalet förekomster av detta ord.

Laboration: `words`

Lärandemål:

- Kunna skapa och använda nyckel-värde-tabeller med samlingstypen `Map`.
- Kunna skapa och använda mängder med samlingstypen `Set`.
- Förstå skillnader och likheter mellan en sekvens och en mängd.
- Förstå likheter och skillnader mellan en sekvens av par och en nyckel-värde-tabell.
- Kunna implementera algoritmer som använder nästlade strukturer.

Uppgifter:

- Dela upp en sträng i ord.
- Skapa ordfrekvenstabeller för böcker som ditt program laddar ner från nätet via projektet Gutenberg med fria böcker.
- Skapa frekvenstabeller för ordföljder, s.k. *n-gram*.
- Skriv ut intressant statistik om ordvalen i olika böcker, t.ex. ur könsrollsperspektiv.
- Valfri uppgift: Gör en bot som genererar slumpvisa, artificiella meningar som liknar mänskligt språk.

Kodgranskning

Att läsa kod

En förutsättning för att kunna *skriva* bra kod är att kunna *läsa* kod aktivt. Du behöver kontinuerlig träning i att **läsa** kod! (inte bara skriva)

Hur läsa kod?

Att läsa kod

En förutsättning för att kunna *skriva* bra kod är att kunna *läsa* kod aktivt. Du behöver kontinuerlig träning i att **läsa** kod! (inte bara skriva)

Hur läsa kod?

- Skapa överblick av vilka kodfiler som finns och vad som finns i vilken fil.
- Studera dokumentation om vad som är **syftet** med olika abstraktioner.
- Studera klassparametrar och metodhuvuden. **Typerna** är dina **tankeverktyg**: ”Vad kommer in?” och ”Vad kommer ut?”
- Ställ dig under läsningen frågorna:
  ”Vad finns?” och ”Vad fattas?” för att du ska kunna lösa din uppgift.
- Läs iterativt. Vänta med implementationsdetaljer. Hoppa mellan deklaration och användning.
- Studera **beroenden**: `Matrix -> Life -> LifeWindow -> Main`
- Två strategier i din stegvisa läsning som kan mixas:
  Bottom-Up: Börja med delar med **minst** beroenden till andra delar.
  Top-Down: Börja med de delar som används av **huvudprogrammet**.
- Var aktiv när du läser! Anteckna; skriv ner frågor; experimentera i REPL.

Kodgranskning i industriell systemutveckling

- Ett effektivt sätt att upptäcka fel är att människor **noga läser igenom** sin egen och andras kod, och försöker hitta relevanta **problem och förbättringsmöjligheter**.
- Man blir ofta ”hemmablind” när det gäller ens egen kod. Därför kan någon annans, oberoende granskning med ”nya, friska” ögon vara mycket fruktbar.
- I samband med kodgranskning kan man med fördel försöka bedöma huruvida koden är:
  - lätt att läsa,
  - lätt att ändra i,
  - annat som är viktigt för den framtida utvecklingen.
- Ofta hittar man vid granskning även enkla programmeringsmisstag, så som felaktiga villkor och loop-räknare som inte räknas upp på rätt sätt etc.

Nyttan med kodgranskningar

Väl genomförda kodgranskningar är effektiva och nyttiga:

- Bra på att upptäcka problem, även sådana som varken kompilator eller testning hittar.
- Sprider kunskap inom en arbetsgrupp, speciellt mellan erfarna och juniora utvecklare.
- En god kodgranskningsprocess främjar en kultur av gemensamt ägarskap och respektfull samverkan.

Kodgranskning under grupplaborationen `snake`

Grupplaborationen `snake` går över två läsveckor (w10–w11). Du ska:

- delta i framtagande av en gemensam checklista för kodgranskning,
- granska minst en annan gruppmedlems kod,
- bjuda in minst en annan gruppmedlem att granska din kod,
- ge konstruktiv feedback,
- ta emot konstruktiv feedback och försöka förbättra din kod,
- på redovisning redogöra för nyttan och utmaningarna med kodgranskningar utifrån dina egna erfarenheter.

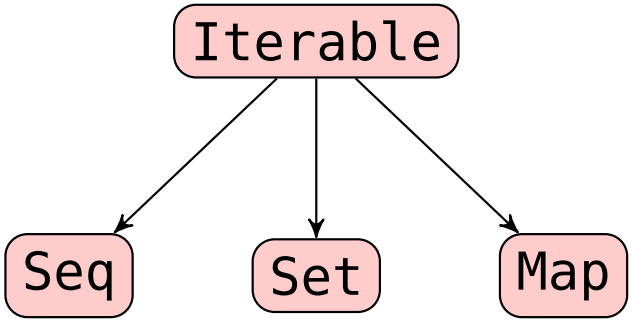
Mer om kodgranskning i efterföljande kurser, t.ex. ”Programvaruutveckling i grupp” och ”Programvaruutveckling för stora system”

Gästföreläsning om kodgranskningar i praktiken

- **Sprid info till alla**: På andra föreläsningen i nästa vecka kommer **gäst från industrin**: den erfarne utvecklaren **Gustaf Lundh** från Axis och gästföreläser om kodgranskningar i praktiken.
- Gästföreläsningen ger dig en flygande start inför de kodgranskningar du ska genomföra i grupplabben `snake`.
- **Missa inte det!**

scala.collection

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 förändringsbara `scala.Array` är automatiskt synlig)

Metoden `iterator` ger en ”engångs-iterator”

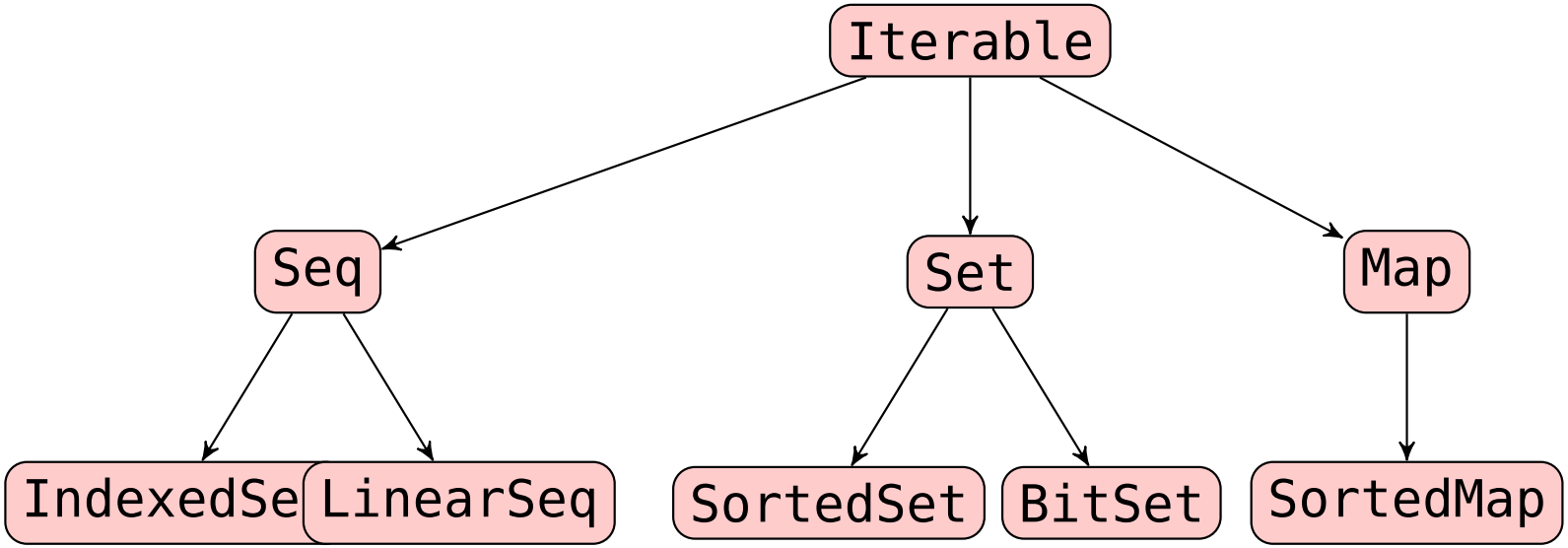
Med `iterator` kan man iterera med **`while`**, men endast **en gång**; sedan är iteratorn ”förbrukad”. (Men man kan be om en ny.) Används ”under huven” i samlingsbiblioteket för att implementera andra metoder.

```
scala> val xs = Vector(1,2,3,4)
val xs: Vector[Int] = Vector(1, 2, 3, 4)

scala> val it = xs.iterator
val it: Iterator[Int] = <iterator>

scala> while it.hasNext do print(it.next)
1234

scala> it.hasNext
val res0: Boolean = false

scala> it.next
java.util.NoSuchElementException: next on empty iterator
```

Normalt behöver man **inte** använda `iterator`: det finns oftast färdiga metoder som gör det man vill, till exempel `foreach`, `map`, `sum`, `min` etc.

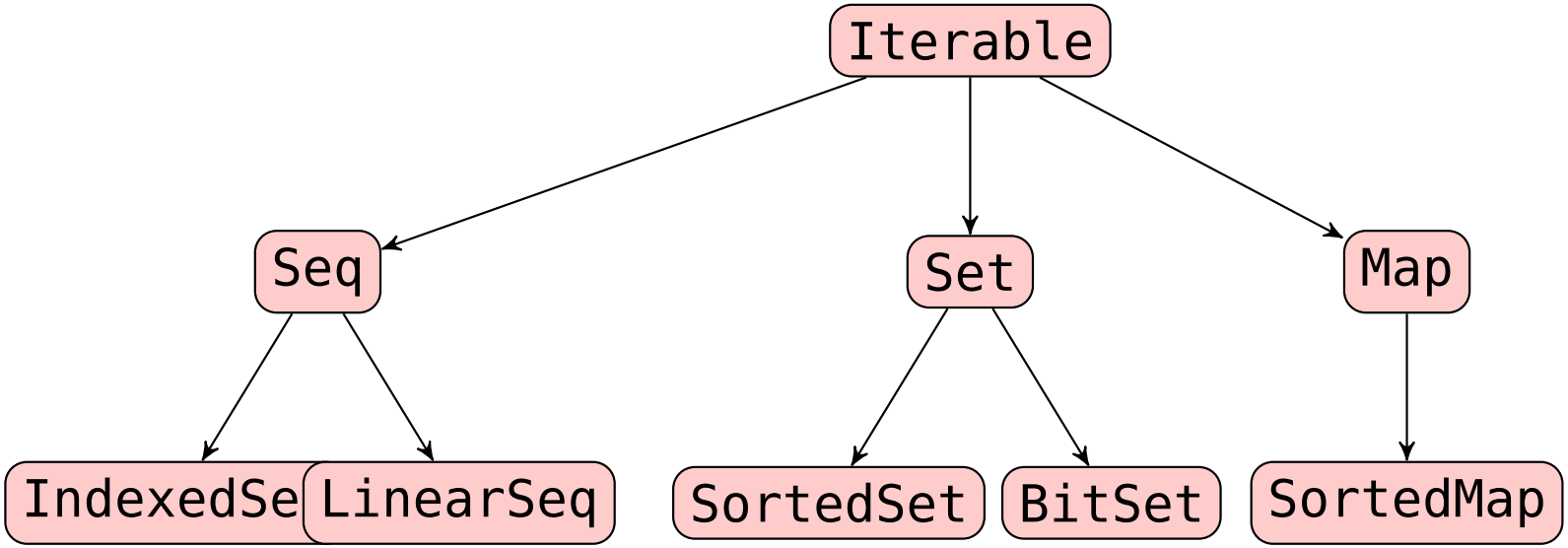
Mer specifika samlingstyper i `scala.collection`

Det finns **mer specifika subtyper** av `Seq`, `Set` och `Map`:



Mer specifika samlingstyper i `scala.collection`

Det finns **mer specifika subtyper** av `Seq`, `Set` och `Map`:



`Vector` är en **`IndexedSeq`** medan **`List`** är en **`LinearSeq`**.

docs.scala-lang.org/overviews/collections-2.13/overview.html

Några oföränderliga och förändringsbara sekvenssamlingar

```
scala.collection.immutable.Seq.
                                IndexedSeq.
                                            Vector
                                            Range
                                LinearSeq.
                                            List
                                            Queue
  scala.collection.mutable.Seq.
                                IndexedSeq.
                                            ArrayBuffer
                                            StringBuilder
                                LinearSeq.
                                            ListBuffer
                                            Queue
```

Fördjupning: Studera samlingars prestanda-egenskaper här:
docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html

Några användbara metoder på samlingar

Iterable	`xs.size`	antal elementet
	`xs.head`	första elementet
	`xs.last`	sista elementet
	`xs.take(n)`	ny samling med de första n elementet
	`xs.drop(n)`	ny samling utan de första n elementet
	`xs.foreach(f)`	gör `f` på alla element, returtyp `Unit`
	`xs.map(f)`	gör `f` på alla element, ger ny samling
	`xs.filter(p)`	ny samling med bara de element där p är sant
	`xs.groupBy(f)`	ger en `Map` som grupperar värdena enligt f
	`xs.mkString(",")`	en kommaseparerad sträng med alla element
	`xs.zip(ys)`	ny samling med par (x, y); ”zippa ihop” xs och ys
	`xs.zipWithIndex`	ger en `Map` med par (x, index för x)
	`xs.sliding(n)`	ny samling av samlingar genom glidande ”fönster”
Seq	`xs.length`	samma som `xs.size`
	`xs :+ x`	ny samling med x sist efter xs
	`x +: xs`	ny samling med x före xs

Prova fler samlingsmetoder ur snabbreferensen: `http://cs.lth.se/quickref`

Minnesregel för `+:` och `:+` **Colon on the collection side**

Digga denna: `https://youtu.be/Lm9JWlEMHjo?si=sNdn_ZDaORlGr3lt`

Repetition: sekvens

Repetition: Vad är en sekvens?

- En sekvens är en **följd av element** som
  - är **numrerade** (t.ex. från noll), och
  - är av en viss **typ** (t.ex. heltal).

Repetition: Vad är en sekvens?

- En sekvens är en **följd av element** som
  - är **numrerade** (t.ex. från noll), och
  - är av en viss **typ** (t.ex. heltal).
- En sekvens kan innehålla **dubbletter**.
- En sekvens kan vara **tom** och ha längden noll.
- Exempel på en icke-tom sekvens med dubbletter:

```
scala> val xs = Vector(42, 0, 42, -9, 0, 13, 7)
val xs: Vector[Int] = Vector(42, 0, 42, -9, 0, 13, 7)
```

Repetition: Vad är en sekvens?

- En sekvens är en **följd av element** som
  - är **numrerade** (t.ex. från noll), och
  - är av en viss **typ** (t.ex. heltal).
- En sekvens kan innehålla **dubbletter**.
- En sekvens kan vara **tom** och ha längden noll.
- Exempel på en icke-tom sekvens med dubbletter:

```
scala> val xs = Vector(42, 0, 42, -9, 0, 13, 7)
val xs: Vector[Int] = Vector(42, 0, 42, -9, 0, 13, 7)
```

- **Indexering** ger ett element via dess ordningsnummer:

```
1  scala> xs(2)
2  val res0: Int = 42
3
4  scala> xs.apply(2)
5  val res1: Int = 42
```

En sträng är också en `IndexedSeq[Char]`

Det sker vid behov **implicit konvertering** från `String` till `IndexedSeq[Char]`.

```
scala> val x: IndexedSeq[Char] = "hej"
val x: IndexedSeq[Char] = hej
```

Detta gör att **alla samlingsmetoder på Seq även funkar på strängar** och även flera andra smidiga strängmetoder erbjuds **utöver** de som finns i `java.lang.String` genom klassen `StringOps`.

```
scala> "hej".  //tryck på TAB och se alla strängmetoder
JLine: do you wish to see all 248 possibilities (42 lines)?
```

Detta är en stor fördel med Scala jämfört med många andra språk, som har strängar som inte kan allt som andra sekvenssamlingar kan.

Konvertera mellan olika samlingstyper

- För vanligt förekommande konverteringar finns metoderna `toVector`, `toList`, `toArray`, `toBuffer`, `toMap`, `toSeq`, `toIndexedSeq`, `toSet`, `toString`
- Metoden `to` (ny från Scala 2.13) tar ett **kompanjonsobjekt** ur samlingsbiblioteket som argument och kan användas för konvertering till godtycklig samlingstyp.
- Detta kräver kopiering om underliggande representation är olika och samlingen är förändringsbar.
- Kan användas för att t.ex. konvertera mellan oföränderlig och förändringsbar samling:

```
scala> val ms = Set(1,2,3).to(collection.mutable.Set)
val ms: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3)
```

Mängd

Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubbletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubbletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

- En mängd är **inte** en sekvens: du kan inte utgå från att elementen ligger i någon viss ordning, t.ex. den ordning som de ges vid konstruktion; en mängd har ej längd, men en **storlek**; metoden `size` ger antalet element men metoden `length` saknas.
- En mängd kan vara **tom** och har då storleken `0`.

Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubbletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

- En mängd är **inte** en sekvens: du kan inte utgå från att elementen ligger i någon viss ordning, t.ex. den ordning som de ges vid konstruktion; en mängd har ej längd, men en **storlek**; metoden `size` ger antalet element men metoden `length` saknas.
- En mängd kan vara **tom** och har då storleken `0`.
- Man kan gå igenom element i **någon** ordning (exakt vilken är ej def.), med till exempel `xs.map(f)` eller `for (x <- xs) yield f(x)`

Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubbletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

- En mängd är **inte** en sekvens: du kan inte utgå från att elementen ligger i någon viss ordning, t.ex. den ordning som de ges vid konstruktion; en mängd har ej längd, men en **storlek**; metoden `size` ger antalet element men metoden `length` saknas.
- En mängd kan vara **tom** och har då storleken `0`.
- Man kan gå igenom element i **någon** ordning (exakt vilken är ej def.), med till exempel `xs.map(f)` eller `for (x <- xs) yield f(x)`
- Det går **inte** att indexera i en mängd med `apply`, som i stället ger **innehållstest**: `Set(1,2,3).apply(3) == true`
- En mängd `Set[T]` med element av typen `T` kan således ses som ett **predikat för innehållstest**: alltså en funktion `T => Boolean` som är `true` om elementet finns annars `false`

Oföränderlig mängd

- **Skapa**:

```
scala> var xs = Set("gurka", "tomat", "banan", "pingvin")
```

- **Läsa**: avgöra medlemskap

```
scala> xs("gurka")
val res1: Boolean = true
```

- **Uppdatera**: lägg till element (händer inget om redan finns)

```
scala> xs = xs + "jordekorre" // en ny, delvis förändrad
```

- **Ta bort**: (om finns, annars händer inget)

```
scala> xs = xs - "gurka" // en ny, delvis förändrad
```

SLUT = Skapa, Läsa, Uppdatera, Ta bort CRUD = Create, Read, Update, Delete

Mysteriet med de försvunna elementen

Vad händer här?

```
scala> val xs1 = Vector(1,2,3,4,5,6)
scala> xs1.map(_ % 2).count(_ == 0)
val res0: Int = 3                          // antalet jämna
scala> val xs2 = Set(1,2,3,4,5,6)
scala> xs2.map(_ % 2).count(_ == 0)
val res1: Int = 1                          // varför?
```

Mysteriet med de försvunna elementen

Vad händer här?

```
scala> val xs1 = Vector(1,2,3,4,5,6)
scala> xs1.map(_ % 2).count(_ == 0)
val res0: Int = 3                          // antalet jämna
scala> val xs2 = Set(1,2,3,4,5,6)
scala> xs2.map(_ % 2).count(_ == 0)
val res1: Int = 1                          // varför?
```

Mängdegenskaper ger att `xs2.map(_ % 2) == Set(0, 1)`
Fundera alltid noga på om du **riskerar att förlora duplikat** som du egentligen hade velat behålla!

Mysteriet med de försvunna elementen

Vad händer här?

```
scala> val xs1 = Vector(1,2,3,4,5,6)
scala> xs1.map(_ % 2).count(_ == 0)
val res0: Int = 3                          // antalet jämna
scala> val xs2 = Set(1,2,3,4,5,6)
scala> xs2.map(_ % 2).count(_ == 0)
val res1: Int = 1                          // varför?
```

Mängdegenskaper ger att `xs2.map(_ % 2) == Set(0, 1)`
Fundera alltid noga på om du **riskerar att förlora duplikat** som du egentligen hade velat behålla!
Använd `toSeq` på mängd om du behöver sekvensegenskaper:

```
scala> xs2.toSeq.map(_ % 2).count(_ == 0)
val res1: Int = 3          // med toSeq blir det som vi ville
```

Förändringsbar mängd

Med en **förändringsbar** mängd kan man stegvis utöka på plats.

```
scala> val mängd = scala.collection.mutable.Set.empty[Int]

scala> for i <- 1 to 1_000_000 do mängd.addOne(i)

scala> mängd.contains(-1)   // samma som mängd(-1) eller mängd.apply(-1)
```

En **mängd** är **snabb** på att avgöra om ett element **finns eller inte** i mängden. Ingen linjärsökning krävs eftersom den smarta implementationen av datastrukturen medger snabb uppslagning (eng. *lookup*) av ett element.

Förändringsbar mängd

Med en **förändringsbar** mängd kan man stegvis utöka på plats.

```
scala> val mängd = scala.collection.mutable.Set.empty[Int]

scala> for i <- 1 to 1_000_000 do mängd.addOne(i)

scala> mängd.contains(-1)   // samma som mängd(-1) eller mängd.apply(-1)
```

En **mängd** är **snabb** på att avgöra om ett element **finns eller inte** i mängden. Ingen linjärsökning krävs eftersom den smarta implementationen av datastrukturen medger snabb uppslagning (eng. *lookup*) av ett element.

Men i en sekvens krävs linjärsökning vid innehållstest:

```
scala> val sekvens = (1 to 1_000_000).toVector

scala> sekvens.contains(-1)   // kräver linjärsökning ända till slutet
```

Förändringsbar mängd

Med en **förändringsbar** mängd kan man stegvis utöka på plats.

```
scala> val mängd = scala.collection.mutable.Set.empty[Int]

scala> for i <- 1 to 1_000_000 do mängd.addOne(i)

scala> mängd.contains(-1)   // samma som mängd(-1) eller mängd.apply(-1)
```

En **mängd** är **snabb** på att avgöra om ett element **finns eller inte** i mängden. Ingen linjärsökning krävs eftersom den smarta implementationen av datastrukturen medger snabb uppslagning (eng. *lookup*) av ett element.

Men i en sekvens krävs linjärsökning vid innehållstest:

```
scala> val sekvens = (1 to 1_000_000).toVector

scala> sekvens.contains(-1)   // kräver linjärsökning ända till slutet
```

Övning: Testa själv att mäta tidsskillnaden med hjälp av:

```
def nanos(b: => Unit) = { val t0 = System.nanoTime; b; System.nanoTime - t0 }
```

Speciella metoder på förändringsbar mängd

Förändringsbara mängder har metoder som ändrar på plats:

```
scala> val s = scala.collection.mutable.Set.empty[Int]

scala> s.addOne(1)     // finns även under namnet += om du gillar operator-notion
val res0: scala.collection.mutable.Set[Int] = HashSet(1)

scala> s.addOne(2).addOne(3).addOne(3).addOne(42) // addOne returnerar this
val res1: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 42)

scala> res0.eq(res1)  // samma instans av mutable.Set (ingen ny har skapats)
val res2: Boolean = true

scala> s.addAll(Vector(3, 4, 5))  // finns även ++= om du gillar operator-notion
val res3: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 4, 5, 42)

scala> s.subtractOne(1).subtractAll(List(1,2,3))  // finns även -= och --=
val res4: scala.collection.mutable.Set[Int] = HashSet(4, 5, 42)

scala> s.filterInPlace(_ > 4)
val res5: scala.collection.mutable.Set[Int] = HashSet(5, 42)
```

`addOne`, `addAll`, `subtractOne`, `subtractAll`, `filterInPlace` returnerar `this` så du kan ändra på plats med kedjade anrop med punktnotation.

Nyckel-värde-tabell

Vad är en nyckel-värde-tabell?

- En **nyckel-värde-tabell** är en samling element som är **par** med: en **nyckel** av någon typ K och ett **värde** av någon typ V.
- En sådan tabell kan skapas ur en sekvens av par `(k, v)` där k är en nyckel och v är ett värde:

```
scala> val ålder = Map("Björn" -> 42, "Sandra" -> 35, "Kim" -> 19)
val ålder: Map[String, Int] = Map(Björn -> 42, Sandra -> 35, Kim -> 19)
```

- Tabellens nycklar utgör en mängd som ges av metoden `keySet`; nycklarna är **unika**.
- Elementen utgör **inte en sekvens** och har ingen speciell ordning; en nyckel-värde-tabell har ej längd, men en **storlek**; metoden `size` ger antalet element.

Vad är en nyckel-värde-tabell?

- En **nyckel-värde-tabell** är en samling element som är **par** med: en **nyckel** av någon typ K och ett **värde** av någon typ V.
- En sådan tabell kan skapas ur en sekvens av par `(k, v)` där k är en nyckel och v är ett värde:

```
scala> val ålder = Map("Björn" -> 42, "Sandra" -> 35, "Kim" -> 19)
val ålder: Map[String, Int] = Map(Björn -> 42, Sandra -> 35, Kim -> 19)
```

- Tabellens nycklar utgör en mängd som ges av metoden `keySet`; nycklarna är **unika**.
- Elementen utgör **inte en sekvens** och har ingen speciell ordning; en nyckel-värde-tabell har ej längd, men en **storlek**; metoden `size` ger antalet element.
- En tabell kan ses som en uppslagsfunktion (eng. *dictionary*): alltså en funktion K **=>** V som ger ett värde givet en nyckel.

Den fantastiska nyckel-värde-tabellen `Map`

- En **nyckel-värde-tabell** (eng. *key-value table*) är en slags generaliserad vektor där man kan ”indexera” med godtycklig typ.
- Kallas även **hashtabell** (eng. *hash table*), **lexikon** (eng. *Dictionary*) eller **mapp** (eng. *Map*) (det blir lätt sammanblandning med metoden `map`).
- Om man vet nyckeln kan man slå upp värdet **snabbt**, på liknande sätt som indexering sker snabbt i en vektor givet heltalsindex.
- Denna datastruktur är **mycket användbar** och fungerar som en slags databas i kombination med filtrering, registrering, etc.

Oföränderlig nyckel-värde-tabell

- **Skapa**: ge par till metoden `apply`

```
scala> var födelse = Map("C" -> 1972,  "C++" -> 1983, "C#" -> 2000,
  "Scala" -> 2004, "Java" -> 1995, "Javascript" -> 1995, "Python" -> 1991)
```

- **Läsa**: slå upp ett värde med hjälp av en nyckel

```
scala> val year = födelse.apply("Scala")
val year: Int = 2004
```

- **Uppdatera**: lägga till ett par, ersätta ett par

```
scala> födelse = födelse + ("Kotlin" -> 2011)
födelse: Map[String, Int] = HashMap(Scala -> 2004, C# -> 2000, Python -> 1991,
Javascript -> 1995, C -> 1972, C++ -> 1983, Kotlin -> 2011, Java -> 1995)
```

- **Ta bort** ett par via nyckeln (om finns, annars händer inget)

```
scala> födelse = födelse - "Python"
födelse: Map[String, Int] = HashMap(Scala -> 2004, C# -> 2000,
Javascript -> 1995, C -> 1972, C++ -> 1983, Kotlin -> 2011, Java -> 1995)
```

Fler exempel nyckel-värde-tabell

Några ofta förekommande metoder på tabeller:

- `xs.keySet` ger en mängd av alla nycklar
- `xs.map(f)` kör funktionen f på alla par (key, value) i **någon** ordning
- `xs.map((k, v) => k -> f(v))` kör funktionen f på alla **värden**

```
scala> val färg = Map("gurka" -> "grön", "tomat"->"röd", "aubergine"->"lila")
val färg: Map[String, String] =
  Map(gurka -> grön, tomat -> röd, aubergine -> lila)

scala> färg("gurka")
val res0: String = grön

scala> färg.keySet
val res1: Set[String] = Set(gurka, tomat, aubergine)

scala> val ärGrönSak = färg.map((k,v) => (k, v == "grön"))
val ärGrönSak: Map[String,Boolean] =
  Map(gurka -> true, tomat -> false, aubergine -> false)

scala> val baklängesFärg = färg.map((k, v) => k -> v.reverse)
val baklängesFärg: Map[String,String] =
  Map(gurka -> nörg, tomat -> dör, aubergine -> alil)
```

Från sekvens av par till tabell

```
scala> val xs = Vector(("Kim",42), ("Pam", 42), ("Kim", 50), ("Pam", 50))
val xs: Vector[(String, Int)] =
  Vector((Kim,42), (Pam,42), (Kim,50), (Pam,50))

scala> xs.toMap
val res0: Map[String, Int] =
  Map(Kim -> 50, Pam -> 50) // inga dublettnycklar

scala> val grupperaEfterNamn = xs.groupBy(_._1)
grupperaEfterNamn: Map[String,Vector[(String, Int)]] =
  Map(Kim -> Vector((Kim,42), (Kim,50)), Pam -> Vector((Pam,42), (Pam,50)))

scala> val grupperaEfterÅlder = xs.groupBy(_._2)
grupperaEfterÅlder: Map[Int,Vector[(String, Int)]] =
  Map(50 -> Vector((Kim,50), (Pam,50)), 42 -> Vector((Kim,42), (Pam,42)))
```

Övning: Implementera en `Multimap`

- Om du lägger till ett värde i en *vanlig* `Map` så ersätts värdet:

```
scala> val m = Map(1 -> 2, 1 -> 3, 2 -> 1, 2 -> 2)
val m: Map[Int, Int] = Map(1 -> 3, 2 -> 2)   //senaste värdet gäller
```

  ...men ibland vill vi i stället lagra alla tillagda värden.

- En **multimap** är en speciell nyckel-värde-tabell där värdena utgör en samling (ofta en mängd).
- En multimap samlar alla värden som har samma nyckel.

```
1  scala> val mm = Multimap(1 -> 2, 1 -> 3, 2 -> 1, 2 -> 2)
2  val mm: Multimap[Int, Int] = Multimap(1 -> Set(2, 3), 2 -> Set(1, 2))
```

Övning: Implementera en multimap som fungerar som ovan, med hjälp av en case-klass med attributet `toMap` som är en oföränderlig nyckel-värde-tabell där värdena är en mängd.
Tips: Använd `groupBy`

Lösning: Multimap

```
case class Multimap[K, V] private (toMap: Map[K,Set[V]]):
  def apply(k: K): Set[V] = toMap(k)

  def +(kv: (K, V)): Multimap[K, V] = kv match
    case (k, v) if toMap.isDefinedAt(k) => Multimap(toMap.updated(k, toMap(k) + v))
    case (k, v) => Multimap(toMap + (k -> Set(v)))

  override def toString = toMap.mkString("Multimap(", ", ")")

object Multimap:
  def apply[K, V](kvs: (K,V)*): Multimap[K, V] =
    new Multimap(kvs.groupBy(_._1).map((k,xs) => k -> xs.map(_._2).toSet))
```

Speciella metoder på förändringsbar tabell

Både `Set` och `Map` finns i **förändringsbara** varianter med extra metoder för uppdatering av innehållet ”på plats” utan att nya samlingar skapas.

```
scala> import scala.collection.mutable

scala> val mm = mutable.Map.empty[String, String]
val mm: scala.collection.mutable.Map[String, String] = HashMap()

scala> mm.addOne("hej" -> "svejs")
val res0: scala.collection.mutable.Map[String, String] =
  HashMap(hej -> svejs)

scala> mm.addAll(Seq("abra" -> "kadabra", "ada" -> "lovelace"))
val res1: scala.collection.mutable.Map[String, String] =
  HashMap(hej -> svejs, abra -> kadabra, ada -> lovelace)

scala> mm("abra")
val res2: String = kadabra
```

Metoden ++= samma som addAll; används gärna med operator-notation:

```
mm ++= Seq("hej" -> "san", "abra" -> "kada", "bra" -> "scala")
```

Tips inför veckans uppgifter

Övning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling: Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. `toSet` på en mängd, eller `toMap` på en hashtabell, eller `toVector` på en `ArrayBuffer` eller `Array`.

Exempel där lösningen har nytta av lokal förändring på plats:

```
def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) = ???
```

```
scala> kastaTärningTillsAllaUtfallUtomEtt()
val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))
```

Övning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling: Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. `toSet` på en mängd, eller `toMap` på en hashtabell, eller `toVector` på en `ArrayBuffer` eller `Array`.

```
def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) =
  /*
    låt s vara en tom förändringsbar heltalsmängd
    låt n vara noll
    så länge mängden s är mindre än sidor - 1 gör:
      lägg till ett nytt tärningskast i s
      uppdatera n så att vi räknar hur många slumptal som dragits
  */
  (n, s.toSet)   // notera toSet som ger oföränderlig mängd
```

```
scala> kastaTärningTillsAllaUtfallUtomEtt()
val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))
```

Lösning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling: Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. `toSet` på en mängd, eller `toMap` på en hashtabell, eller `toVector` på en `ArrayBuffer` eller `Array`.

```
def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) =
  val s = scala.collection.mutable.Set.empty[Int] //förändringsbar lokalt
  var n = 0
  while s.size < sidor - 1 do
    s.addOne(util.Random.nextInt(sidor) + 1)
    n += 1
  (n, s.toSet)   // notera toSet som ger oföränderlig mängd
```

```
scala> kastaTärningTillsAllaUtfallUtomEtt()
val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))
```

I veckans uppgifter används detta i en s.k. **builder**: Först bygga upp en förändringsbar struktur i `FreqMapBuilder` steg för steg, och sedan, då alla tillägg är gjorda, övergå till oföränderlig struktur `Map[String, Int]`.

Metoden `sliding`

Metoden `sliding(n)` skapar med ett ”glidande fönster” en sekvens av delsekvenser av längd n genom att ”svepa fönstret” från början till slut:

```
scala> val xs = "fem myror är fler än fyra elefanter".split(' ').toVector
val xs: Vector[String] = Vector(fem, myror, är, fler, än, fyra, elefanter)

scala> xs.sliding(2).toVector
val res0: Vector[Vector[String]] =
  Vector(Vector(fem, myror), Vector(myror, är), Vector(är, fler),
      Vector(fler, än), Vector(än, fyra), Vector(fyra, elefanter))

scala> xs.sliding(3).toVector
val res1: Vector[Vector[String]] =
  Vector(Vector(fem, myror, är), Vector(myror, är, fler),
    Vector(är, fler, än), Vector(fler, än, fyra),
      Vector(än, fyra, elefanter))
```

Denna metod har du nytta av på veckans laboration!
(se fler exempel på övning)

Metoderna zipWithIndex, groupBy

```
scala> val kort = Vector("Knekt", "Dam", "Kung", "Äss")

scala> val kortIndex = kort.zipWithIndex.toMap
kortIndex: Map[String,Int] = Map(Knekt -> 0, Dam -> 1, Kung -> 2, Äss -> 3)

scala> kortIndex("Kung") > kortIndex("Knekt")
res0: Boolean = true

scala> kortIndex.map(p => p._1 -> (p._2 + 11))

scala> val tärningskast = Vector(1,2,3,4,5,6,2,4,6)

scala> val grupperaStörreÄnFyra = tärningskast.groupBy(_ > 4)
grupperaStörreÄnFyra: Map[Boolean,Vector[Int]] =
  Map(false -> Vector(1, 2, 3, 4, 2, 4), true -> Vector(5, 6, 6))

scala> val grupperaLika = tärningskast.groupBy(x => x)
grupperaLika: Map[Int,Vector[Int]] = Map(5 -> Vector(5), 1 -> Vector(1),
  6 -> Vector(6, 6), 2 -> Vector(2, 2), 3 -> Vector(3), 4 -> Vector(4, 4))

scala> val frekvens = tärningskast.groupBy(x => x).map((k,v) => k -> v.size)
frekvens: Map[Int,Int] = Map(5 -> 1, 1 -> 1, 6 -> 2, 2 -> 2, 3 -> 1, 4 -> 2)
```

Fler användbara samlingsmetoder

Exempel att öva på: räkna bokstäver i ord.
Undersök vad som händer i REPL:

```
val ord = "sex laxar i en laxask sju sjösjuka sjömän"
val uppdelad = ord.split(' ').toVector
val ordlängd = uppdelad.map(_.length)
val ordlängdMap = uppdelad.map(s => (s, s.size)).toMap
val grupperaEfterFörstaBokstav = uppdelad.groupBy(s => s(0))
val bokstäver = ord.toVector.filter(_ != ' ')
val antalX = bokstäver.count(_ == 'x')
val grupperade = bokstäver.groupBy(ch => ch)
val antal = grupperade.map(p => p._1 -> p._2.size)
//samma som ovan men utnyttjar "parameter untupling":
val antal2 = grupperade.map((k,v) => k -> v.size)
val sorterat = antal.toVector.sortBy(_._2)
val vanligast = antal.maxBy(_._2)
```

Serialisering och deserialisering

Serialisering och deserialisering

- Att **serialisera** innebär att **koda objekt** i minnet till en avkodningsbar **sekvens av symboler**, som kan lagras t.ex. i en fil på din hårddisk.
- Att **de-serialisera** innebär att **avkoda en sekvens av symboler**, t.ex. från en fil, och **återskapa objekt** i minnet.

Läsa text från fil och URL

I paketet `scala.io` finns singelobjektet `Source` med metoderna `fromFile` och `fromUrl` för läsning från fil resp. från URL, alltså Universal Resource Locator, som börjar t.ex. med `http://`

```
def läsFrånFil(filnamn: String): String =
  val s = scala.io.Source.fromFile(filnamn)
  try s.mkString finally s.close // säkerställ stängning även vid krasch

def läsRaderFrånFil(filnamn: String): Vector[String] =
  val s = scala.io.Source.fromFile(filnamn)
  try s.getLines.toVector finally s.close

def läsFrånWebbsida(url: String): String =
  val s = scala.io.Source.fromURL(url)
  try s.mkString finally s.close

def läsRaderWebbsida(url: String, kodning: String = "UTF-8"): Vector[String] =
  val s = scala.io.Source.fromURL(url, kodning) // läs med given teckenkodning
  try s.getLines.toVector finally s.close
```

Se vidare veckans övning. Exempel på annan teckenkodning: `"ISO-8859-1"`

Serialisering i modulen `introprog.IO`

- I kursens kodbibliotek `introprog` finns ett singelobjekt `IO` som samlar smidiga funktioner för serialisering och de-serialisering.
- Se api-dokumentation här:
  `http://cs.lth.se/pgk/api/`
  Sök på IO och klicka på singelobjektet.
- Se koden här:
  `https://github.com/lunduniversity/introprog-scalalib/blob/master/src/main/scala/introprog/IO.scala`
- Om du vill får du gärna använda `introprog.IO` istället för `scala.io.Source` på labben.

10 Arv och komposition

- Gäst: Gustaf Lundh, Axis
- Om grupplaborationen `snake`
- Vad är arv?
- Bastyp, `trait`, `extends`
- Subtypspolymorfism och dynamisk bindning
- Överskuggningsregler
- Algebraiska datatyper

Gäst: Gustaf Lundh, Axis

Kodgranskning under grupplaborationen snake

Grupplaborationen snake går över två läsveckor (w10–w11). Du ska:

- delta i framtagande av en gemensam checklista för kodgranskning,
- granska minst en annan gruppmedlems kod,
- bjuda in minst en annan gruppmedlem att granska din kod,
- ge konstruktiv feedback,
- ta emot konstruktiv feedback och försöka förbättra din kod,
- på redovisning redogöra för nyttan och utmaningarna med kodgranskningar utifrån dina egna erfarenheter.

Mer om kodgranskning i efterföljande kurser, t.ex. ”Programvaruutveckling i grupp” och ”Programvaruutveckling för stora system”

Gästföreläsning: "Kodgranskningar i praktiken"

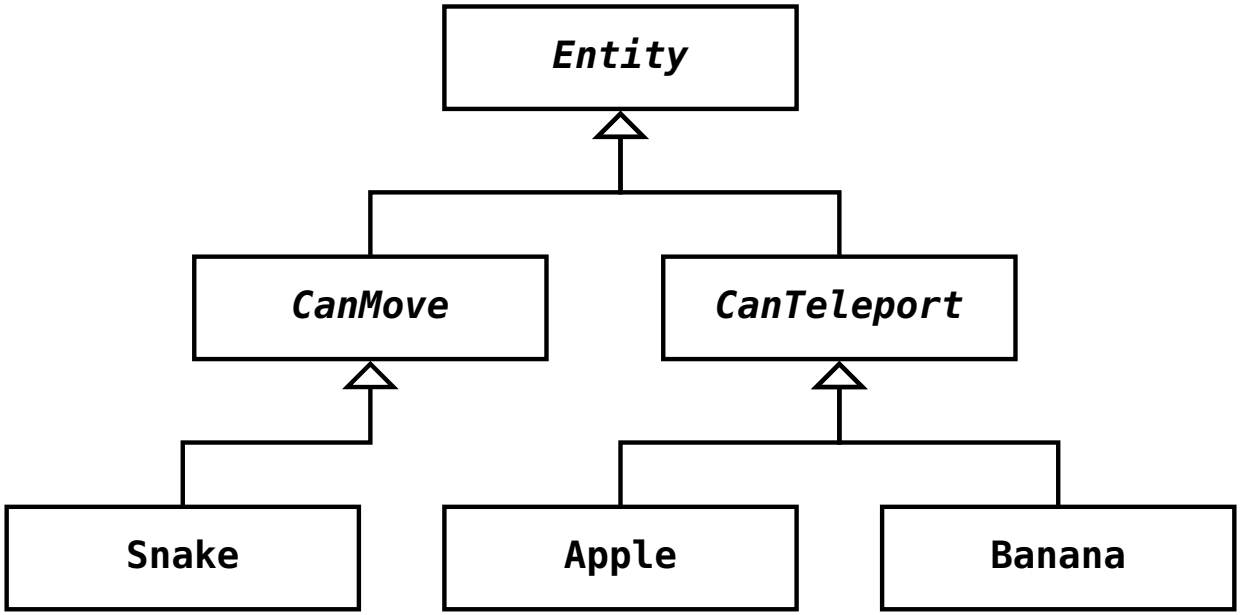
Hjärtligt välkommen:
Gustaf Lundh, Senior Developer, Axis

Om grupplaborationen snake

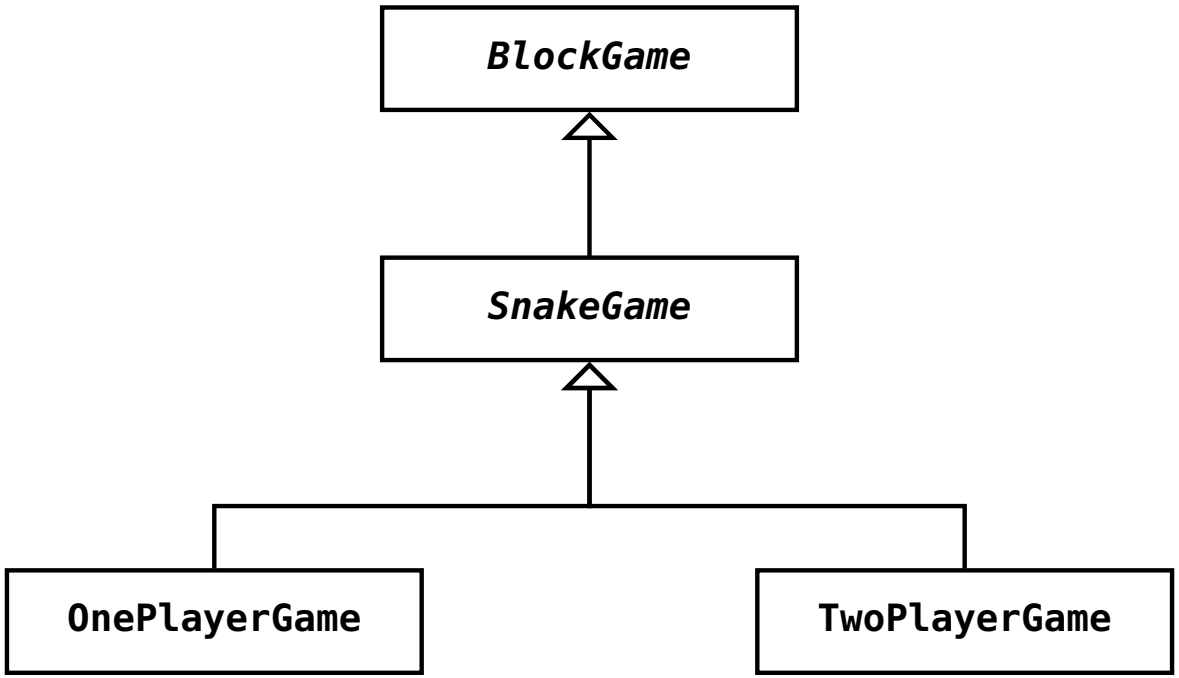
Grupplaboration: snake över 2 veckor

- Kunna använda arv.
- Kunna göra överskuggning av medlemmar i en supertyp.
- Kunna förklara begreppet dynamisk bindning.
- Kunna använda abstrakta klasser och skapa en klasshierarki.

Arvshierarki i snake: Olika typer av varelser



Arvshierarki i snake: Olika typer av spel



Krav vid respektive gruppstorlek

Krav som minst ska implementeras vid olika gruppstorlek:

Krav / **Antal personer**	1	2	3	4	5	6
`Player`	√	√	√	√	√	√
`OnePlayerGame`	√				√	√
`TwoPlayerGame`		√	√	√	√	√
`Snake`	√	√	√	√	√	√
`Apple`	√		√	√	√	√
`Banana`				√		√
`Monster`			√			√
`Settings`	√	√	√	√	√	√

- Uppgiften lämpar sig bäst för minst 3 gruppmedlemmar.
- Prata med handledare: slå ihop två små grupper till en med max 6 st.
- Om du har särskilda skäl, t.ex. många restlabbar, kan du efter godkännande från kursansvarig göra labben enskilt.

Instruktioner Grupplaboration (text ur kompendiet)

- Diskutera i din samarbetsgrupp hur ni ska dela upp koden mellan er i flera olika delar, som ni kan arbeta med var för sig. En sådan del kan vara en klass, en trait, ett objekt, ett paket, eller en funktion.
- Varje del ska ha en **huvudansvarig** individ.
- Arbetsfördelningen ska vara någorlunda jämnt fördelad mellan gruppmedlemmarna.
- Den som är huvudansvarig för en viss del redovisar den delen.
- Ni ska ta fram en gruppgemensam checklista för kodgranskning. Varje gruppmedlem ska granska minst en annan gruppmedlems kod enligt checklistan.
- Grupplaborationen görs över **två veckor** uppdelat på två delredovisningar. Vid första redovisningen ska arbetsupplägget och pågående utveckling redovisas. Vid andra tillfället ska de färdig lösningarna presenteras av respektive huvudansvarig individ.
- Vid första redovisningen ska du redogöra för handledaren hur ni delat upp koden och vem som är huvudansvarig för vad och vad ditt ansvar omfattar, samt hur ni jobbar praktiskt med att synkronisera er utveckling.
- Grupplaborationen är en **extra stor uppgift** och grupparbetet behöver ledtid för att ni ska hinna koordinera er sinsemellan. Du behöver därför planera för att arbeta med något i grupplabben i stort sett varje dag under de tillgängliga veckorna, och vara redo att bidra i diskussioner.

Redovisning på obligatorisk schemalagd labbtid

Första `snake`-veckan (w10):

- Redovisa uppdelning av uppgiften mellan gruppmedlemmar och hur långt du har kommit med din del. Visa en skriftlig plan för implementationsarbetet för andra veckan, med valda deluppgifter och sluttidpunkter.
- Redovisa arbetet med granskningar, minst checklista och planering för arbetet med granskningar i andra veckan. Bra om ni du gjort någon granskning.
- Jobba vidare med `snake` och få hjälp av handledare om du kör fast.

Andra `snake`-veckan (w11):

- Redogör för uppdelningen av uppgiften och avgränsningen av ditt huvudansvar.
- Förklara övergripande hela kodbasens struktur och syftet med de olika kod-delarna.
- Förklara mer ingående de delar där du bidragit till implementationen.
- Gör en kort demonstration av det körande systemet och visa vilka valfria uppgifter ni valt att göra.
- Beskriv utmaningar med systemutveckling i grupp utifrån dina egna erfarenheter.
- Redogör för nyttan och utmaningarna med kodgranskningar utifrån dina egna erfarenheter.

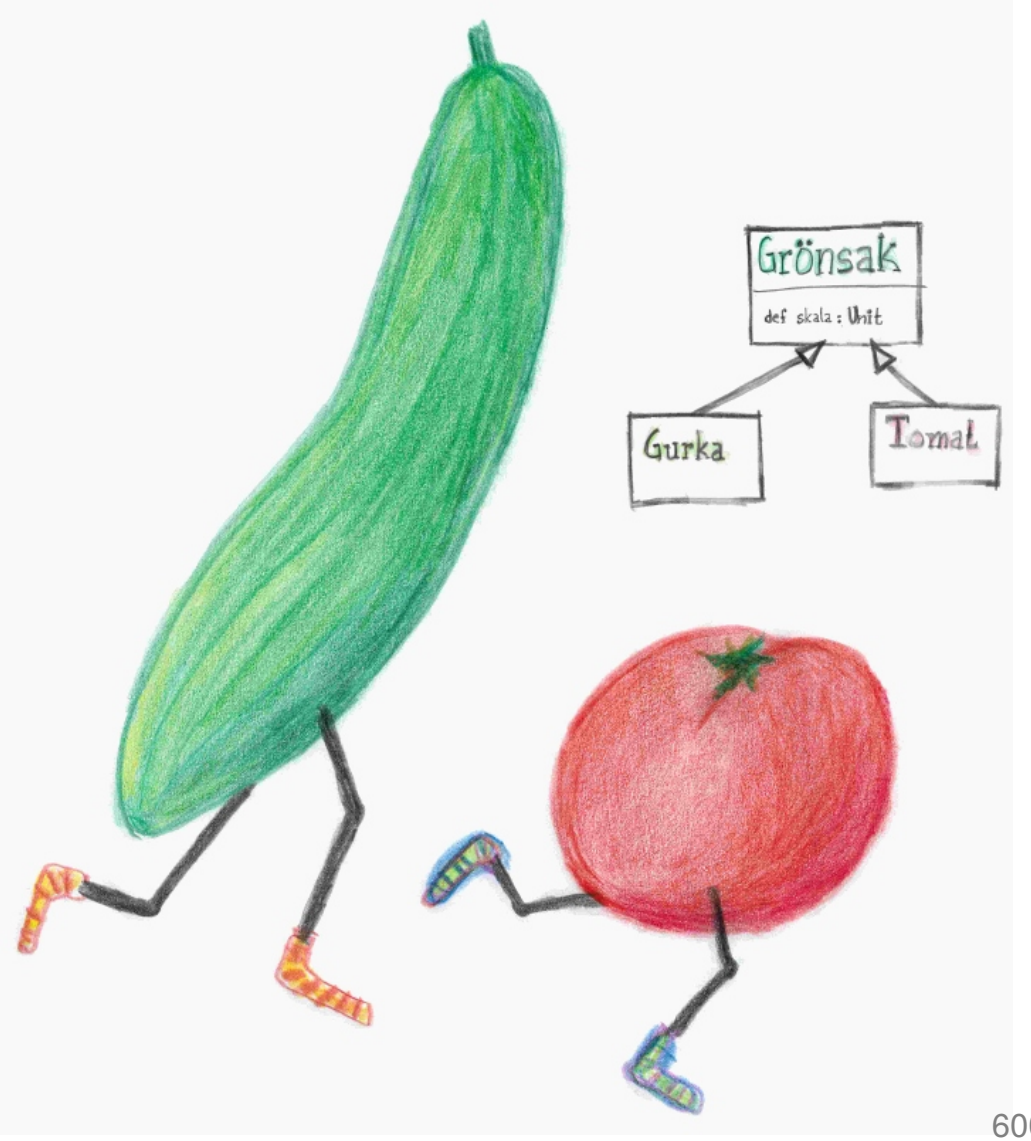
Övning w10: `inheritance`

- Kunna deklarera och använda en arvshierarki i flera nivåer med nyckelordet **`extends`**.
- Känna till synlighetsregler vid arv och nyttan med privata och skyddade medlemmar och nyckelorden **`private`** och **`protected`**.
- Kunna deklarera och använda överskuggade medlemmar och nyckelordet **`override`**.
- Kunna deklarera och använda en hierarki av klasser där konstruktorparametrar överförs till superklasser.
- Kunna deklarera och använda uppräknade värden med case-objekt och gemensam bastyp.
- Känna till reglerna som gäller vid överskuggning av olika sorters medlemmar.
- Känna till nyttan med finala klasser och finala attribut och nyckelordet **`final`**.
- Kännedom om dessa begrepp: bastyp, sypertyp, subtyp, körtidstyp, dynamisk bindning, polymorfism, trait, inmixning, överskuggad medlem, anonym klass, skyddad medlem, abstrakt medlem, abstrakt klass, referenstyp, värdetyp.

Vad är arv?

Vad är arv?

Arv (eng. *inheritance*) beskriver relationen *X* **är en** *Y*



Varför behövs arv?

- Man kan använda arv för att dela upp kod i:
  - **generella** (gemensamma) delar och
  - **specifika** (specialanpassade) delar.
- Man kan åstadkomma **kontrollerad flexibilitet**:
  - Klientkod kan **utvidga** (eng. *extend*) ett givet API med egna specifika tillägg.
- Man kan använda arv för att deklarera en gemensam **bastyp** så att variabler och generiska samlingar kan ges en lagom specifik elementtyp.
  - Det räcker att man vet bastypen för att kunna nå gemensamma medlemmar för alla element i samlingen.
  - Exempel: Alla grönsaker har en vikt.

Klassdiagram med UML (Unified Modeling Language)

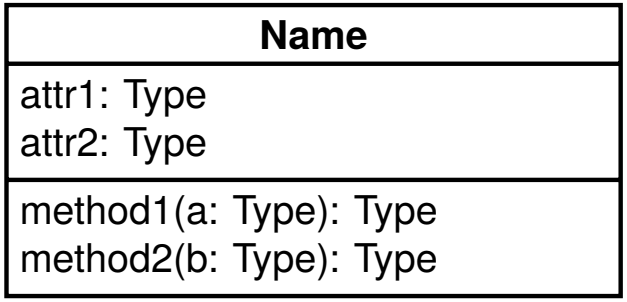
UML Klassdiagram:

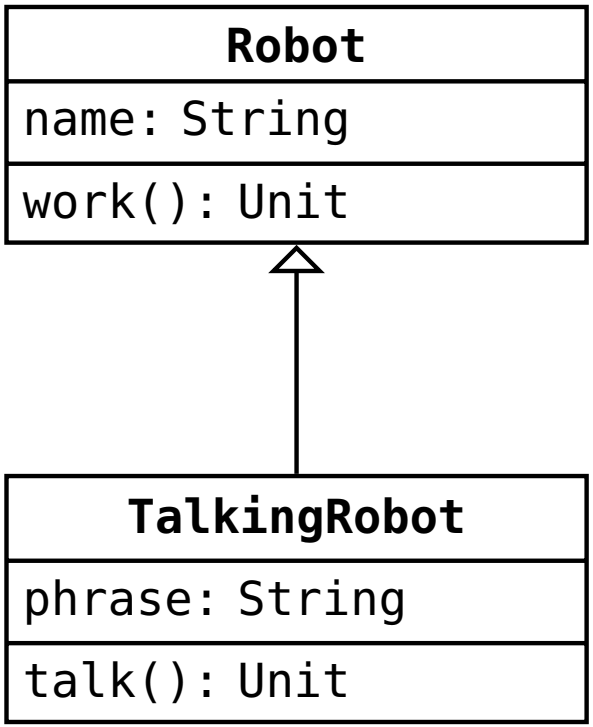
Name
attr1: Type attr2: Type
method1(a: Type): Type method2(b: Type): Type

Mer om UML-diagram i senare kurser.
en.wikipedia.org/wiki/Class_diagram

Klassdiagram med UML (Unified Modeling Language)

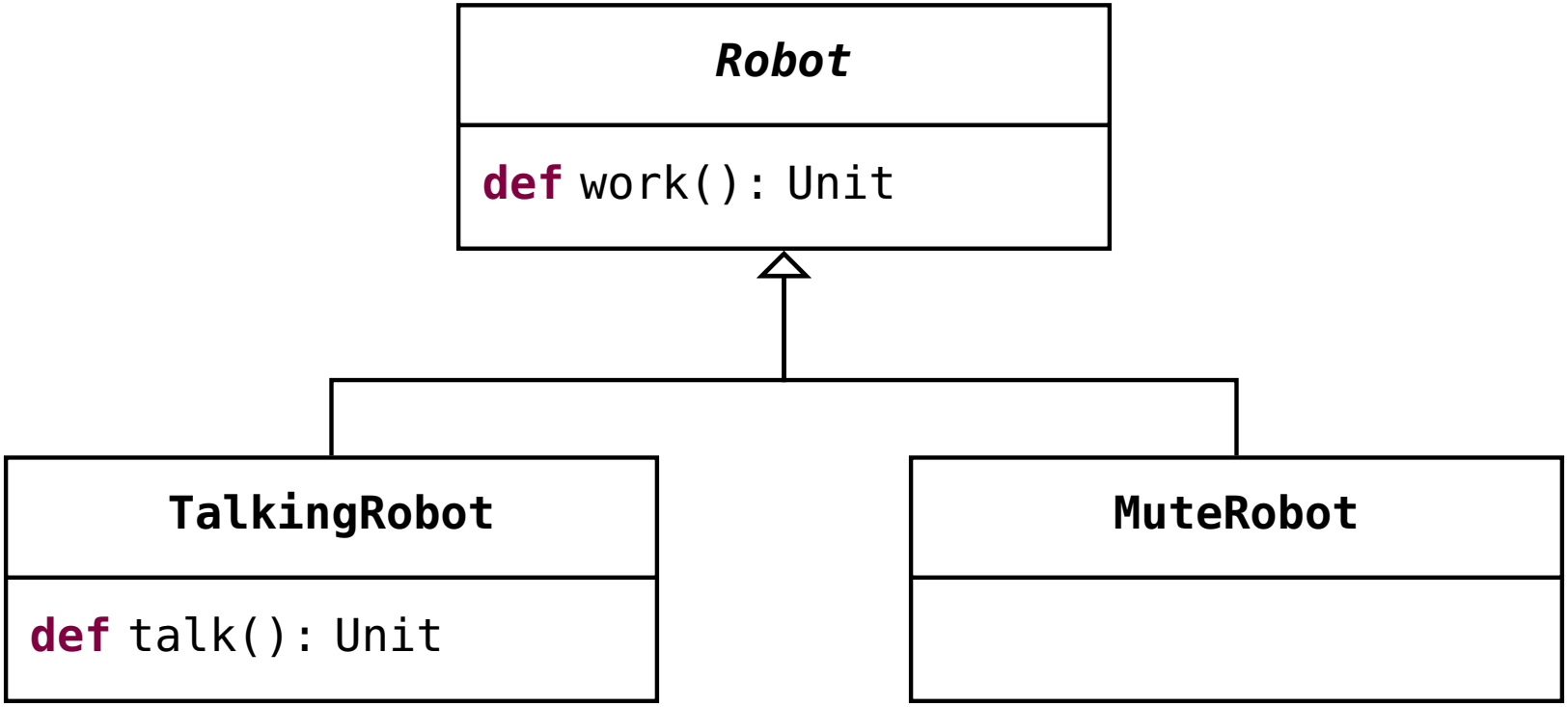
UML Klassdiagram:





Mer om UML-diagram i senare kurser.
en.wikipedia.org/wiki/Class_diagram

Exempel: Robot som bastyp för två subtyper

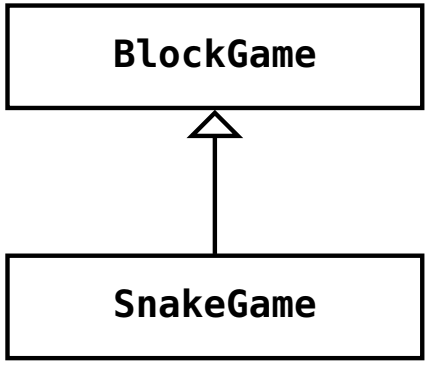


(Ibland utelämnar man attribut och/eller metoder i ett UML-diagram för att minska antalet detaljer i modellen och ge en överblick.)

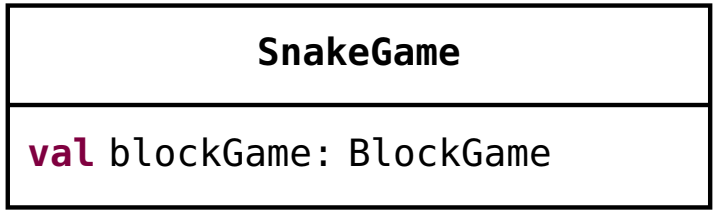
Alternativ till arv: komposition

- Som alternativ till att klassen X ärver klassen Y kan man i stället använda **komposition** (eng. *composition*), som innebär att klassen X **har ett attribut** som **refererar** till klassen Y.

Exempel på arv i `snake`-labben:
SnakeGame **är** ett BlockGame



Ett alternativ vore **komposition**:
SnakeGame **har** ett BlockGame



- Komposition där **X har en Y** är ofta ett bättre alternativ än arv om:
  1. det *inte* finns en tydlig **X-är-en-Y**-relation
  2. det *inte* behövs en **gemensam bastyp**
  3. det *inte* är önskvärt ärva och exponera *alla* Y:s medlemmar via X

Exempel på komposition i snake-labben

En `Player` **har en** [19] `snake`.

Player
`val snake: Snake`

```
class Player(val snake: Snake)
```

[19] Läs mer om komposition och de relaterade begreppen aggregering, association, ägarskap etc. här: `https://en.wikipedia.org/wiki/Object_composition`

Behovet av gemensam bastyp

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

scala> case class Tomat(vikt: Int)

scala> val gurkor = Vector(Gurka(200), Gurka(300))
val gurkor: Vector[Gurka] = Vector(Gurka(200), Gurka(300))

scala> gurkor.map(_.vikt)
res0: Vector[Int] = Vector(200, 300)

scala> val grönsaker = Vector(Gurka(200), Tomat(42))
val grönsaker: Vector[Gurka | Tomat] = Vector(Gurka(200), Tomat(42))

scala> grönsaker.map(_.vikt)
-- [E008] Not Found Error: -------------------------
1 |grönsaker.map(_.vikt)
  |              ^^^^^^
  |              value vikt is not a member of Gurka | Tomat
```

Hur får vi detta att fungera som vi vill?

Behovet av gemensam bastyp

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

scala> case class Tomat(vikt: Int)

scala> val gurkor = Vector(Gurka(200), Gurka(300))
val gurkor: Vector[Gurka] = Vector(Gurka(200), Gurka(300))

scala> gurkor.map(_.vikt)
res0: Vector[Int] = Vector(200, 300)

scala> val grönsaker = Vector(Gurka(200), Tomat(42))
val grönsaker: Vector[Gurka | Tomat] = Vector(Gurka(200), Tomat(42))

scala> grönsaker.map(_.vikt)
-- [E008] Not Found Error: --------------------------
1 |grönsaker.map(_.vikt)
  |              ^^^^^^
  |              value vikt is not a member of Gurka | Tomat
```

Hur får vi detta att fungera som vi vill?
→ Skapa en bastyp **bastyp** med gemensamt attribut `vikt`!

Varför syns inte gemensam medlem i en typunion?

```
scala> val grönsaker = Vector(Gurka(200), Tomat(42))
val grönsaker: Vector[Gurka | Tomat] = Vector(Gurka@15f11bfb,Tomat@16a499d1)

scala> grönsaker.map(_.vikt)
-- [E008] Not Found Error: --------------------------
1 |grönsaker.map(_.vikt)
  |              ^^^^^^
  |              value vikt is not a member of Gurka | Tomat
```

- Typerna `Grönsak` och `Tomat` är **orelaterade**. (`AnyRef` saknar `vikt`)
- En medlem som råkar ha samma namn har inte alltid samma innebörd.
- I Scala måste du explicit relatera medlemmarna genom **arv** av bastyp med gemensam medlem, eller så får du **matcha** på unionens delar.

Varför syns inte gemensam medlem i en typunion?

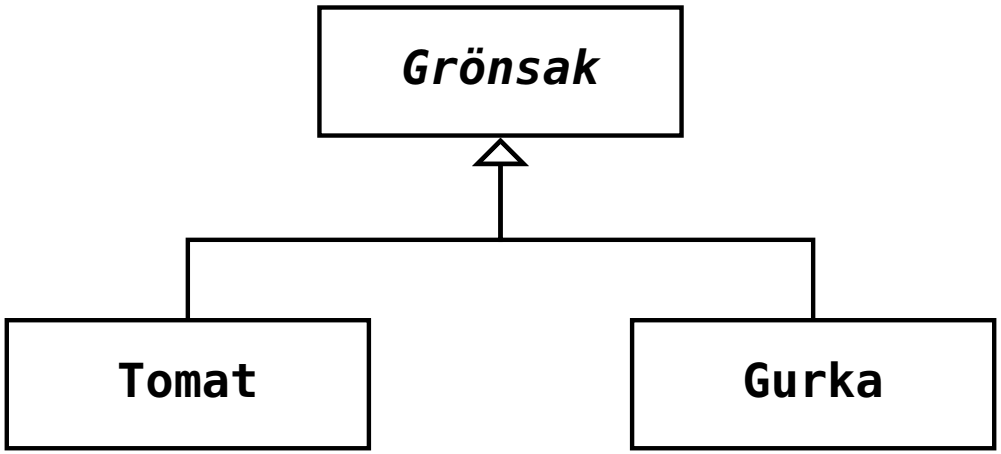
```
scala> val grönsaker = Vector(Gurka(200), Tomat(42))
val grönsaker: Vector[Gurka | Tomat] = Vector(Gurka@15f11bfb,Tomat@16a499d1)

scala> grönsaker.map(_.vikt)
-- [E008] Not Found Error: --------------------------
1 |grönsaker.map(_.vikt)
  |              ^^^^^^
  |              value vikt is not a member of Gurka | Tomat
```

- Typerna `Grönsak` och `Tomat` är **orelaterade**. (`AnyRef` saknar `vikt`)
- En medlem som råkar ha samma namn har inte alltid samma innebörd.
- I Scala måste du explicit relatera medlemmarna genom **arv** av bastyp med gemensam medlem, eller så får du **matcha** på unionens delar.
- Du kan erbjuda en sådan matchning som en **extensionsmetod**:

```
scala> extension (gEllerT: Gurka | Tomat) def vikt = gEllerT match
         case g: Gurka => g.vikt
         case t: Tomat => t.vikt

scala> val vikter = grönsaker.map(_.vikt)
val vikter: Vector[Int] = Vector(200, 42)
```

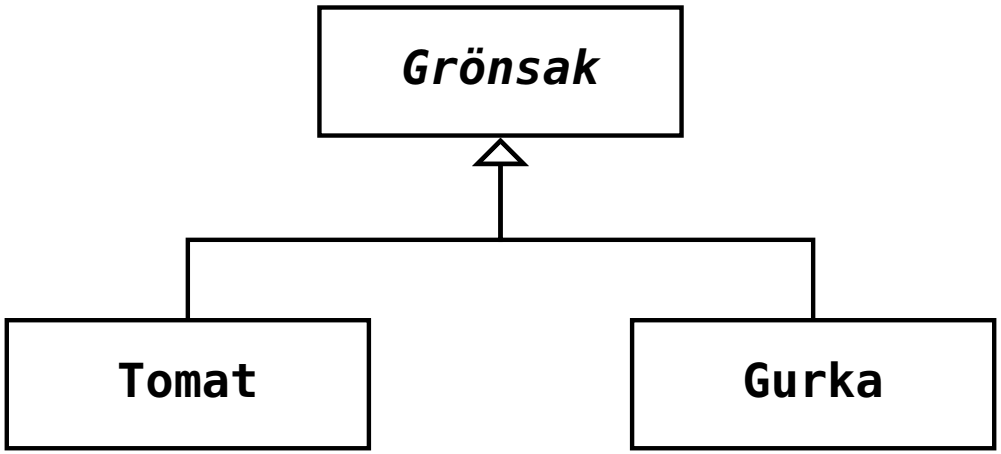
Skapa en gemensam bastyp med arv

Typen ***`Grönsak`*** är en **bastyp** i nedan arvshierarki:



Skapa en gemensam bastyp med arv

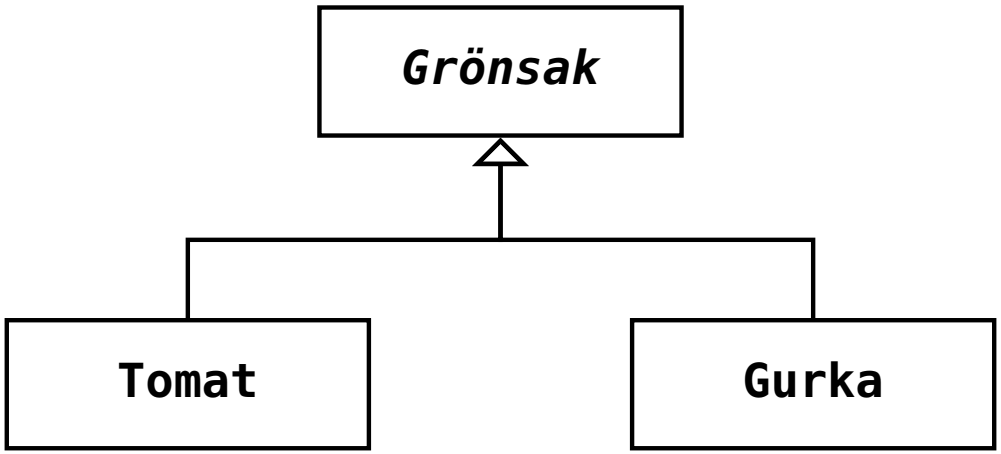
Typen ***`Grönsak`*** är en **bastyp** i nedan arvshierarki:



Pilen ↑ betecknar **arv** och utläses ”**är en**”

Skapa en gemensam bastyp med arv

Typen ***`Grönsak`*** är en **bastyp** i nedan arvshierarki:



Pilen ↑ betecknar **arv** och utläses ”**är en**”

Typerna `Tomat` och `Gurka` är **subtyper** typen `Grönsak`.
Bastyp som ej ska instansieras direkt är **abstrakt** (visas med *kursiv* stil).

Bastyp, `trait`, `extends`

Skapa en gemensam bastyp med `trait` och `extends`

Med en **`trait`** `Grönsak` kan klasserna `Gurka` och `Tomat` få en gemensam abstrakt **bastyp** genom att båda **subtyperna** gör **`extends`** `Grönsak`:

```
scala> trait Grönsak

scala> case class Gurka(vikt: Int) extends Grönsak

scala> case class Tomat(vikt: Int) extends Grönsak

scala> val grönsaker = Vector(Gurka(200), Tomat(42))
val grönsaker: Vector[Grönsak] = Vector(Gurka(200), Tomat(42))
```

Skapa en gemensam bastyp med `trait` och `extends`

Med en **`trait`** `Grönsak` kan klasserna `Gurka` och `Tomat` få en gemensam abstrakt **bastyp** genom att båda **subtyperna** gör **`extends`** `Grönsak`:

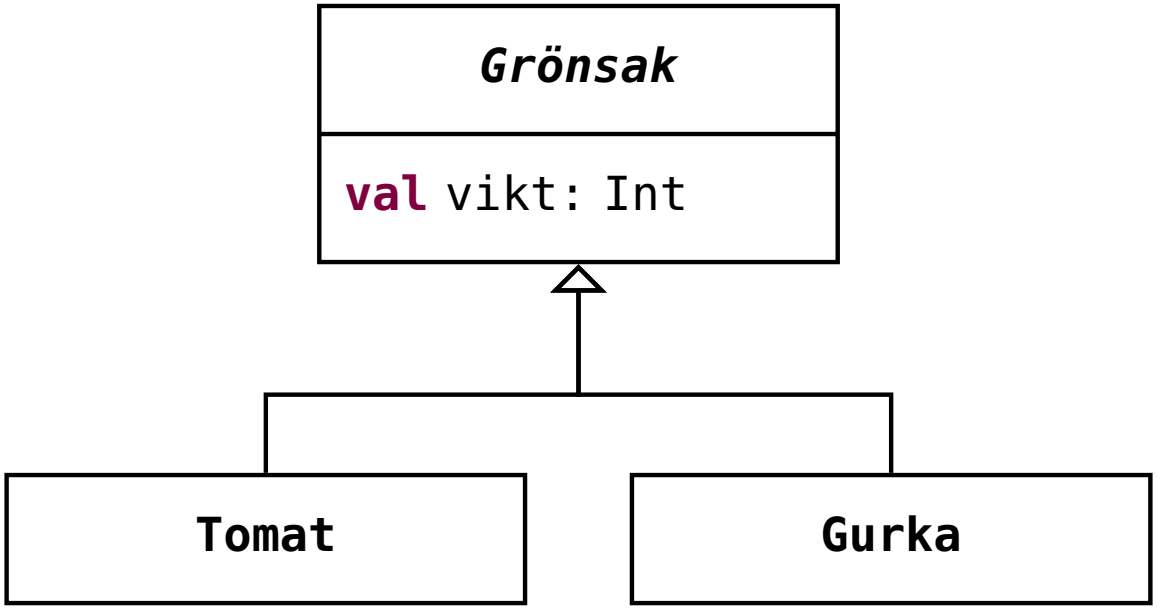
```
scala> trait Grönsak

scala> case class Gurka(vikt: Int) extends Grönsak

scala> case class Tomat(vikt: Int) extends Grönsak

scala> val grönsaker = Vector(Gurka(200), Tomat(42))
val grönsaker: Vector[Grönsak] = Vector(Gurka(200), Tomat(42))
```

Men det är ännu **inte** som vi vill ha det:

```
scala> grönsaker.map(_.vikt)
-- [E008] Not Found Error: ----------------------------
1 |grönsaker.map(_.vikt)
  |              ^^^^^^
  |              value vikt is not a member of Grönsak
```

En gemensam bastyp med gemensamma delar

Placera gemensamma medlemmar i bastypen:



- Alla grönsaker har nu attributet `val vikt`.
- Det specifika värdet på vikten definieras **inte** i bastypen.
- Medlemen `vikt` kallas **abstrakt** eftersom den **saknar implementation** och kan därför inte instansieras direkt.

Placera gemensamma delar i bastypen

Vi inkluderar det gemensamma attributet `val vikt` som en **abstrakt medlem** i bastypen:

```
trait Grönsak:
  val vikt: Int    // implementation saknas, inget =

case class Gurka(vikt: Int) extends Grönsak

case class Tomat(vikt: Int) extends Grönsak
```

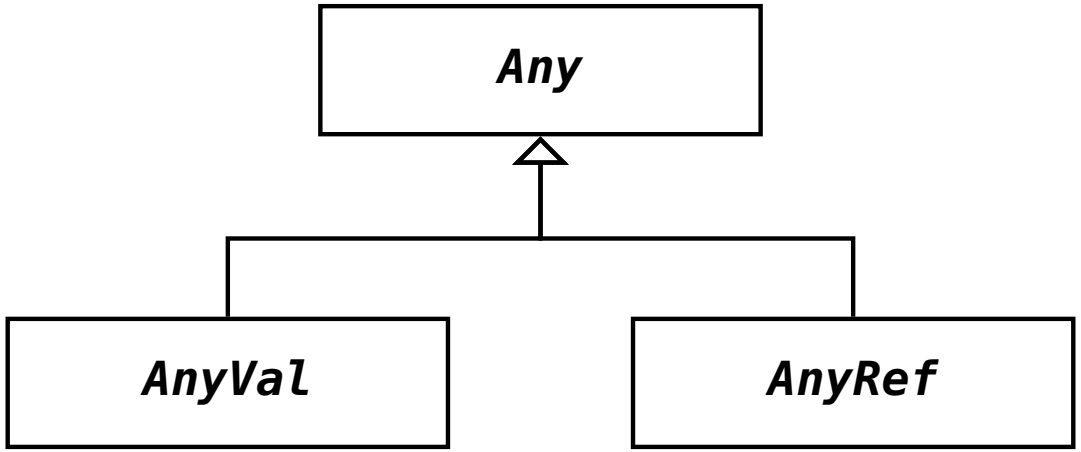
Nu har du explicit sagt till kompilatorn att du vill att alla grönsaker har en vikt:

```
scala> val grönsaker = Vector(Gurka(200), Tomat(42))
val grönsaker: Vector[Grönsak] = Vector(Gurka(200), Tomat(42))

scala> grönsaker.map(_.vikt)
val res0: Vector[Int] = Vector(200, 42)
```

Den abstrakta medlemmen `vikt` i den abstrakta typen `Grönsak` **implementeras** i en konkret subklass, här som en klassparameter.

Scalas typhierarki

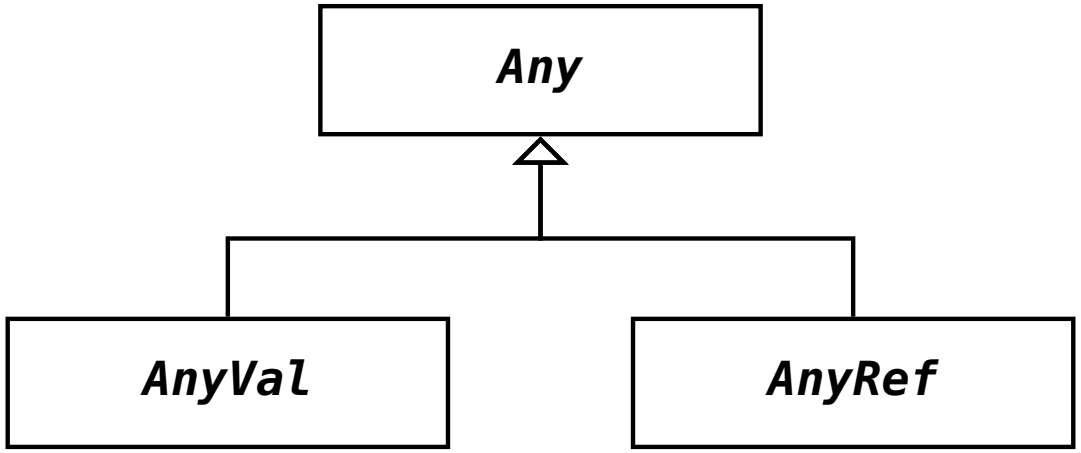
En förenklad bild av den översta delen av typhierarkin i Scala:



- De numeriska typerna `Int`, `Double`, etc är subtyper till **`AnyVal`** och kallas **värdetyper** och lagras på ett speciellt, effektivt sätt i minnet.
- Alla dina egna klasser är subtyper till **`AnyRef`** och kallas **referenstyper** och kan (direkt eller indirekt) konstrueras med **`new`**.
- `AnyRef` motsvaras av **`java.lang.Object`** i JVM.

Scalas typhierarki

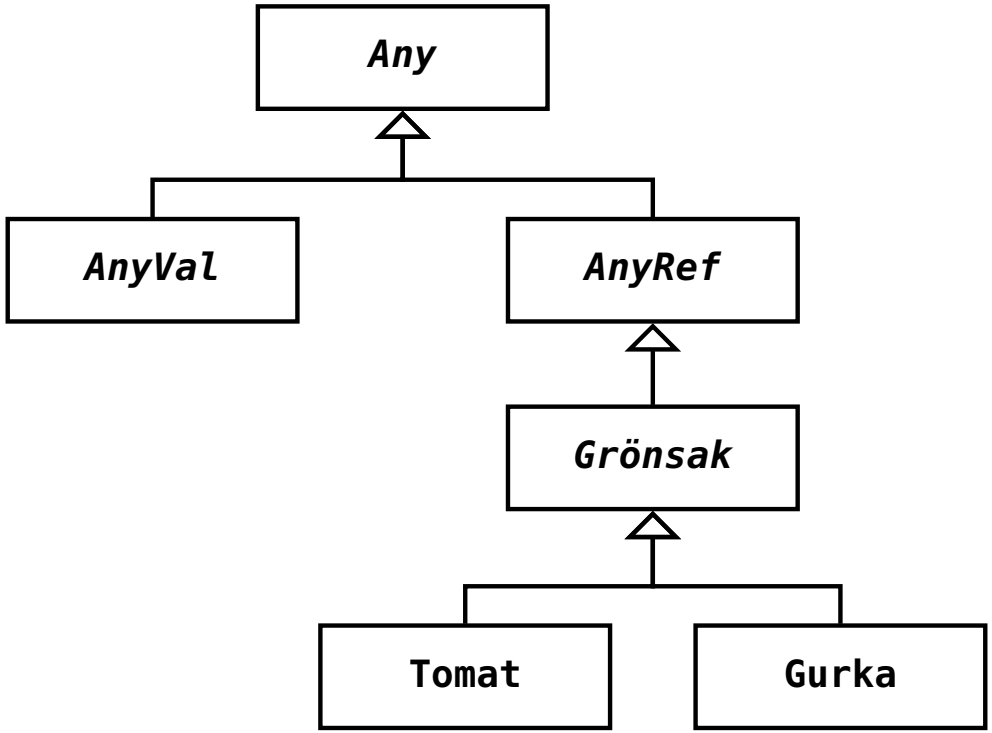
En förenklad bild av den översta delen av typhierarkin i Scala:



- De numeriska typerna `Int`, `Double`, etc är subtyper till **`AnyVal`** och kallas **värdetyper** och lagras på ett speciellt, effektivt sätt i minnet.
- Alla dina egna klasser är subtyper till **`AnyRef`** och kallas **referenstyper** och kan (direkt eller indirekt) konstrueras med **`new`**.
- `AnyRef` motsvaras av **`java.lang.Object`** i JVM.
- (Det finns även `Matchable` som är subtyp till `Any` och supertyp till `AnyRef` och `AnyVal`. Typen `Matchable` behövs för att skilja mellan typer som kan undersökas med mönstermatchning och andra s.k. **opaka typer** (överkurs).)

Implicita supertyper till dina egna klasser

Alla dina egna typer ingår underförstått i Scalas typhierarki:



Vad är en trait?

- **Trait** betyder **egenskap**.
- En trait liknar en klass, **men** speciella regler gäller:
  - den **kan** innehålla delar som **saknar implementation**
  - den **kan mixas** med flera andra traits så att olika koddelar kan kombineras på flexibla sätt.
  - den **kan inte** instansieras direkt.
  - den **kan** ha **parametrar**[20] på samma sätt som klasser.

[20] I gamla Scala 2 kan traits ej ha parametrar

Vad används en trait till?

En **trait** kan användas för att skapa en bastyp som kan vara hemvist för gemensamma delar hos subtyper:

```
trait Bastyp { val x = 42 }                    // Bastyp har medlemmen x
class Subtyp1 extends Bastyp { val y = 43 } // Subtyp1 ärver x, har även y
class Subtyp2 extends Bastyp { val z = 44 } // Subtyp2 ärver x, har även z
```

Vad används en trait till?

En **trait** kan användas för att skapa en bastyp som kan vara hemvist för gemensamma delar hos subtyper:

```
trait Bastyp { val x = 42 }                    // Bastyp har medlemmen x
class Subtyp1 extends Bastyp { val y = 43 } // Subtyp1 ärver x, har även y
class Subtyp2 extends Bastyp { val z = 44 } // Subtyp2 ärver x, har även z
```

```
scala> val a = new Subtyp1
val a: Subtyp1 = Subtyp1@51016012

scala> a.x
val res0: Int = 42

scala> a.y
val res1: Int = 43

scala> a.z
-- Error:
  value z is not a member of Subtyp1

scala> new Bastyp
--Error:
  Bastyp is a trait; it cannot be instantiated
```

En trait kan ha abstrakta medlemmar

```
trait X { val x: Int }   // x är abstrakt, d.v.s. saknar implementation
class A extends X { val x = 42 }   // x ges en implementation
class B extends X { val x = 43 }   // x ges en annan implementation
```

En trait kan ha abstrakta medlemmar

```scala
trait X { val x: Int }   // x är abstrakt, d.v.s. saknar implementation
class A extends X { val x = 42 }   // x ges en implementation
class B extends X { val x = 43 }   // x ges en annan implementation
```

```
scala> val a = new A
val a: A = A@5faeada1

scala> val b = B()    // fungerar utan new men då behövs ()
val b: B = B@cb51256

scala> val xs = Vector(a,b)
val xs: Vector[X] = Vector(A@5faeada1, B@cb51256)

scala> xs.map(_.x)
val res0: Vector[Int] = Vector(42, 43)

scala> class Y { val y: Int }
-- Error:
  class Y needs to be abstract, since val y: Int in class Y is not defined
```

En trait kan ha parametrar

```
trait X(val x: Int)
class A extends X(42)
class B(y: Int) extends X(y) // värdet av y blir argument till x i X
```

En trait kan ha parametrar

```scala
trait X(val x: Int)
class A extends X(42)
class B(y: Int) extends X(y) // värdet av y blir argument till x i X
```

```
scala> val a = A()
val a: A = A@5faeada1

scala> val b = B(43)
val b: B = B@cb51256

scala> val xs = Vector(a,b)
val xs: Vector[X] = Vector(A@5faeada1, B@cb51256)

scala> xs.map(_.x)
val res0: Vector[Int] = Vector(42, 43)

scala> b.y
--  Error:
  value y cannot be accessed as a member of (b : B)
```

Hur kan vi göra medlemmen y synlig?

En trait kan ha parametrar

```scala
trait X(val x: Int)
class A extends X(42)
class B(y: Int) extends X(y) // värdet av y blir argument till x i X
```

```
scala> val a = A()
val a: A = A@5faeada1

scala> val b = B(43)
val b: B = B@cb51256

scala> val xs = Vector(a,b)
val xs: Vector[X] = Vector(A@5faeada1, B@cb51256)

scala> xs.map(_.x)
val res0: Vector[Int] = Vector(42, 43)

scala> b.y
--  Error:
  value y cannot be accessed as a member of (b : B)
```

Hur kan vi göra medlemmen y synlig?

Lägg till **`val`** framför parameternamnet, eller använd en **`case`**-klass.

Abstrakta och konkreta medlemmar

```
object exempelVego1:

  trait Grönsak:
    var vikt: Double              // abstrakt medlem, saknar implementation
    var ärSkalad: Boolean = false // konkret medlem, har implementation

    def skala(): Unit             // abstrakt medlem, saknar implementation

  class Gurka(var vikt: Double) extends Grönsak:
    def skala(): Unit =           // implementation specifik för Gurka
      if !ärSkalad then
        println("Skalas med skalare.")
        vikt = 0.99 * vikt
        ärSkalad = true

  class Tomat(var vikt: Double) extends Grönsak:
    def skala(): Unit =           // implementation specifik för Tomat
      if !ärSkalad then
        println("Skållas.")
        vikt = 0.99 * vikt
        ärSkalad = true
```

Undvika kodduplicering med hjälp av arv

```
object exempelVego2:

  trait Grönsak:  // innehåller gemensamma delar; hjälper oss undvika upprepning
    val skalningsmetod: String      // abstrakt
    val skalfaktor = 0.99           // konkret
    var vikt: Double                // abstrakt
    var ärSkalad: Boolean = false   // konkret

    def skala(): Unit = if !ärSkalad then // konkret
      println(skalningsmetod)
      vikt = skalfaktor * vikt
      ärSkalad = true

  class Gurka(var vikt: Double) extends Grönsak: // det som är speciellt för gurkor
    val skalningsmetod = "Skalas med skalare."

  class Tomat(var vikt: Double) extends Grönsak: // det som är speciellt för tomater
    val skalningsmetod = "Skållas."
```

Varför kan kodduplicering orsaka problem?

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg
- Fler kodrader att underhålla:
  - Om man rättar en bug på ett ställe måste man komma ihåg att göra **exakt samma ändring** på alla de ställen där kodduplicering förekommer → **risk för nya buggar**

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg
- Fler kodrader att underhålla:
  - Om man rättar en bug på ett ställe måste man komma ihåg att göra **exakt samma ändring** på alla de ställen där kodduplicering förekommer → **risk för nya buggar**
- Principen på engelska: `def DRY = "Don't Repeat Yourself!"`

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg
- Fler kodrader att underhålla:
  - Om man rättar en bug på ett ställe måste man komma ihåg att göra **exakt samma ändring** på alla de ställen där kodduplicering förekommer → **risk för nya buggar**
- Principen på engelska: `def DRY = "Don't Repeat Yourself!"`

Det *finns* tillfällen när **kodduplicering faktiskt är att föredra**:

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg
- Fler kodrader att underhålla:
  - Om man rättar en bug på ett ställe måste man komma ihåg att göra **exakt samma ändring** på alla de ställen där kodduplicering förekommer → **risk för nya buggar**
- Principen på engelska: `def DRY = "Don't Repeat Yourself!"`

Det *finns* tillfällen när **kodduplicering faktiskt är att föredra**: t.ex. om man vill att olika delar av koden ska vara **helt oberoende** av varandra.

Subtypspolymorfism och dynamisk bindning

Subtypspolymorfism och dynamisk bindning

```
trait Robot { def work(): Unit }

case class CleaningBot(name: String) extends Robot:
  def work(): Unit = println(" Städa Städa")

case class TalkingBot(name: String) extends Robot:
  def work(): Unit = println(" Prata Prata")
```

Polymorfism betyder ”många former”. Referenserna r och bot nedan kan ha olika ”former”, d.v.s de kan referera till olika sorters robotar.
Dynamisk bindning innebär att körtidstypen avgör vilken metod som körs.

```
scala> def robotDoWork(bot: Robot) = { print(bot); bot.work() }

scala> var r: Robot = CleaningBot("Wall-E")

scala> robotDoWork(r)
CleaningBot(Wall-E) Städa Städa

scala> r = TalkingBot("C3PO")

scala> robotDoWork(r)
TalkingBot(C3PO) Prata Prata
```

Exempel: Överskuggning och `override`

```
object exempelVego3:

  trait Grönsak:
    val skalningsmetod: String      // abstrakt
    val skalfaktor = 0.99           // konkret
    var vikt: Double                // abstrakt
    var ärSkalad: Boolean = false   // konkret

    def skala(): Unit = if !ärSkalad then
      println(skalningsmetod)
      vikt = skalfaktor * vikt
      ärSkalad = true

  class Gurka(var vikt: Double) extends Grönsak:
//nyckelordet override behövs ej om abstrakt medlem i supertyp
    val skalningsmetod = "Skalas med skalare."

  class Tomat(var vikt: Double) extends Grönsak:
//nyckelordet override behövs ej om abstrakt medlem i supertyp men tillåtet:
    override val skalningsmetod = "Skållas."
//  override val skalningmetod = "Skållas." //kompilatorn hittar stavfelet!
    override val skalfaktor = 0.95  // override måste anges vid ändrad impl.
```

En final medlem kan ej överskuggas

```
object exempelVego4:

  trait Grönsak:
    val skalningsmetod: String
    final val skalfaktor = 0.99  // en final medlem kan ej överskuggas
    var vikt: Double
    var ärSkalad: Boolean = false

    def skala(): Unit = if !ärSkalad then
      println(skalningsmetod)
      vikt = skalfaktor * vikt
      ärSkalad = true

  class Gurka(var vikt: Double) extends Grönsak:
    val skalningsmetod = "Skalas med skalare."

  class Tomat(var vikt: Double) extends Grönsak:
    val skalningsmetod = "Skållas."
//  override val skalfaktor = 0.95
//  ger KOMPILERINGSFEL: "cannot override final member"
```

Protected ger synlighet begränsad till subtyper

```
scala> trait Super:
         private val minHemlis = 42
         protected val vårHemlis = 42

scala> class Sub extends Super { def avslöjad = minHemlis }
- Error: Not found: minHemlis

scala> class Sub extends Super { def avslöjad = vårHemlis }

scala> val s = Sub()
val s: Sub = Sub@2eee9593

scala> s.avslöjad
val res0: Int = 42

scala> s.minHemlis
-- Error:
  value minHemlis is not a member of Sub - did you mean s.vårHemlis?

scala> s.vårHemlis
-- Error:
  Access to protected value vårHemlis not permitted because enclosing object
  is not a subclass of trait Super where target is defined
```

Filnamnsregler och -konventioner

- I flera språk, t.ex. Java, gäller dessa regler (men **inte** i Scala):
  - Det går bara ha **en enda publik klass per kodfil**.
  - Kodfilen måste ha **samma namn** som den publika klassen, t.ex. `KlassensNamn.java`
- Scala är **friare**:
  - I Scala får man ha **så många** icke-privata klasser/traits/singelobjekt i samma kodfil **som man vill**.
  - I Scala får man döpa kodfilerna **oberoende** av deras innehåll.

Filnamnsregler och -konventioner

- I flera språk, t.ex. Java, gäller dessa regler (men **inte** i Scala):
  - Det går bara ha **en enda publik klass per kodfil**.
  - Kodfilen måste ha **samma namn** som den publika klassen, t.ex. `KlassensNamn.java`
- Scala är **friare**:
  - I Scala får man ha **så många** icke-privata klasser/traits/singelobjekt i samma kodfil **som man vill**.
  - I Scala får man döpa kodfilerna **oberoende** av deras innehåll.
- Dessa **konventioner** brukar användas i Scala:
  - Om en kodfil bara innehåller **en enda** klass/trait/singelobjekt ge filen samma namn som innehållet, t.ex. `KlassensNamn.scala`
  - Om en kodfil innehåller **flera** saker, döp filen till något som återspeglar hela innehållet och använd **liten begynnelsebokstav**, t.ex. `drawing-utils.scala` eller `bastypensNamn.scala`
- Scala 3 varnar vid arv utanför samma kodfil (se öppna klasser senare).

Klasser, arv och klassparametrar

Klasser kan ärva andra typer (klasser och traits). Om supertypen har parametrar så **måste** subtypen ge argument efter **`extends`**.

```
object personExample1:

  class Person(val namn: String)

  class Akademiker(
    override val namn: String,
    val universitet: String) extends Person(namn)

  class Student(
    override val namn: String,
    override val universitet: String,
    val program: String) extends Akademiker(namn, universitet)

  class Forskare(
    override val namn: String,
    override val universitet: String,
    val titel: String) extends Akademiker(namn, universitet)

  def main(args: Array[String]): Unit =
    val kim = new Student("Kim Robinsson", "Lund", "Data")
    println(s"${kim.namn} ${kim.universitet} ${kim.program}")
```

Statisk och dynamisk typ

```
var p: Person = Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är `Person` vilket gör att vi sedan kan låta p referera till *andra* instanser som är av typen Person.

```
p = Student("Kim Robinson", "Lund", "Data")
```

Statisk och dynamisk typ

```
var p: Person = Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för `p` är `Person` vilket gör att vi sedan kan låta `p` referera till *andra* instanser som är av typen Person.

```
p = Student("Kim Robinson", "Lund", "Data")
```

- Med ”statisk typ” menas den typinformation som finns vid **kompileringstid**.

Statisk och dynamisk typ

```
var p: Person = Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är `Person` vilket gör att vi sedan kan låta p referera till *andra* instanser som är av typen Person.

```
p = Student("Kim Robinson", "Lund", "Data")
```

- Med ”statisk typ” menas den typinformation som finns vid **kompileringstid**.
- Den **dynamiska typen**, även kallad **körtidstypen** som gäller vid exekvering kan vara mer specifik: den dynamiska typen för p är nu efter ovan tilldelning Student, men också Akademiker och Person.

Statisk och dynamisk typ

```
var p: Person = Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för `p` är `Person` vilket gör att vi sedan kan låta `p` referera till *andra* instanser som är av typen Person.

```
p = Student("Kim Robinson", "Lund", "Data")
```

- Med ”statisk typ” menas den typinformation som finns vid **kompileringstid**.
- Den **dynamiska typen**, även kallad **körtidstypen** som gäller vid exekvering kan vara mer specifik: den dynamiska typen för `p` är nu efter ovan tilldelning Student, men också Akademiker och Person.
- Man kan undersöka om den dynamiska typen för `p` är `EnVissTyp` med `p.isInstanceOf[EnVissTyp]` (men använd hellre **`match`**)

Statisk och dynamisk typ

```
var p: Person = Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för `p` är `Person` vilket gör att vi sedan kan låta `p` referera till *andra* instanser som är av typen Person.

```
p = Student("Kim Robinson", "Lund", "Data")
```

- Med ”statisk typ” menas den typinformation som finns vid **kompileringstid**.
- Den **dynamiska typen**, även kallad **körtidstypen** som gäller vid exekvering kan vara mer specifik: den dynamiska typen för `p` är nu efter ovan tilldelning Student, men också Akademiker och Person.
- Man kan undersöka om den dynamiska typen för `p` är `EnVissTyp` med `p.isInstanceOf[EnVissTyp]` (men använd hellre **`match`**)
- Man kan säga åt kompilatorn: **”jag garanterar att `p` är av typen `EnVissTyp` så du kan omforma den statiska typen till `EnVissTyp` och så får jag stå ut med körtidsfel om jag ljuger”** genom att göra **typomvandling** (eng. *type casting*) med `p.asInstanceOf[EnVissTyp]` (detta är mycket ovanligt i normal Scala-kod, eftersom typtest + typomvandling görs säkrare med **`match`**)

Inmixning

Man kan ärva flera traits. Detta kallas **inmixning** (eng. *mix-in*) och görs med en komma-separerad typ-lista efter **`extends`**.

```
object personExample2:

  trait Person     { val namn: String }
  trait Akademiker { val universitet: String }
  trait Examinerad { val titel: String }

  class Student(val namn: String,
                val universitet: String,
                val program: String) extends Person, Akademiker

  class Forskare(val namn: String,
                 val universitet: String,
                 val titel: String) extends Person, Akademiker, Examinerad

  def main(args: Array[String]): Unit =
    val f = new Forskare("B. Regnell", "Lunds universitet", "Professor Dr")
    println(s"${f.titel} ${f.namn}, ${f.universitet}")
```

isInstanceOf och asInstanceOf

Testa körtidstyp med `isInstanceOf[Typ]`. Lova kompilatorn (och ta själv ansvar för) att det är en viss körtidstyp med `asInstanceOf[Typ]`. OBS! Använd hellre **`match`**.

```
object personExample3:

  trait Person     { val namn: String }
  trait Akademiker { val universitet: String }
  trait Examinerad { val titel: String }

  class Student(val namn: String,
                val universitet: String,
                val program: String) extends Person, Akademiker

  class Forskare(val namn: String,
                 val universitet: String,
                 val titel: String) extends Person, Akademiker, Examinerad

  def main(args: Array[String]): Unit =
    var p: Person = new Forskare("B. Regnell", "Lunds universitet", "Professor Dr")
    if p.isInstanceOf[Akademiker] then println(p.namn + " är akademiker")
    p = new Student("Kim Robinson", "Lund", "Data")  // går det att göra p.program?
    if p.isInstanceOf[Student] then println(p.asInstanceOf[Student].program)
  // Ovan görs hellre med match
```

Anonym klass

Om man har en abstrakt typ med saknade implementationer kan man fylla i det som fattas i dessa i ett extra block som ”hängs på” vid instansiering:

```
scala> trait Grönsak { val vikt: Int }
// defined trait Grönsak

scala> new Grönsak    // eller Grönsak()
-- Error:
1 |new Grönsak
  |    ^^^^^^^
  |    Grönsak is a trait; it cannot be instantiated

scala> new Grönsak { val vikt = 42 }
val res0: Grönsak = anon1@4e3f2908
```

Man får då vad som kallas en **anonym klass**. (I detta fall en ganska konstig grönsak som inte är någon speciell sorts grönsak, men som ändå har en vikt.)

Den allra enklaste (och mest meningslösa) anonyma klassen är:

```
scala> new {}
val res0: Object = anon1@5bb37371
```

Hur förhindra subtypning?

Du kan förhindra subtypning på dessa sätt:

- Med **final** skapar du en final klass som ej går att ärva:

```
final class Person(name: String)
```

```
scala> object Björn extends Person("Björn")
-- Error:
1 |object Björn extends Person("Björn")
  |       ^
  |       object Björn cannot extend final class Person
```

- Med **sealed** kan du försegla en hel typhierarki:

```
scala> sealed trait T  // tryck Alt+TAB för att vänta med evaluering
     | final class A extends T
     | final class B extends T
     |
scala> new T{}
-- Error:
1 |new T{}
|^
|Cannot extend sealed trait T in a different source file
```

Med en försegla typhierarki kan du bara ärva bastypen inom samma

Förseglade typer med `sealed`

Med en **`sealed`** kan du skapa en **förseglad** uppräkning:

```
sealed trait Färg(val toInt: Int)
object Färg:
  val values = Vector(Spader, Hjärter, Ruter, Klöver)
  case object Spader  extends Färg(0)
  case object Hjärter extends Färg(1)
  case object Ruter   extends Färg(2)
  case object Klöver  extends Färg(3)
```

Nyckelordet **`sealed`** **förhindrar** vidare subtypning av `Färg` i **annan kodfil** och **ger varning** om matchning är ofullständig – tips för att undvika körtidsfel.

```
scala> Färg.values(0) match { case Färg.Spader => "hej" }
-- Warning:
1 |Färg.values(0) match { case Färg.Spader => "hej" }
  |^^^^^^^^^^^^^^
  |match may not be exhaustive.
  |
  |It would fail on pattern case: Hjärter, Ruter, Klöver
val res0: String = hej
```

Använd hellre **`enum`** så får du både **`sealed`** och mer godis på köpet!

Öppen klass signalerar uppmuntrad subtypning

- Om man ärver en klass får man tillgång till alla medlemmar som inte är privata och kan byta till godtycklig implementation om typerna stämmer.
- Detta kan vara riskabelt om den som skrivit klassen inte planerat för detta och noga dokumenterat hur klassen är tänkt att användas vid arv.
- Genom att skapa **öppna klasser** med nyckelordet **open** signalerar du att klassen är tänkt att vara en supertyp vid arv.

```
open class Gurka(val vikt: Int, val pris: Double):
  /** Gör override på denna om du vill ha annat alternativ. */
  def alternativ: String = s"Det går precis lika bra med selleri!"
```

```
scala> object StorGurkan extends Gurka(1000, 1_000_000): // ärv på bäst du vill!
         override def alternativ = "kan kanske också funka med en betongpelare"
```

- **open** krävs om du vill tysta varning vid **arv från en annan kodfil**.
- Du kan även komma undan varningen med `import scala.language.adhocExtensions`

https://youtu.be/aFmIS5qeetA?t=206

Trait eller abstrakt klass?

Nyckelordet **`abstract`** behövs framför **`class`** om abstrakta medlemmar:

```
scala> class X { val x: Int }
1 |class X { val x: Int }
  |      ^
  |      class X needs to be abstract, since val x: Int in class X is not defined

scala> abstract class X { val x: Int }  // fungerar!
```

Men går det inte lika bra med en trait?

Trait eller abstrakt klass?

Nyckelordet **`abstract`** behövs framför **`class`** om abstrakta medlemmar:

```
scala> class X { val x: Int }
1 |class X { val x: Int }
  |      ^
  |      class X needs to be abstract, since val x: Int in class X is not defined

scala> abstract class X { val x: Int }  // fungerar!
```

Men går det inte lika bra med en trait? Det går ofta precis lika bra med en trait.

Använd en **trait** om...

- ...du är osäker på vilket som är bäst. (Du kan alltid ändra till en abstrakt klass senare.)
- ...du vill kunna mixa in din trait tillsammans med andra traits.
- ...du vill göra din trait, om inmixad, transparent vid typhärledning.

Använd en (abstrakt) **klass** om...

- ...du vill begränsa inmixning – man kan bara ärva från en enda klass.
- ...du vill vidarebefordra parameter till supertyp (se nästa bild).
- ...du vill ärva din klass i Java-kod.
- ...du vill minimera tid för omkompilering vid ändringar (spar tid vid stora projekt).

En trait får ej vidarebefordra parametrar

En trait får inte skicka vidare parametrar till en supertyp (det skulle bli knepiga problem annars vid inmixning):

```
scala> trait X(x: Int)
// defined trait X

scala> trait Y(y: Int) extends X(y)
-- Error:
1 |trait Y(y: Int) extends X(y)
  |                         ^^^^
  |  trait Y may not call constructor of trait X
```

Men det funkar fint med en **`abstract class`** eller en **`class`** (som ju **inte** får vara inmixningar):

```
scala> abstract class Y(y: Int) extends X(y)
// defined class Y
```

Mer detaljer här: dotty.epfl.ch/docs/reference/other-new-features/trait-parameters.html

Överskuggningsregler

Medlemmar, arv och överskuggning

Olika sorters överskuggningsbara medlemmar i **Scala**:

- `def`
- `val`
- `lazy val`
- `var`

Medlemmar, arv och överskuggning

Olika sorters överskuggningsbara medlemmar i **Scala**:

- `def`
- `val`
- `lazy val`
- `var`

Olika sorters överskuggningsbara instansmedlemmar i **Java**:

- variabel
- metod

Medlemmar som är `static` kan ej överskuggas (men döljas) vid arv.

Medlemmar, arv och överskuggning

Olika sorters överskuggningsbara medlemmar i **Scala**:

- `def`
- `val`
- `lazy val`
- `var`

Olika sorters överskuggningsbara instansmedlemmar i **Java**:

- variabel
- metod

Medlemmar som är `static` kan ej överskuggas (men döljas) vid arv.

- När man överskuggar (eng. *override*) en medlemmen med en annan medlem med samma namn i en subtyp, får denna medlem en (ny) implementation.
- Singelobjekt kan ej ärvas (och medlemmar i singelobjekt kan därmed ej överskuggas).
- När man konstruerar ett objektorienterat språk gäller det att man definierar sunda överskuggningsregler vid arv. Detta är förvånansvärt knepigt.
- Alla knepiga regler för överskuggning är svåra att lära sig utantill, men som tur är så hjälper kompilatorns felmeddelande dig att följa reglerna `:`)

Fördjupning: Regler för överskuggning i Scala

En medlem M1 i en supertyp får överskuggas av en medlem M2 i en subtyp, enligt dessa regler: (se även `https://stackoverflow.com/questions/40057337`)

1. M1 och M2 ska ha samma namn och typerna ska matcha.
2. **`def`** får bytas ut mot: **`def`**, **`val`**, och **`lazy val`**, och under speciella förutsättningar **`var`** (mer om att byta **`def`** mot **`var`** snart)
3. **`val`** får bytas ut mot **`val`**. Om medlemmen i M1 som överskuggas är abstrakt så får en **`val`** även bytas mot en **`lazy val`**.
4. En abstrakt **`var`** får bara bytas ut mot en **`var`**. En konkret **`var`** får ej bytas ut.
5. **`lazy val`** får bara bytas ut mot en **`lazy val`**.
6. Om en medlem i en supertyp är abstrakt *behöver* man inte använda nyckelordet **`override`** i subtypen. (Men det är bra att göra det ändå så att kompilatorn hjälper dig att kolla att du verkligen överskuggar något.)
7. Om en medlem i en supertyp är konkret *måste* man använda nyckelordet **`override`** i subtypen, annars ges kompileringsfel.
8. M1 får inte vara **`final`**.
9. M1 får inte vara **`private`**, men kan vara **`private`**`[X]` om M2 också är **`private`**`[X]`, eller **`private`**`[Y]` om X är (en del av) Y.
10. Om M1 är **`protected`** måste även M2 vara det.

Fördjupning: Överskugga `var` med `var`

Det krävs speciella förutsättningar för att överskugga en **`var`** med en **`var`**.

- En konkret medlem får **inte** bytas ut mot en **`var`** – inte ens en konkret **`var`**:

```
scala> trait ConcreteVar { var x = 42 }
scala> trait Sub extends ConcreteVar { override var x = 43 }
-- Error:
1 |trait Sub extends ConcreteVar { override var x = 43 }
  |                                             ^
  |             error overriding variable x in trait ConcreteVar of type Int;
  |               variable x of type Int cannot override a mutable variable
```

- En abstrakt **`var`**-medlem får bytas ut mot en **`var`** om du **inte** skriver **`override`**:

```
scala> trait AbstractVar { var x: Int }
scala> trait Sub extends AbstractVar { override var x = 43 }
-- Error:
1 |trait Sub extends AbstractVar { override var x = 43 }
  |                                             ^
  |             error overriding variable x in trait AbstractVar of type Int;
  |               variable x of type Int cannot override a mutable variable
scala> trait Sub extends AbstractVar { var x = 43 }   // funkar utan override
// defined trait Sub
```

Undvik abstrakta **`var`**-medlemmar – oftast bättre med abstrakt **`def`**.

Fördjupning: Överskugga `def` med `var`

- En abstrakt **`def`**-medlem får bytas ut mot en **`var`** om du **inte** skriver **`override`**:

```
scala> trait Super { def x: Int }
scala> trait Sub extends Super { override var x = 43 }
-- Error:
1 |trait Sub extends Super { override var x = 43 }
  |                                           ^
  |                                           setter x_= overrides nothing

scala> trait Sub extends Super { var x = 43 }   // funkar om ej override
```

  Den abstrakta **`def`**-medlemmen blir då implementerad av en konkret getter.
- Egentligen är en publik **`var`**-medlem en kombination av en getter och en setter. Du kan skapa konkret getter+setter och överskugga gettern explicit med **`override`** (notera att settern inte kan göra **`override`**, eftersom superklassen inte har någon motsvarande metod att byta ut – jämför felmeddelande ovan):

```
scala> trait Sub2 extends Super:
         private var myPrivateValue = 42
         override def x: Int = myPrivateValue
         def x_(newValue: Int): Unit = myPrivateValue = newValue
```

  Ovan fungerar fint eftersom vi nu har en giltig kombination av getter+setter.

Att skilja på mitt och ditt med super

```
scala> class X { def gurka = "super pepino" }

scala> class Y extends X:
         override val gurka = ":("
         val sg = super.gurka

scala> val y = new Y
y: Y = Y@26ba2a48

scala> y.gurka
res0: String = :(
```

Att skilja på mitt och ditt med super

```
scala> class X { def gurka = "super pepino" }

scala> class Y extends X:
         override val gurka = ":("
         val sg = super.gurka

scala> val y = new Y
y: Y = Y@26ba2a48

scala> y.gurka
res0: String = :(
```

Super Pepinos to the rescue:

```
scala> y.sg
res1: String = super pepino
```

Att skilja på mitt och ditt med `super`

```
scala> class X { def gurka = "super pepino" }

scala> class Y extends X:
         override val gurka = ":("
         val sg = super.gurka

scala> val y = new Y
y: Y = Y@26ba2a48

scala> y.gurka
res0: String = :(
```

Super Pepinos to the rescue:

```
scala> y.sg
res1: String = super pepino
```

https://youtu.be/NPhjiXskz34

Fördjupning: Intersektionstyp

Vid inmixning blir supertypen en intersektionstyp (eng. *intersection type*), vilket indikeras med typoperatorn & i exempel nedan:

```
trait Grönsak(val vikt: Int)
trait HarSmak(val smak: String)

object Gurka extends Grönsak(42), HarSmak("vattning")
object Tomat extends Grönsak(43), HarSmak("syrlig")
```

```
scala> Vector(Gurka, Tomat)
val res0: Vector[Grönsak & HarSmak] =
  Vector(Gurka@560742f7, Tomat@3cc82e23)
```

Det går att skapa egna intersektionstyper med **`with`**:

```
scala> class X { val x = 42 }
scala> trait Y { val y = "hej" }
scala> val a: X & Y = new X         // -- Error: Found: X Required: X & Y
scala> val b: X & Y = new X with Y  // Funkar! En anonym klass av typen X & Y
```

Fördjupning: Transparent trait

Du kan göra din trait **genomskinlig** vid typhärledning av supertypen för inmixningen med nyckelordet **`transparent`**:

```
trait Grönsak(val vikt: Int)
transparent trait HarSmak(val smak: String)

object Gurka extends Grönsak(42), HarSmak("vattning")
object Tomat extends Grönsak(43), HarSmak("syrlig")
```

```
scala> Vector(Gurka, Tomat)
val res0: Vector[Grönsak] =
  Vector(Gurka@560742f7, Tomat@3cc82e23)
```

Vilken typ hade visats utan **`transparent`**?

Fördjupning: Transparent trait

Du kan göra din trait **genomskinlig** vid typhärledning av supertypen för inmixningen med nyckelordet **`transparent`**:

```
trait Grönsak(val vikt: Int)
transparent trait HarSmak(val smak: String)

object Gurka extends Grönsak(42), HarSmak("vattning")
object Tomat extends Grönsak(43), HarSmak("syrlig")
```

```
scala> Vector(Gurka, Tomat)
val res0: Vector[Grönsak] =
  Vector(Gurka@560742f7, Tomat@3cc82e23)
```

Vilken typ hade visats utan **`transparent`**?
`Vector[Grönsak & HarSmak]`

Fördjupning: Typunioner med eller-operator

Typunioner skapas typ-operatorn | mellan typer:

```
scala> var x: Int | String = 42
var x: Int | String = 42

scala> x = "hej"
x: Int | String = hej

scala> type IntOrErr = Int | String
// defined alias type IntOrErr = Int | String

scala> def div(nom: Int, denom: Int): IntOrErr =
          if denom != 0 then nom / denom else "div. by zero"
```

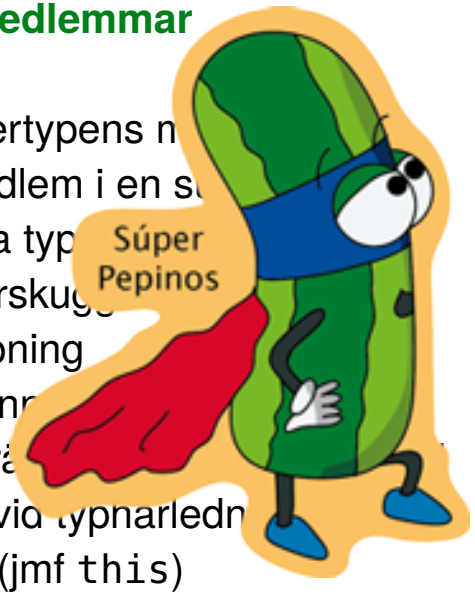
Fördel jämfört med klass: rudimentärt enkelt.
Nackdelar: kan inte deklarera parametrar, medlemmar och allt annat som en klass kan; i viss mån kan detta kompensera med **`extension`** och **`match`**.
Läs mer här:
`https://dotty.epfl.ch/docs/reference/new-types/union-types.html`

Terminologi och nyckelord vid arv

subtyp	en typ som ärver en supertyp
supertyp	en typ som ärvs av en subtyp
bastyp	en typ som är rot i ett arvsträd
abstrakt medlem	en medlem som saknar implementation
konkret medlem	en medlem som ej saknar implementation
abstrakt typ	en typ som kan ha abstrakta medlemmar; kan ej instansieras
konkret typ	en typ som ej har abstrakta medlemmar; kan instansieras
anonym klass	en namnlös klass som skapas direkt vid instansiering
`class`	en konkret typ som **kan ej ha abstrakta medlemmar**
`abstract class`	en abstrakt typ som **kan ha abstrakta medlemmar**
`trait`	är en abstrakt typ som **kan mixas in**
`extends`	står före en supertyp, medför arv av supertypens medlemmar
`override`	en medlem överskuggar (byter ut) en medlem i en superttyp
`protected`	gör en medlem synlig i subtyper till denna typ (jmf `private`)
`final def gurka`	gör medlemen gurka final: förhindrar överskuggning
`final class`	gör klassen final: förhindrar vidare subtypning
`sealed`	förseglad trait/klass: enbart subtyper i denna kodfil, koll av match
`open class`	berätta att den är tänkt att ärvas, `open` krävs för arv i annan kodfil
`transparent trait`	gör typen ”genomskinlig” (alltså osynlig) vid typhärledning
`super.gurka`	refererar till supertypens medlem `gurka` (jmf `this`)

Terminologi och nyckelord vid arv

subtyp	en typ som ärver en supertyp
supertyp	en typ som ärvs av en subtyp
bastyp	en typ som är rot i ett arvsträd
abstrakt medlem	en medlem som saknar implementation
konkret medlem	en medlem som ej saknar implementation
abstrakt typ	en typ som kan ha abstrakta medlemmar; kan ej instansieras
konkret typ	en typ som ej har abstrakta medlemmar; kan instansieras
anonym klass	en namnlös klass som skapas direkt vid instansiering
`class`	en konkret typ som **kan ej ha abstrakta medlemmar**
`abstract class`	en abstrakt typ som **kan ha abstrakta medlemmar**
`trait`	är en abstrakt typ som **kan mixas in**
`extends`	står före en supertyp, medför arv av supertypens m
`override`	en medlem överskuggar (byter ut) en medlem i en s
`protected`	gör en medlem synlig i subtyper till denna typ
`final def` gurka	gör medlemen gurka final: förhindrar översku
`final class`	gör klassen final: förhindrar vidare subtypning
`sealed`	förseglad trait/klass: enbart subtyper i denn
`open class`	berätta att den är tänkt att ärvas, `open` krä
`transparent trait`	gör typen ”genomskinlig” (alltså osynlig) vid typharledn
`super`.gurka	refererar till supertypens medlem `gurka` (jmf `this`)



Algebraiska datatyper

Vad är en algebraisk datatyp?

Algebraisk datatyp (eng. *algebraic data type*), förk. ADT:

- En datatyp som består av (en kombination av):
- **Produkt**-typer, **Summa**-typer:
  - En produkt-typ är en ”och”-typ (ä.k. record, struct), exempel:
    `case class Person(namn: String, ålder: Int)`
    består av attributen namn **OCH** ålder.
  - En summa-typ är en ’Eller”-typ (ä.k. disjunkt union, co-produkt).
  - Exempel: `enum Färg { case Röd, Svart}`
    Färg kan vara antingen Röd **ELLER** Svart.
    En `Grönsak` är antingen en `Gurka` **ELLER** en `Tomat`

```
sealed trait Grönsak { val vikt: Int }
case class Gurka(vikt: Int) extends Grönsak
case class Tomat(vikt: Int) extends Grönsak
```

https://en.wikipedia.org/wiki/Algebraic_data_type

En case-klass är en produkt.

Klassen `Person` nedan har ett namn **och** en ålder.

```
scala> case class Person(namn: String, ålder: Int)

scala> Person("Kim",42)
val res0: Person = Person(Kim,42)

scala> res0.isInstanceOf[Product]
val res1: Boolean = true

scala> res0.product   // Tryck TAB
productArity          productElement    productElementName
productElementNames   productIterator   productPrefix

scala> res0.productElementNames.toVector
val res2: Vector[String] = Vector(namn, ålder)

scala> res0.productElement
val res3: Int => Any = Lambda1981/0x00000008408f2840@44498af0

scala> res0.productElement(0)
val res4: Any = björn
```

Algebraisk datatyp, kombinerad produkt och summa

Med trait, case-klass och objekt:

```
sealed trait PersonId // summan av två subtyper (Number eller Missing)
object PersonId:
  case class Number(n: Long) extends PersonId  // produkt med ett element
  case object Missing extends PersonId         // ett enkelt värde
```

Algebraisk datatyp, kombinerad produkt och summa

Med trait, case-klass och objekt:

```
sealed trait PersonId // summan av två subtyper (Number eller Missing)
object PersonId:
  case class Number(n: Long) extends PersonId  // produkt med ett element
  case object Missing extends PersonId         // ett enkelt värde
```

Motsvarande med **enum**:

```
enum PersonId:
  case Number(n: Long)
  case Missing
```

Algebraisk datatyp med typparameter

Denna ADT har ett fall som är en **generisk** case-klass:

```
sealed trait Kanske[+A]   // +A gör typen flexibel, ä.k. "kovariant" mer om det senare
object Kanske:
  case class Någon[A](a: A) extends Kanske[A]
  case object Ingen extends Kanske[Nothing]
```

Algebraisk datatyp med typparameter

Denna ADT har ett fall som är en **generisk** case-klass:

```
sealed trait Kanske[+A]   // +A gör typen flexibel, ä.k. "kovariant" mer om det senare
object Kanske:
  case class Någon[A](a: A) extends Kanske[A]
  case object Ingen extends Kanske[Nothing]
```

Motsvarande med **`enum`**: (kompilatorn fyller i **`extends`** ... automatiskt)

```
enum Kanske[+A]:
  case Någon(a: A)
  case Ingen
```

Känner du igen denna?

Algebraisk datatyp med typparameter

Denna ADT har ett fall som är en **generisk** case-klass:

```
sealed trait Kanske[+A]   // +A gör typen flexibel, ä.k. "kovariant" mer om det senare
object Kanske:
  case class Någon[A](a: A) extends Kanske[A]
  case object Ingen extends Kanske[Nothing]
```

Motsvarande med **`enum`**: (kompilatorn fyller i **`extends`** `...` automatiskt)

```
enum Kanske[+A]:
  case Någon(a: A)
  case Ingen
```

Känner du igen denna? Jämför inbyggda typen `Option`

Implementation av `getOrElse` med extensionsmetod, tips: använd **`match`**

Algebraisk datatyp med typparameter

Denna ADT har ett fall som är en **generisk** case-klass:

```
sealed trait Kanske[+A]   // +A gör typen flexibel, ä.k. "kovariant" mer om det senare
object Kanske:
  case class Någon[A](a: A) extends Kanske[A]
  case object Ingen extends Kanske[Nothing]
```

Motsvarande med **`enum`**: (kompilatorn fyller i **`extends`** `...` automatiskt)

```
enum Kanske[+A]:
  case Någon(a: A)
  case Ingen
```

Känner du igen denna? Jämför inbyggda typen `Option`

Implementation av `getOrElse` med extensionsmetod, tips: använd **`match`**

```
extension [A](k: Kanske[A]) def getOrElse(default: A):A = k match
  case Kanske.Någon(a) => a
  case Kanske.Ingen => default
```

11 Varians och kontextparametrar

- Repetition: typparametrar
- Generiska typgränser
- Varians: flexibla typparametrar
- Vad är ett bra api?
- Kontextuella abstraktioner
- Översikt av kursens avslutning

Repetition: typparametrar

Typparameter, generisk struktur, typkonstruktor

- Med hjälp av **typparametrar** (eng. *type parameters*) kan du skapa **generiska strukturer** (eng. *generic structures*).
- Typparametrar skrivs inom hakparenteser, exempelvis: `[T, U]`
- En generisk struktur fungerar för *godtycklig* typ, **okänd** vid **deklaration**.
- Kompilatorn säkerställer **korrekt användning** redan vid *kompilering*.
- På användningsplatsen i koden (vid anrop eller instansiering) **binds** typparametrar till ”verkliga” typer enligt typsystemets regler.

```
case class Pair[A, B](a: A, b: B):  // A och B är obundna (fria) inom []
  def swap: Pair[B, A] = Pair(b, a) // A och B bundna då swap saknar []
```

- Skriver du inte ut typerna försöker kompilatorn **härleda** (eng. *infer*) dem:

```
scala> val p = Pair("hej", 42)
val p: Pair[String, Int] = Pair(hej,42)

scala> p.swap
val res0: Pair[Int, String] = Pair(42,hej)
```

- Klassen `Pair` kallas **typkonstruktor** (eng. *type constructor*) då den ”färdiga” typen `Pair[String, Int]` ”konstrueras” vid användning.

Typparameter, generisk struktur, typkonstruktor

- Med hjälp av **typparametrar** (eng. *type parameters*) kan du skapa **generiska strukturer** (eng. *generic structures*).
- Typparametrar skrivs inom hakparenteser, exempelvis: `[T, U]`
- En generisk struktur fungerar för *godtycklig* typ, **okänd** vid **deklaration**.
- Kompilatorn säkerställer **korrekt användning** redan vid *kompilering*.
- På användningsplatsen i koden (vid anrop eller instansiering) **binds** typparametrar till ”verkliga” typer enligt typsystemets regler.

```
case class Pair[A, B](a: A, b: B):  // A och B är obundna (fria) inom []
  def swap: Pair[B, A] = Pair(b, a) // A och B bundna då swap saknar []
```

- Skriver du inte ut typerna försöker kompilatorn **härleda** (eng. *infer*) dem:

```
scala> val p = Pair("hej", 42)
val p: Pair[String, Int] = Pair(hej,42)

scala> p.swap
val res0: Pair[Int, String] = Pair(42,hej)
```

- Klassen `Pair` kallas **typkonstruktor** (eng. *type constructor*) då den ”färdiga” typen `Pair[String, Int]` ”konstrueras” vid användning.

Övning: Ändra klassen `Pair` ovan så att båda elementen i paret har samma typ.

Generiska typgränser

Olika sätt att begränsa generiska typer

Det finns i Scala flera olika sätt att att begränsa vilka typer du vill tillåta – kompilatorn att hjälper dig att kontrollera detta.

- **Övre gräns** (eng. *upper bound*) med `A <: B`
- **Undre gräns** (eng. *lower bound*) med `A >: B`
- **Egentyp** (eng. *self type*) begränsar hur du kan mixa in en trait.

Olika sätt att begränsa generiska typer

Det finns i Scala flera olika sätt att att begränsa vilka typer du vill tillåta – kompilatorn att hjälper dig att kontrollera detta.

- **Övre gräns** (eng. *upper bound*) med `A <: B`
- **Undre gräns** (eng. *lower bound*) med `A >: B`
- **Egentyp** (eng. *self type*) begränsar hur du kan mixa in en trait. Ge ett godtyckligt namn följt av en typanotering och en raket i klasskroppen:

```
trait MinTrait:
  self: ÄrDennaTyp =>

  // Kod här kan utgå från att denna instans är av typen ÄrDennaTyp
  // namnet, här self, är valfritt, men self är vanligaste valet
```

  Kompilatorn kontrollerar att inmixing sker med `ÄrDennaTyp`.
- Det finns också något som heter **kontextgräns** (eng. *context bound*) där `[A: B]` gör så att typkonstruktorn `B` blir en kontextparameter `B[A]` – mer om det senare i avsnittet om kontextuella abstraktioner.

Övre och undre typgräns

Med typoperatorerna `<:` och `>:` går det att begränsa vilka typer som kan bindas till en typparameter i en generiska struktur.

- Minnesregel för typgränser: **kolon på slutet**.

Antag att T är en **obunden** typparameter, medan U och L är **bundna**.

- med T `<:` U blir U en övre gräns (eng. *upper bound*) för T
  U är ”högsta” möjliga typ som T får vara (jämför ”mindre eller lika med”)
- med T `>:` L blir L en undre gräns (eng. *lower bound*) för T
  U är ”lägsta” möjliga typ som T får vara (jämför ”större eller lika med”)
- För alla typer `A` gäller att `A >: Nothing` och `A <: Any`

Övre och undre typgräns

Med typoperatorerna `<:` och `>:` går det att begränsa vilka typer som kan bindas till en typparameter i en generiska struktur.

- Minnesregel för typgränser: **kolon på slutet**.

Antag att T är en **obunden** typparameter, medan U och L är **bundna**.

- med T `<:` U blir U en övre gräns (eng. *upper bound*) för T
  U är ”högsta” möjliga typ som T får vara (jämför ”mindre eller lika med”)
- med T `>:` L blir L en undre gräns (eng. *lower bound*) för T
  U är ”lägsta” möjliga typ som T får vara (jämför ”större eller lika med”)
- För alla typer A gäller att `A >: Nothing` och `A <: Any`

Exempel:

```
trait Grönsak { def vikt: Int }

def f[T <: Grönsak](x: T): Int = x.vikt
```

Kompilatorn använder den övre typgränsen för att konstatera att metoden `vikt` är tillgänglig via den generiska parametern x.

Exempel på övre och undre typgräns

```scala
class Djur
class Katt extends Djur
class Hund extends Djur
class Robothund extends Hund

def testUpperBound[T <: Hund](x: T) = println(x)
def testLowerBound[T >: Hund](x: T) = println(x)
```

```
scala> testUpperBound[Katt](Katt())
-- Error:
1 |testUpperBound[Katt](Katt())
  |          ^
  |          Type argument Katt does not conform to upper bound Hund

scala> testLowerBound[Robothund](Robothund())
-- Error:
1 |testLowerBound[Robothund](Robothund())
  |          ^
  |          Type argument Robothund does not conform to lower bound Hund
```

Varians: flexibla typparametrar

Vad är varians?

Vad är varians?

Är en kattbur också en djurbur??

Om vi tillåter **varians** så blir generiska strukturer mer **flexibla**.

Varför behövs varians?

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Varför behövs varians?

```scala
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Nedan fungerar inte! Buren ovan är **invariant** (oflexibel i sin typparameter).

```
scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
-- Error:
1 |val djurbur: Bur[Djur] = Bur[Katt](Katt())
  |                          ^^^^^^^^^^^^^^^^^
  |                          Found:    Bur[Katt]
  |                          Required: Bur[Djur]
```

Varför behövs varians?

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Nedan fungerar inte! Buren ovan är **invariant** (oflexibel i sin typparameter).

```
scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
-- Error:
1 |val djurbur: Bur[Djur] = Bur[Katt](Katt())
  |                          ^^^^^^^^^^^^^^^^^
  |                          Found:    Bur[Katt]
  |                          Required: Bur[Djur]
```

Varför fungerar detta??

```
scala> val djur: Vector[Djur] = Vector[Katt](Katt())
val djur: Vector[Djur] = Vector(Katt())
```

Varför behövs varians?

```scala
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[A](a: A)
```

Nedan fungerar inte! Buren ovan är **invariant** (oflexibel i sin typparameter).

```
scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
-- Error:
1 |val djurbur: Bur[Djur] = Bur[Katt](Katt())
  |                          ^^^^^^^^^^^^^^^^^
  |                          Found:    Bur[Katt]
  |                          Required: Bur[Djur]
```

Varför fungerar detta??

```
scala> val djur: Vector[Djur] = Vector[Katt](Katt())
val djur: Vector[Djur] = Vector(Katt())
```

`Vector` är deklarerad som **kovariant** och därmed mer flexibel!

Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[T] <: F[U]` (subtypsflexibel ”på samma håll”)
- I Scala kan du få en kovariant typkonstruktor med + före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A)  // kovariant tack vare + före A
```

Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[T] <: F[U]` (subtypsflexibel ”på samma håll”)
- I Scala kan du få en kovariant typkonstruktor med + före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A)  // kovariant tack vare + före A
```

- Nu funkar det :)

```
scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
val djurbur: Bur[Djur] = Bur(Katt())
```

- `Bur[Katt]` är nu en **subtyp** till `Bur[Djur]`.

Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[T] <: F[U]` (subtypsflexibel ”på samma håll”)
- I Scala kan du få en kovariant typkonstruktor med + före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A)  // kovariant tack vare + före A
```

- Nu funkar det :)

```
scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
val djurbur: Bur[Djur] = Bur(Katt())
```

- `Bur[Katt]` är nu en **subtyp** till `Bur[Djur]`.
- **Oföränderliga** samlingar är ofta kovarianta, t.ex Vector, Option, List.

Kovarians (eng. *covariance*)

- För en **kovariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[T] <: F[U]` (subtypsflexibel ”på samma håll”)
- I Scala kan du få en kovariant typkonstruktor med + före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

case class Bur[+A](a: A)  // kovariant tack vare + före A
```

- Nu funkar det :)

```
scala> val djurbur: Bur[Djur] = Bur[Katt](Katt())
val djurbur: Bur[Djur] = Bur(Katt())
```

- `Bur[Katt]` är nu en **subtyp** till `Bur[Djur]`.
- **Oföränderliga** samlingar är ofta kovarianta, t.ex Vector, Option, List.
- Generiska enumerationer **behöver** vara kovarianta för att de olika fallen ska få en flexibel typparameter. `Ledig` är en `Toalett[Nothing]` här:
  `enum Toalett[+T] { case Upptagen(x: T); case Ledig }`

Kontravarians

Kontravarians

Är en kattveterinär också en djurveterinär?

Ibland vill vi ha variansen på andra hållet: En veterinär som bara kan behandla katter ska inte få behandla vilket djur som helst.

Kontravarians (eng. *contravariance*)

- För en **kontravariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[U] <: F[T]` (subtypsflexibel ”på fel håll”)
- Du skapar en kontravariant typkonstruktor med `-` före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

class Veterinär[-A]:    // kontravariant med -
  def behandla(x: A) = println(s"$this har behandlat $x")
```

Kontravarians (eng. *contravariance*)

- För en **kontravariant** typkonstruktor F gäller att:
  om `T <: U` **så** `F[U] <: F[T]` (subtypsflexibel ”på fel håll”)
- Du skapar en kontravariant typkonstruktor med `-` före typparametern:

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur

class Veterinär[-A]:    // kontravariant med -
  def behandla(x: A) = println(s"$this har behandlat $x")
```

- Nu funkar det ”baklänges” (men inte på andra hållet):

```
scala> val kattveterinär: Veterinär[Katt] = Veterinär[Djur]()
val kattveterinär: Veterinär[Katt] = Veterinär@77b6d94c

scala> val kattveterinär: Veterinär[Djur] = Veterinär[Katt]()
-- Error:
1 |val kattveterinär: Veterinär[Djur] = Veterinär[Katt]()
  |                                     ^^^^^^^^^^^^^^^^^
  |                                     Found:    Veterinär[Katt]
  |                                     Required: Veterinär[Djur]
```

Variansproblem – tack kompilatorn!

- En typkonstruktor kan **inte** vara kovariant om typparametern används som parametertyp för metoder (så kallad *kontravariant position*):

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur
```

```
scala> case class Bur[+A](a: A):
         def bytTill(x: A): Bur[A] = Bur(x)
-- Error:
2 |  def bytTill(x: A): Bur[A] = Bur(x)
  |          ^^^^
  |  covariant type A occurs in contravariant position in type A of parameter x
```

Variansproblem – tack kompilatorn!

- En typkonstruktor kan **inte** vara kovariant om typparametern används som parametertyp för metoder (så kallad *kontravariant position*):

```
trait Djur
case class Katt() extends Djur
case class Hund() extends Djur
```

```
scala> case class Bur[+A](a: A):
         def bytTill(x: A): Bur[A] = Bur(x)
-- Error:
2 |  def bytTill(x: A): Bur[A] = Bur(x)
  |          ^^^^
  |  covariant type A occurs in contravariant position in type A of parameter x
```

- Här måste typen tillåtas variera ”uppåt” med hjälp av en **undre gräns**:

```
case class Bur[+A](a: A):
  def bytTill[B >: A](x: B): Bur[B] = Bur(x)
```

```
scala> Bur[Katt](Katt()).bytTill(Hund())
val res1: Bur[Djur] = Bur(Hund())
```

När använda vilken slags varians?

- Oföränderliga generiska klasser som har metoder som är **producenter** av nya oföränderliga generiska typer passar ofta bäst som **kovarianta**, t.ex. `Vector[+T]`, `Option[+T]` .
- En typkonstruktor av `T`, som har metoder som är **konsumenter** av `T`, kan vara **kontravariant**, t.ex. `Veterinär[-T]`.
  Den kan också vara **kovariant** om konsumerande metoder **vidgar** inparametertypen till `B` där `B >: T`
- En typkonstruktor med flera parametrar kan vara **både** kontravariant **och** kovariant, t.ex. `Function1[-A, +B]`
  ett parametervärde `:A` *konsumeras* och ett returtvärde `:B` *produceras*
  (`Function1[-A, +B]` är egentliga typen av ”syntaktiska sockret” `A => B`

När använda vilken slags varians?

- Oföränderliga generiska klasser som har metoder som är **producenter** av nya oföränderliga generiska typer passar ofta bäst som **kovarianta**, t.ex. `Vector[+T]`, `Option[+T]` .
- En typkonstruktor av T, som har metoder som är **konsumenter** av T, kan vara **kontravariant**, t.ex. `Veterinär[-T]`.
  Den kan också vara **kovariant** om konsumerande metoder **vidgar** inparametertypen till B där `B >: T`
- En typkonstruktor med flera parametrar kan vara **både** kontravariant **och** kovariant, t.ex. `Function1[-A, +B]`
  ett parametervärde `:A` *konsumeras* och ett returtvärde `:B` *produceras*
  (`Function1[-A, +B]` är egentliga typen av ”syntaktiska sockret” `A => B`
- **Förändringsbara** strukturer måste vara **invarianta**, annars väntar **kaos**!
  Antag att det finns en `Bur` som kan ändras på plats via metoden `bytTill` *och samtidigt* vore kovariant (varning för känsliga kodare):

```
ejscala> val kattBur: Bur[Katt] = Bur(Katt())

ejscala> val djurBur: Bur[Djur] = kattBur  // en kovariant referens till samma Bur

ejscala> djurBur.bytTill(Hund())           // ändra på plats

ejscala> val katt: Katt = kattBur.släppUt  // KAOS! typosäkert hundkatt-monster :(
```

Typjoker: varning för gränslösa typer

Invarianta klasser har oflexibel typparameter, vilket begränsar användningen.

```
scala> class Box[A](val value: A)

scala> val b: Box[Any] = Box[Int](42)
-- Error:
1 |val b: Box[Any] = Box[Int](42)
  |                  ^^^^^^^^^^^^
  |                  Found:    Box[Int]
  |                  Required: Box[Any]
```

Tveksam räddning: En **typjoker** (eng. *wildcard type*) skrivs med ett frågetecken och ger en helt okänd typ som **ej kontrolleras av kompilatorn**.

```
scala> var whatever: Box[?] = Box[Int](42)
var whatever: Box[?] = Box@70d7a49b

scala> whatever = Box("hej")
whatever: Box[?] = Box@43120a77
```

Använd bara typpjoker om det *verkligen* behövs.

Mer om varians för den nyfikne

- https://docs.scala-lang.org/scala3/book/types-variance.html
- https://en.wikipedia.org/wiki/Covariance_and_contravariance_-(computer_science)
- https://www.artima.com/pins1ed/type-parameterization.html

Egentyp + anonym klass för att ”injektera” beroenden

- Det går att få ett explicit, valfritt namn, t.ex. `self`, på ”mig själv”, alltså denna instans, genom att skriva `self =>` i kroppen på en trait eller klass.
- En typbegränsning på den egna instansen kallas **egentyp** (eng. *self type*).
- Användbart vid inmixning: du kan begränsa med vad denna trait får mixas in.

```
trait Grönsak { def vikt: Int }

trait KanSkalas:
  self: Grönsak =>
  def viktEfterSkalning = 0.99 * vikt
```

- Notera att *även* om `KanSkalas` *inte* gör **`extends`** så är *ändå* `vikt` tillgänglig i dess kropp, eftersom vi med egentypen kräver att `KanSkalas` är en `Grönsak`.

Egentyp + anonym klass för att ”injektera” beroenden

- Det går att få ett explicit, valfritt namn, t.ex. `self`, på ”mig själv”, alltså denna instans, genom att skriva `self =>` i kroppen på en trait eller klass.
- En typbegränsning på den egna instansen kallas **egentyp** (eng. *self type*).
- Användbart vid inmixning: du kan begränsa med vad denna trait får mixas in.

```
trait Grönsak { def vikt: Int }

trait KanSkalas:
  self: Grönsak =>
  def viktEfterSkalning = 0.99 * vikt
```

- Notera att *även* om `KanSkalas` *inte* gör `extends` så är *ändå* `vikt` tillgänglig i dess kropp, eftersom vi med egentypen kräver att `KanSkalas` är en `Grönsak`.
- Du kan du kombinera anonym klass och dynamisk inmixning med `with`:

```
scala> val g = new Grönsak with KanSkalas { val vikt = 100 }
val g: Grönsak & KanSkalas = anon1@70a91d72

scala> g.viktEfterSkalning
val res0: Double = 99.0
```

- Detta är ett elegant sätt att **injektera beroenden** (eng. *dependency injection*). `https://en.wikipedia.org/wiki/Dependency_injection`

Vad är fördelen med egentyper i stället för arv?

Arv tillåter **inte** ömsesidiga (cykliska) beroenden...

```
scala> trait Tulpan extends Ros; trait Ros extends Tulpan
-- [E110] Syntax Error: ----------------------------------------------------------------
1 |trait Tulpan extends Ros; trait Ros extends Tulpan
  |                     ^^^
  |                     Cyclic inheritance: trait Tulpan extends itself
  |
```

...medan egentyper *kan* vara **ömsesidigt beroende**:

```
trait Tulpan:
  self: Ros =>

trait Ros:
  self: Tulpan =>
```

```
scala> val tulipanaros = new Tulpan with Ros
val tulipanaros: Tulpan & Ros = anon1@1c63b39a
```

https://sv.wikipedia.org/wiki/Tulipanaros

Vad är ett bra api?

Vad är ett bra api?

- Ett api (eng. *Application Programming Interface*) är ett gränssnitt för applikationsprogrammering.
- Med ”gränssnitt” menas att det (i koden) finns en gräns mellan vad som syns utåt och vad som finns innanför ”under huven”.
- Det finns många kvalitetsaspekter som är önskvärda för ett api:

Vad är ett bra api?

- Ett api (eng. *Application Programming Interface*) är ett gränssnitt för applikationsprogrammering.
- Med ”gränssnitt” menas att det (i koden) finns en gräns mellan vad som syns utåt och vad som finns innanför ”under huven”.
- Det finns många kvalitetsaspekter som är önskvärda för ett api:
  - Enkelt att använda på utsidan även om insidan är komplex.
  - Vadå ”enkelt att använda”?
    - Lätt att lära
    - Lätt att komma ihåg
    - Lätt att begripa
    - Najs upplevelse (subjektivt)
  - Löser ett (generellt) problem på ett bra sätt. Vadå ”bra”?
    - Snabbt (hög prestanda)
    - Snålt (effektiv användning av resurser, t.ex. minne)
    - ...
- Intressant föredrag av Joshua Bloch (Google), om god api-design:
  - `https://research.google.com/pubs/archive/32713.pdf`
  - `https://youtu.be/aAb7hSCtvGw`

Api-desgin med Scala

- Kombinera paradigm OO+FP för att välja bäst lämpade lösningen.
- Avancerade abstraktionsmekanismer som kan vara utmanande för api-konstruktören men samtidigt bli enkelt för api-användaren
- Gör det möjligt att skapa ett api som fungerar i flera körmiljöer:
  - På desktop och i back-end med JVM och Graal VM
  - I front-end i webbläsaren med Scala JS
  - Direkt till plattformsspecifik maskinkod med Scala Native

Api-desgin med Scala

- Kombinera paradigm OO+FP för att välja bäst lämpade lösningen.
- Avancerade abstraktionsmekanismer som kan vara utmanande för api-konstruktören men samtidigt bli enkelt för api-användaren
- Gör det möjligt att skapa ett api som fungerar i flera körmiljöer:
  - På desktop och i back-end med JVM och Graal VM
  - I front-end i webbläsaren med Scala JS
  - Direkt till plattformsspecifik maskinkod med Scala Native
- Avancerade saker som vi inte gått in på i kursen men som kan hjälpa api-konstruktörer att göra lättanvänt api:
  - Kontextfunktioner `?=>`
  - Opaka typer `opaque type`
  - Typklassderivering `derives`
  - Metaprogrammering `inline`
  - Typ-lambda och match-typer `=>>`
  - ... (forskning pågår)

Köpa ett api för 55 miljarder?

Ericssons jätteaffär

Ericsson köper amerikanska Vonage, en plattform för app-utveckling, för 55 miljarder kronor. Strategin är att växa inom trådlösa företagslösningar.

–Vi är oerhört glada och uppåt här på Ericsson i dag, säger finanschefen Carl Mellander.

Mindre glada var aktieägarna. Att dra fördel av och få lönsamhet i sådana här stora affärer har historiskt visat sig vara svårt. Köpet är ett av de större i svensk företagshistoria.

Hade ni inte kunnat välja att samarbeta bara?

–Vi har utvärderat det. Kunde vi utveckla det här själv, ingå partnerskap? Vi landade i att förvärv var det bästa, den enda möjligheten att växa in i det här snabbt, säger Mellander.

Att plocka in mer mjukvara i Ericssonkoncernen bejon registrerade utvecklare globalt. Speciellt det sistnämnda är själva nyckeln, enligt Carl Mellander.

–Vad det här förvärvet handlar om, vi pratar om API:er, det är den mjukvara som gör att system och applikationer kan prata med varandra, säger Carl Mellander.

Han förklarar det som att Vonage är bryggan mellan app-utvecklaren och vad 5G-nätverken kan erbjuda.

FAKTA

Vonage i siffror

- Intäkterna under tredje kvartalet var 358 miljoner dollar, vilket under en rullande tolvmånadersperiod innebär årliga intäkter på 1,4 miljarder dollar.
- Efter skatt landade resultatet på minus 2 miljoner dollar under tredje kvartalet. Det justerade resultatet före avskrivningar blev plus 51 miljoner dollar.
- Aktien hade, innan affären med Ericsson annonserades, under året lyft med nästan 30

sta halvåret 202
har i dag över 5
kronor netto i k

Jonas Olavi, fon
på Alpcot, anser
är strategisk. Ha
att Ericsson har
på börstajminge
siktet inställt p
ska generera int
fram i tiden.

–Affären är li
kande, eftersor
har varit några
kulationer innar
man tänker till

Sydsvenskan 2021-11-23

Kontextuella abstraktioner

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**
- **Kombinationen** av OO och FP är **extra** kraftfull och flexibel!

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**
- **Kombinationen** av OO och FP är **extra** kraftfull och flexibel!
- Men det finns **svårigheter**:
  - OO: ibland svårt att resonera om ändrade tillstånd och dynamisk bindning
  - FP: kan bli många parametrar som måste upprepas vid nästlade anrop

Sammanhanget är avgörande när du kodar!

- **Kontexten**, alltså sammanhanget, styr vilka namn som syns var.
- **Objektorientering** (OO) skapar sammanhang med:
  - **Namnrymder**
  - **Tillstånd**
  - **Subtypspolymorfism**
- **Funktionsprogrammering** (FP) skapar sammanhang med:
  - **Parametrar**
  - **Funktionsvärden**
  - **Parametrisk polymorfism**
- **Kombinationen** av OO och FP är **extra** kraftfull och flexibel!
- Men det finns **svårigheter**:
  - OO: ibland svårt att resonera om ändrade tillstånd och dynamisk bindning
  - FP: kan bli många parametrar som måste upprepas vid nästlade anrop
- Till vår räddning: fler **coola grejer** i Scala:
  - **Kontextparametrar**: möjliggör att **givna** (implicita) värden kan framkallas automatiskt av kompilatorn.
  - **Ad hoc polymorfism**: möjliggör olika implementationer beroende på **statisk** typ utan att det krävs en arvsrelation eller dynamisk bindning.

Repetition: default-argument

- Vi har tidigare sett att kompilatorn, med **default-argument**, kan **fylla i** värden, som vi inte måste skriva, vid funktionsanrop:

```
scala> def f(x: Int, y: Int = 42) = x + y
def f(x: Int, y: Int): Int

scala> f(1, 2)        // explicit y-värde
val res0: Int = 3

scala> f(1)          // vi kan också skippa y-värdet
val res1: Int = 43   // då används default-argumentet
```

Repetition: uppdelade parameterlistor

- Vi har tidigare sett uppdelade parameterlistor, som möjliggör stegvis applicering (s.k. Curry-funktioner):

```
scala> def f(x: Int)(y: Int = 42) = x + y
def f(x: Int)(y: Int): Int

scala> val g = f(1)  // en ny funktion utan default-arg
val g: Int => Int = Lambda1352/0x08406d0840@dbc7e0a

scala> f(1)()   // y-värdet fylls i av kompilatorn
val res4: Int = 43
```

Repetition: uppdelade parameterlistor

- Vi har tidigare sett uppdelade parameterlistor, som möjliggör stegvis applicering (s.k. Curry-funktioner):

```
scala> def f(x: Int)(y: Int = 42) = x + y
def f(x: Int)(y: Int): Int

scala> val g = f(1)  // en ny funktion utan default-arg
val g: Int => Int = Lambda1352/0x08406d0840@dbc7e0a

scala> f(1)()   // y-värdet fylls i av kompilatorn
val res4: Int = 43
```

- Kan vi ta detta ett steg till och **frikoppla** deklarationen av funktionen **från** det givna värdet, och ge detta på annan plats baserat på typen?

Repetition: uppdelade parameterlistor

- Vi har tidigare sett uppdelade parameterlistor, som möjliggör stegvis applicering (s.k. Curry-funktioner):

```
scala> def f(x: Int)(y: Int = 42) = x + y
def f(x: Int)(y: Int): Int

scala> val g = f(1)  // en ny funktion utan default-arg
val g: Int => Int = Lambda1352/0x08406d0840@dbc7e0a

scala> f(1)()   // y-värdet fylls i av kompilatorn
val res4: Int = 43
```

- Kan vi ta detta ett steg till och **frikoppla** deklarationen av funktionen **från** det givna värdet, och ge detta på annan plats baserat på typen?
- JA! I Scala 3 görs detta med **givna** värden och **kontextparametrar**, som skapas med nyckelorden **`given`** respektive **`using`**
(i Scala 2 användes gamla nyckelordet **`implicit`**)

Givna värden + kontextparameter

Exmpel på användning av **given** och **using**:

```
case class Default(value: Int)

object Default:
  given d: Default = Default(0)  // Använd detta om inget annat givet värde finns lokalt
```

```
scala> def f(x: Int)(using d: Default) = x + d.value

scala> f(1)(using Default(2))    //explicit värde används alltid först om sådant finns
val res0: Int = 3

scala> f(1)  // kompilatorn framkallar ett givet värde för Default ur kompanjonsobjektet
val res1: Int = 1

scala> given d: Default = Default(42)  // låt d vara givna värdet i denna kontext
lazy val given_Default: Default

scala> f(1)       // kompilatorn framkallar nu det givna värdet i denna lokala kontext
val res2: Int = 43
```

Man kan utelämna namnet på värdet, eftersom det oftast inte behövs:
`given Default = Default(42)` eller ännu kortare: `given Default(42)`

Går det inte lika bra att ha en global variabel?

Varning för känsliga kodare!

- Ett försök att skapa ett default-värde med en global **`var`**-variabel:

```
scala> var globalVar = Default(42)
var ctx: Int = 42

scala> def f(x: Int, d: Default = globalVar) = x + d.value

scala> f(1)
val res3: Int = 43

scala> globalVar = Default(0) // FÖRÄNDRINGEN SLÅR GLOBALT I HELA DITT PROGRAM!!

scala> f(1)
val res4: Int = 1
```

- Här utgörs ”kontexten” av referensen till ett föränderligt värde som bestäms vid **körtid** i den globala namnrymden. Funktionen `f` är ej äkta!
- Denna lösning kan **inte** erbjuda **olika** kontextberoende default-värden; alla funktionsanrop använder **ett** globalt föränderliga värde. **Buggrisk**!

Givna värden med **`given`** härleds istället vid **kompileringstid** ur en speciell **implicit namnrymd** (eng. *implicit scope*) enligt speciella regler.

Import av kontextparameter

```
object EnNamnrymd:
  given enGivenSträng: String = "ett givet värde i EnNamnrymd"
  def framkalla(using s: String) = s     //kontextparametrar märks med using
```

Det är den **lokala** kontexten vid **användning** som styr vad som kan framkallas (och inte den namnrymd där kontextparametern deklarerades):

```
scala> EnNamnrymd.framkalla
-- [E172] Type Error: ----------------------------------------------------------
1 |EnNamnrymd.framkalla
  |                    ^
  | No given instance of type String was found for parameter s of method framkalla
  | in object EnNamnrymd. The following import might fix the problem:
  | import EnNamnrymd.enGivenSträng
```

Du kan importera givna värden med speciell syntax där typen anges istället för namnet:

```
scala> import EnNamnrymd.given String    // speciell import-syntax baserat på typ

scala> EnNamnrymd.framkalla
val res0: String = ett givet värde i EnNamnrymd
```

Framkalla värde med summon

I standardbiblioteket för Scala 3 finns en **generisk** variant av metoden `framkalla` i föregående exempel, som är definierad så här:

```
def summon[T](using x: T) = x
```

Funktionen `summon` kan användas för att testa vilket värde kompilatorn framkallar i en viss kontext:

```
object EnAnnanNamnrymd:
  given String = "tagen för givet"    // namn behövs ej, det räcker med typ
```

```
scala> summon[String]
-- Error:
1 |summon[String]
  |              ^
  | no implicit argument of type String was found for parameter x of
  | method summon in object Predef. The following import might fix the problem:
  | import EnAnnanNamnrymd.given_String

scala> import EnAnnanNamnrymd.given    // importerar alla givna värden i MinKontext

scala> summon[String]
val res0: String = tagen för givet
```

Prioritetsordning vid framkallning av givna värden

- Det kan finnas flera **olika** givna värden av **samma** typ om de finns i olika namnrymder.
- Det får **inte** vara **tvetydigt** vilket värde som ska framkallas.
- Därför finns det speciella prioriteringsregler:
  1. **Explicita** argument till kontextparametrar märkta med `using`
  2. `given` och `import given ...` i aktuell namnrymd (eng. *current scope*)
  3. `given`-värden i **kompanjonsobjekt** för den använda typen.
  4. ... (fler regler i speciella fall som vi inte går in på här)
- *Specialregel:* Om flera givna värden kan framkallas för typer som ingår i en gemensam **arvshierarki** så väljer kompilatorn det givna värdet som är av den **mest specifika** typen.

Framkallning av givna värden har flera namn på engelska:
to summon, eller *term inference*, eller *implicit resolution*.

Ad hoc polymorfism

- Med så kallad **Ad hoc polymorfism** kan du ha *olika* implementationer av en funktion för *olika* typer *utan* att du behöver använda arv och dynamisk bindning.
- Detta uppfanns i språket ML och vidareutvecklades i språket Haskell.
- I Haskell skapas Ad hoc polymorfism med en s.k. ”**typklass**” (eng. *type class*).
- Detta görs i Scala genom att kombinera typparametrar och kontextparametrar.
- En ”typklass” i Scala är en tillståndslös **`trait`** med minst en typparameter och minst en abstrakt metod.

```
trait Parser[T]: //en typklass för tolkning och omvandling av strängar till godtycklig typ
  def fromString(value: String): Option[T]

object Parser:            //implementationer för en viss typ kan erbjudas som givna värden
  given Parser[Int] with        //nyckelordet with behövs då abstrakt medlem implementeras
    def fromString(value: String): Option[Int] = value.toIntOption
```

- Den specifika implementationen framkallas vid anrop med kontextparameter:

```
scala> def användParser[T](s: String)(using p: Parser[T]) = p.fromString(s)

scala> användParser[Int]("12")        // hittar givet värde i kompanjonsobjektet
val res0: Option[Int] = Some(12)
```

Hur få typklassen `Parser` att funka för fler typer?

```
scala> användParser[java.awt.Color]("Color(120,10,0)")
-- Error:
1 |användParser[java.awt.Color]("Color(120,10,0)")
  |                                           ^
  | no implicit argument of type Parser[java.awt.Color] was found
  | for parameter p of method användParser
```

```scala
given Parser[java.awt.Color] with
  def fromString(value: String): Option[java.awt.Color] =
    if !value.startsWith("Color(") then None else
      val trimmed = value.trim.stripPrefix("Color(").stripSuffix(")")
      trimmed.split(",").map(_.toIntOption) match
        case Array(Some(r),Some(g),Some(b)) => Some(java.awt.Color(r, g, b))
        case _ => None
```

```
scala> användParser[java.awt.Color]("Color(120,10,0)")
val res1: Option[java.awt.Color] = Some(java.awt.Color[r=120,g=10,b=0])
```

Hur få typklassen `Parser` att funka för fler typer?

```
scala> användParser[java.awt.Color]("Color(120,10,0)")
-- Error:
1 |användParser[java.awt.Color]("Color(120,10,0)")
  |                                           ^
  | no implicit argument of type Parser[java.awt.Color] was found
  | for parameter p of method användParser
```

```scala
given Parser[java.awt.Color] with
  def fromString(value: String): Option[java.awt.Color] =
    if !value.startsWith("Color(") then None else
      val trimmed = value.trim.stripPrefix("Color(").stripSuffix(")")
      trimmed.split(",").map(_.toIntOption) match
        case Array(Some(r),Some(g),Some(b)) => Some(java.awt.Color(r, g, b))
        case _ => None
```

```
scala> användParser[java.awt.Color]("Color(120,10,0)")
val res1: Option[java.awt.Color] = Some(java.awt.Color[r=120,g=10,b=0])
```

Detta ger **flexibilitet**: Jag kan ge givna värden för mina **egna** typer!

Namnet på kontextparametrar kan utelämnas

- Namnet på det givna värdet behövs ofta inte – det är ju *typen* som är det viktiga.
- Därför är det tillåtet att utelämna parameternamnet vid **`using`**.
- Det givna värdet kan istället framkallas med `summon` vid behov:

```
def användParser[T](s: String)(using Parser[T]) =
  summon[Parser[T]].fromString(s)
```

Kontextgräns

Denna form av **using**-parameter med en typkonstruktor...

```
def användParser[T](s: String)(using Parser[T])
```

...är så vanlig i Scala att det finns kortare skrivsätt som kallas **kontextgräns**:

```
def användParser[T: Parser](s: String)
```

Om `F[A]` är en typkonstruktor och du skriver `[T: F]` så blir `F` en så kallad **kontextgräns** (eng. *context boundary*). Kompilatorn expandera detta automatiskt till kontextparametern `(using F[T])`

Ännu smidigare typklass med extensionsmetod

Det får att kombinera kontextgräns med extensionsmetoder:

```
extension [T: Parser](s: String)
  def parseOrElse(default: T): T =
    summon[Parser[T]].fromString(s).getOrElse(default)
```

Nu har vi smidig punktnotation:

```
scala> "12".parseOrElse(default = 42)
val res2: Int = 12

scala> "gurka".parseOrElse(default = 42)
val res3: Int = 42

scala> "Color(100,100,0)".parseOrElse(default = java.awt.Color.black)
val res4: java.awt.Color = java.awt.Color[r=100,g=100,b=0]

scala> "fel färg".parseOrElse(default = java.awt.Color.black)
val res5: java.awt.Color = java.awt.Color[r=0,g=0,b=0]
```

Sortera samlingar med given ordning

```
scala> case class Gurka(namn: String, vikt: Int)

scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))
val xs: Vector[Gurka] = Vector(Gurka(a,100), Gurka(b,50), Gurka(c,100))

scala> xs.sorted
-- Error:
1 |xs.sorted
  | ^
  | No implicit Ordering defined for B
  | where: B is a type variable with constraint >: Gurka
```

Sortera samlingar med given ordning

```
scala> case class Gurka(namn: String, vikt: Int)

scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))
val xs: Vector[Gurka] = Vector(Gurka(a,100), Gurka(b,50), Gurka(c,100))

scala> xs.sorted
-- Error:
1 |xs.sorted
  | ^
  | No implicit Ordering defined for B
  | where: B is a type variable with constraint >: Gurka
```

Detta kan fixas genom att tillhandahålla en given ordning för `Gurka`:

```
given Ordering[Gurka] with
  def compare(x: Gurka, y: Gurka): Int =
    if (x == y) then 0
    else if x.vikt < y.vikt then -1
    else 1
```

```
scala> xs.sorted   // nu funkar det :)
val res0: Vector[Gurka] = Vector(Gurka(b,50), Gurka(a,100), Gurka(c,100))
```

Sortera samlingar med ännu smidigare given ordning

- Det är vanligt att man vill definiera egna ordningsrelationer.
- Därför finns en smidig hjälpmetod i kompanjonsobjektet för typklassen `Ordering` som heter `fromLessThan`:

```
given Ordering[Gurka] =
  Ordering.fromLessThan((g1, g2) => g1.vikt < g2.vikt)
```

- Den tar som inparameter en funktion som tar två instanser som ska ordnas och ger **`true`** om den första ska anses **mindre** (alltså kommer **före**) enligt valfri definition.
- `fromLessThan` returnerar en `Ordering` som du kan låta vara givet.
- Prova gärna detta på veckans fördjupningsövningar.
- Läs mer om typklasser i Scala här: `https://docs.scala-lang.org/scala3/reference/contextual/type-classes.html`

Förslag på användning av kontextparameter i snake-labben

Antag att klassen Snake har två kontextparametrar:

```
class Snake (
  val initPos: Pos,
  val initDir: Dir,
  val headColor: Color,
  val tailColor: Color,
  )(using ctx: SnakeGame, settings: Settings) extends CanMove
```

- Var erbjuda default-värden för kontext-paramterar?

Förslag på användning av kontextparameter i snake-labben

Antag att klassen Snake har två kontextparametrar:

```
class Snake (
  val initPos: Pos,
  val initDir: Dir,
  val headColor: Color,
  val tailColor: Color,
  )(using ctx: SnakeGame, settings: Settings) extends CanMove
```

- Var erbjuda default-värden för kontext-paramterar?
  - Ett bra ställe att placera `given default: Settings = ???` är i kompanjonsobjekt till klassen `Settings`.
  - Du kan i kroppen på klassen `SnakeGame` ha en `given SnakeGame = this`
    Då kommer kompilatorn använda aktuell instans av `SnakeGame` som kontext-parameter när spelet i sin tur skapar instanser av `Snake`.

Förslag på användning av kontextparameter i snake-labben

Antag att klassen `Snake` har två kontextparametrar:

```
class Snake (
  val initPos: Pos,
  val initDir: Dir,
  val headColor: Color,
  val tailColor: Color,
  )(using ctx: SnakeGame, settings: Settings) extends CanMove
```

- Var erbjuda default-värden för kontext-paramterar?
  - Ett bra ställe att placera `given default: Settings = ???` är i kompanjonsobjekt till klassen `Settings`.
  - Du kan i kroppen på klassen `SnakeGame` ha en `given SnakeGame = this`
    Då kommer kompilatorn använda aktuell instans av `SnakeGame` som kontext-parameter när spelet i sin tur skapar instanser av `Snake`.

Notera denna text i labben, avsnitt *Tips och förslag*: ”Det är tillåtet att ändra, ta bort och lägga till, så länge de obligatoriska designkraven uppfylls.”

Översikt av kursens avslutning

Översikt av kursens avslutning

Återstående moment:

- W11: `snake` — Alla ska vara aktiva på redovisningen.
- W12-W13: Projekt — Individuellt arbete, välj gärna `bank`.
- W14: Munta — På schematider endast ons-tors.
  - Om du är väl förberedd för muntan kan du göra den redan W11–W13 så snart alla labbar t.om. snake är klara.
  - Prata med handledare om din pluggplan.
  - Förbered dig noga med hjälp av `https://cs.lth.se/pgk/muntabot`
  - Läs om muntan i kompendiet.
- Tentaperiod januari: Valfri tentamen för överbetyg. Anmälan enl. LTH:s normala rutiner.

Se schema: `https://cs.lth.se/pgk/schema/`
Mer info i kompendiet.

12 Fördjupning, Projekt

- Om projektuppgiften
- Avslutning: munta och tenta
- Repetition: Vad är en algoritm?
- Binärsökning
- Sortering

Kom på föreläsning om systemutveckling!

- Tors. **12/12 kl 15-17 i E:1406** ger Björn Regnell en föreläsning i C:arnas kurs om digitalisering EITA65:
  - **Kravhantering** för mjukvara: hur bestämma vad koda?
  - Hur bäst dra nytta av **öppen källkod**?
- Alla **D-are, W-are** som går pgk är också **hjärtligt välkomna!!!**

Om projektuppgiften

Om din avslutande projektuppgift

Läs noga kompendium Del 1, kapitel -1 avsnitt ”Projektuppgift”!
Några viktiga punkter:

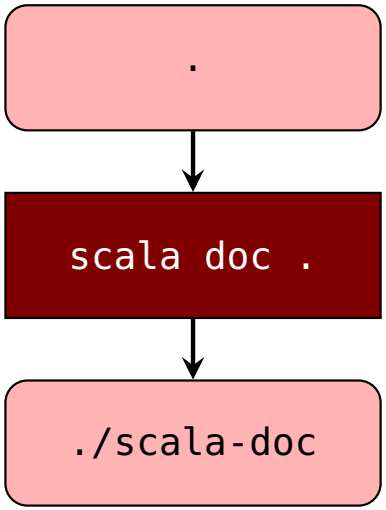
- Mål: skapa ett stort program med många samverkande klasser/moduler.
- Du väljer själv projektuppgift. Börja redan idag att planera ditt arbete!
- Projektet går över **två veckor**. Schemalagda labbar i december är **obligatoriska** för de som har kvar obligatoriska moment.
- I kompendium Del 2, kapitel 13, finns flera förslag att välja bland, men du kan också definiera ett eget projekt som passar just dig.
- Inför redovisningen ska du skapa automatiskt genererad dokumentation utifrån relevanta dokumentationskommentarer för minst hälften av dina publika metoder, enligt instruktioner i Appendix E.
- Redovisning sker i datorsal på schemalagd tid:
  - Förklara hur din kod fungerar.
  - Beskriv framväxten av ditt program.
  - Gå igenom den genererade dokumentationen av din kod.

Projektuppgifter

- `bank`
  - känd domän: skapa bank med transaktionshistorik
  - oföränderlig data tillsammans med tillståndsförändring
- `music`
  - skapa ett enkelt kompositionsverktyg som spelar musik
  - sätta sig in i en domänmodell
- `photo`
  - en enkel variant av photoshop
  - inblick i enkel matrismatematik
- egendefinierat projekt
  - Lagom svårt: ej för enkel uppgift, men ta dig inte vatten över huvudet!
  - Diskutera med en handledare och dokumentera egna uppgiften och dess omfattning och relation till lärandemålen i kursen så att andra handledare och kursansvarig kan förstå varför projektet passar i kursen.
  - Du måste få OK från en handledare innan du startar egendefinierat projekt.

Skapa dokumentation

Scala CLI kan ta dokumentationskommentarerna i källkoden och skapa en webbsajt med dokumentation.



Aktuell katalog med `.scala`-filer

`index.html` och en massa andra filer som ger en hel webbsajt!

Öppna `scala-doc/index.html` i en webbläsare.

Dokumentationskommentarer

För att kod ska bli begriplig för människor är det bra att dokumentera vad den gör. Det finns **tre olika sorters kommentarer**:

```
// Enradskommentarer börjar med dubbla snedstreck
//       men de gäller bara till radslut

/* Flerradskommentarer börjar med
   snedstreck-asterisk
   och slutar med asterisk-snedstreck.  */

/** Dokumentationskommentarer placeras före
 *   t.ex. en funktion och berättar vad den gör
 *   och vad eventuella parametrar används till.
 *   Börjar med snedstreck-asterisk-asterisk.
 *   Varje ny kommentarsrad börjar med asterisk.
 *   Avslutas med asterisk-stjärna.
 */
```

Kommentarer påverkar inte hur maskinen exekverar koden, men hjälper människor att använda koden och verktyg att visa hjälp. Se Appendix E.

Avslutning: munta och tenta

Avslutning och uppsamling

- Gör en **detaljerad** plan dag-för-dag för din kursavslutning.
- Föreläsningar denna vecka: om projekt, om munta, jämföra Scala–Java.
- Föreläsningarna nästa vecka består av repetition och tentaträning.
- Föreläsningarna anpassas efter era önskemål (se #önska i Discord).
- Inga föreläsningar sista läsveckan. **Sista pgk-föreläsningen** är tisdagen den 10:e december i E:A kl 13-15.
- Schematider i **sista läsveckan** är till för **muntligt prov, projektredovisning, redovisning av restlabbar**.
- Det är ok att redovisa projekt och göra munta i förtid i mån av plats. Men PLUGGA NOGA inför muntan!!! (Se nästa slajd.)
- Alla labbar klara innan du muntar (OK munta parallellt med projektet)
- För att få göra den valfria tentan krävs att **alla** obligatoriska moment (kontrollskrivning, labbar, projekt, muntligt prov) är **godkända**.
- **Godkänd** grundkurs är **krav** för att få påbörja efterföljande `pfk`.
- **Kolla din status i Canvas** på sida med inklippt ögonblicksbild ur vårt system SAM där du ser vad du har **klarat** och vad du ev. har kvar.

Obligatoriskt muntligt prov

- Syftet med det muntliga provet är att säkerställa att alla som påbörjar efterföljande kurs har tillräckliga **grundkunskaper**, med fokus på **konceptuell förståelse**.
- Det muntliga provet tar mellan 10 och 30 min beroende på hur många frågor handledare behöver ställa för att kunna kan avgöra om du har tillräcklig förståelse.
- Enda tillåtna hjälpmedel: snabbref, och på anmodan av handledare även REPL. Medtag papper och penna!
- **Alla labbar måste vara godkända** innan du får göra provet.
- Det är ok att göra det muntliga provet även innan projektet är klart, i mån av tid om det inte är redovisningskö.
- Om du är godkänd på alla obligatoriska moment får du minst 3:a i kursen och du har då klarat förkunskapskraven för efterföljande kurser.
- Du ber handledare att få göra muntligt prov på samma sätt som du tidigare bett om att få redovisa labbar. Munta sker på plats i datorsal.
- Träna på provet här: `https://cs.lth.se/pgk/muntabot`

Gör den valfria tentan!

- **Alla** som kan & vill uppmuntras göra den valfria tentan!
- **Alla** obligatoriska moment måste vara klara innan du får tentera.
- Den valfria tentan ger högre betyg om du klarar gränserna för 4:a el. 5:a.
- Tentan liknar de extentor som finns på kurshemsidan.
- Tentan är på plats och skrivs med papper och penna.
- Enda hjälpmedel: snabbreferensen.
- Snabbreferens beställs i Canvas och hämtas hos `birger.swahn@cs.lth.se`
- Tentan ges en gång om året i januari. Det är tillåtet att ”plussa” ett senare år om du inte är nöjd med ditt betyg.
- **OBS! Obligatorisk anmälan** i LTH:s ordinarie tentasystem. `https://www.student.lth.se/mina-studier/tentamen/`

Repetition: Vad är en algoritm?

Repetition: Vad är en algoritm?

Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: SWAP, MIN, Registrering, Sökning, Sortering

Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: SWAP, MIN, Registrering, Sökning, Sortering

Problemlösningsprocessens olika steg (inte nödvändigtvis i denna ordning):

- Dela upp problemet i enklare delproblem och sätt samman.
- Finns redan färdig lösning på (del)problem?
- Formulera (del)**problemet** och ange tydligt indata och utdata:
  exempel MIN: indata: sekvens av heltal; utdata: minsta talet
- Kom på en **lösningsidé**: (kan vara mycket klurigt och svårt)
  exempel MIN: iterera över talen och håll reda på ”minst hittills”
- Formulera en **stegvis beskrivning** som löser problemet:
  exempel: pseudo-kod med sekvens av instruktioner
- Implementera en **körbar lösning** i ”riktig” kod:
  exempel: en Scala-metod i en klass eller i ett singelobjekt
- Har algoritmen acceptabla tids- och minneskrav?

Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: SWAP, MIN, Registrering, Sökning, Sortering

Problemlösningsprocessens olika steg (inte nödvändigtvis i denna ordning):

- Dela upp problemet i enklare delproblem och sätt samman.
- Finns redan färdig lösning på (del)problem?
- Formulera (del)**problemet** och ange tydligt indata och utdata:
  exempel MIN: indata: sekvens av heltal; utdata: minsta talet
- Kom på en **lösningsidé**: (kan vara mycket klurigt och svårt)
  exempel MIN: iterera över talen och håll reda på ”minst hittills”
- Formulera en **stegvis beskrivning** som löser problemet:
  exempel: pseudo-kod med sekvens av instruktioner
- Implementera en **körbar lösning** i ”riktig” kod:
  exempel: en Scala-metod i en klass eller i ett singelobjekt
- Har algoritmen acceptabla tids- och minneskrav?

Det krävs ofta **kreativitiet** i stegen ovan – även i att **känna igen** problemet!
Simpelt exempel: Du stöter på problemet MAX och ser likheten med MIN.

Repetition: Vad är en algoritm?

En algoritm är en stegvis beskrivning av hur man löser ett problem.
Exempel: SWAP, MIN, Registrering, Sökning, Sortering

Problemlösningsprocessens olika steg (inte nödvändigtvis i denna ordning):

- Dela upp problemet i enklare delproblem och sätt samman.
- Finns redan färdig lösning på (del)problem?
- Formulera (del)**problemet** och ange tydligt indata och utdata:
  exempel MIN: indata: sekvens av heltal; utdata: minsta talet
- Kom på en **lösningsidé**: (kan vara mycket klurigt och svårt)
  exempel MIN: iterera över talen och håll reda på ”minst hittills”
- Formulera en **stegvis beskrivning** som löser problemet:
  exempel: pseudo-kod med sekvens av instruktioner
- Implementera en **körbar lösning** i ”riktig” kod:
  exempel: en Scala-metod i en klass eller i ett singelobjekt
- Har algoritmen acceptabla tids- och minneskrav?

Det krävs ofta **kreativitiet** i stegen ovan – även i att **känna igen** problemet!
Simpelt exempel: Du stöter på problemet MAX och ser likheten med MIN.

Övning: Diskutera hur du löser detta problem i relation till stegen ovan:
Att räkna antalet förekomster av olika unika ord i en textsträng.

Binärsökning

Binärsökning

Binärsökning är mycket snabbare än linjärsökning men kräver sorterad sekvens.

```
scala> val enMiljon = 1000000

scala> val xs = Array.tabulate(enMiljon)(i => i + 1)   // sorterad

scala> xs(enMiljon - 1)
res0: Int = 1000000

scala> timed { xs.indexOf(enMiljon) }        // linjärsökning
res42: (Double, Int) = (0.016622994,999999)

scala> import scala.collection.Searching.*  // ger tillgång till metoden search
import scala.collection.Searching._

scala> timed { xs.search(enMiljon) }        // binärsökning
res54: (Double, collection.Searching.SearchResult) = (2.45691E-4,Found(999999))
```

Binärsökning

Binärsökning är mycket snabbare än linjärsökning men kräver sorterad sekvens.

```
scala> val enMiljon = 1000000

scala> val xs = Array.tabulate(enMiljon)(i => i + 1)   // sorterad

scala> xs(enMiljon - 1)
res0: Int = 1000000

scala> timed { xs.indexOf(enMiljon) }        // linjärsökning
res42: (Double, Int) = (0.016622994,999999)

scala> import scala.collection.Searching.*  // ger tillgång till metoden search
import scala.collection.Searching._

scala> timed { xs.search(enMiljon) }        // binärsökning
res54: (Double, collection.Searching.SearchResult) = (2.45691E-4,Found(999999))
```

Citat från scaladoc för `search`:
”The sequence **should be sorted** before calling; otherwise, the results are **undefined.**”

Binärsökning: lösningsidé

Problemlösningsidé: Om sekvensen är sorterad kan vi utnyttja detta för en mer effektiv sökning, genom att jämföra med mittersta värdet och se om det vi söker finns före eller efter detta, och upprepa med ”halverad” sekvens tills funnet.

Binärsökning: lösningsidé

Problemlösningsidé: Om sekvensen är sorterad kan vi utnyttja detta för en mer effektiv sökning, genom att jämföra med mittersta värdet och se om det vi söker finns före eller efter detta, och upprepa med ”halverad” sekvens tills funnet.

- **Indata**: sorterad sekvens av heltal och det eftersökta elementet
- **Utdata**: `Found(index)` för det eftersökta elementet
  om saknas: `InsertionPoint(index)`

Binärsökning: lösningsidé

Problemlösningsidé: Om sekvensen är sorterad kan vi utnyttja detta för en mer effektiv sökning, genom att jämföra med mittersta värdet och se om det vi söker finns före eller efter detta, och upprepa med ”halverad” sekvens tills funnet.

- **Indata**: sorterad sekvens av heltal och det eftersökta elementet
- **Utdata**: `Found(index)` för det eftersökta elementet
  om saknas: `InsertionPoint(index)`

```
sealed trait SearchResult {
  def insertionPoint: Int
}

case class Found(foundIndex: Int) extends SearchResult {
  override def insertionPoint = foundIndex
}

case class InsertionPoint(insertionPoint: Int) extends SearchResult
```

Binärsökning: pseudokod, imperativ lösning

Pseudo-kod: imperativ lösning

```
def binarySearch(xs: Vector[Int])(elem: Int): SearchResult = {
  var found = false
  var (low, high) = (/* lägsta index */, /* högsta index */)
  var mid = /* något startvärde */
  while (!found && /* finns fler element kvar */) {
    mid = /* mittpunkten i intervallet (low, high) */
    if (xs(mid) == elem) found = true
    else if (xs(mid) < elem) /* flytta intervallets undre gräns */
    else /* flytta intervallets övre gräns" */
  }
  if (found) Found(mid)
  else InsertionPoint(low)
}
```

Binärsökning: implementation, imperativ lösning

Implementation: imperativ lösning

```
def binarySearch(xs: Vector[Int])(elem: Int): SearchResult = {
  var found = false
  var (low, high) = (0, xs.length - 1)
  var mid = -1
  while (!found && low <= high) {
    mid = (low + high) / 2
    if (xs(mid) == elem) found = true
    else if (xs(mid) < elem) low = mid + 1
    else high = mid - 1
  }
  if (found) Found(mid)
  else InsertionPoint(low)
}
```

Binärsökning: instrumentering av imperativ lösning

```
def waitForEnter: Unit = scala.io.StdIn.readLine("")
def show(msg: String): Unit = {println(msg); waitForEnter}

def binarySearch(xs: Vector[Int])(elem: Int): SearchResult = {
  var found = false                           ; show(s"found = $found")
  var (low, high) = (0, xs.length - 1)        ; show(s"(low, high) = ($low, $high)")
  var mid = -1                                ; show(s"mid = $mid")
  while (!found && low <= high) {             ; show(s"while ${!found && low <= high}")
    mid = (low + high) / 2                    ; show(s"mid = $mid")
    if (xs(mid) == elem) {found = true        ; show(s"found = $found")}
    else if (xs(mid) < elem) {low = mid + 1 ; show(s"low = $low")}
    else {high = mid - 1                      ; show(s"high = $high")}
  }
  if (found) Found(mid)
  else InsertionPoint(low)
}
```

```
scala> binarySearch(Vector(0,1,2,3,42,43))(42)
found = false
(low, high) = (0, 5)
mid = -1
while true
mid = 2
low = 3
while true
mid = 4
found = true
res0: collection.Searching.SearchResult = Found(4)
```

Binärsökning: rekursiv lösning

Fördjupning: rekursiv lösning

```
def binarySearch(xs: Vector[Int])(elem: Int): SearchResult = {
  def loop(low: Int, high: Int): SearchResult =
    if (low > high) InsertionPoint(low)
    else (low + high) / 2 match {
      case mid if xs(mid) == elem => Found(mid)
      case mid if xs(mid) < elem  => loop(mid + 1, high)
      case mid                    => loop(low, mid - 1)
    }

  loop(0, xs.length - 1)
}
```

Binärsökning: generisk rekursiv lösning

Fördjupning: iterativ generisk lösning med implicit ordning

```
def binarySearch[T](xs: Seq[T])(elem: T)(implicit ord: Ordering[T]): SearchResult = {
  import ord._
  def loop(low: Int, high: Int): SearchResult =
    if (low > high) InsertionPoint(low)
    else (low + high) / 2 match {
      case mid if xs(mid) == elem => Found(mid)
      case mid if xs(mid) < elem  => loop(mid + 1, high)
      case mid                    => loop(low, mid - 1)
    }

  loop(0, xs.length - 1)
}
```

För den intresserade:
Se fördjupningsuppgifter om implicita ordningar i veckans övning.

Tidskomplexitet, sökning

Fördjupning:

Algoritmteoretisk analys av sökalgoritmerna ger:

- Linjärsökning: tiden är proportionell mot n, skrivs: $O(n)$
- Binärsökning: tiden är proportionell mot $\log_2 n$, skrivs: $O(\log n)$

Tidskomplexitet, sökning

Fördjupning:
Algoritmteoretisk analys av sökalgoritmerna ger:

- Linjärsökning: tiden är proportionell mot n, skrivs: $O(n)$
- Binärsökning: tiden är proportionell mot $\log_2 n$, skrivs: $O(\log n)$

Empirisk analys: Vi har en vektor med 1000 element. Vi har mätt tiden för att söka upp ett element många gånger och funnit att det tar ungefär 1 μs både med linjärsökning och binärsökning.

Hur lång tid tar det om vi har fler element i vektorn?

	1000	10 000	100000	1 000 000	10 000 000
linjärsökning	1	10	100	1000	10 000
binärsökning	1	1.33	1.67	2.00	2.33

Kurserna:
Utvärdering av programvarusystem, obl. för D1, studerar detta **empiriskt**
Algoritmer, datastrukturer och komplexitet, obl. för D2, studerar detta **analytiskt**

Sortering

Sorteringsproblemet

Problem: Vi har en osorterad sekvens med heltal. Vi vill ordna denna osorterade sekvens i en sorterad sekvens från minst till störst.

Sorteringsproblemet

Problem: Vi har en osorterad sekvens med heltal. Vi vill ordna denna osorterade sekvens i en sorterad sekvens från minst till störst.

En *generalisering* av problemet:

Vi har många element av godtycklig typ och en **ordningsrelation** som säger vad vi menar med att ett element är *mindre än* eller *större än eller lika med* ett annat element.

Vi vill lösa problemet att ordna elementen i sekvens så att för varje element på plats i så är efterföljande element på plats $i + 1$ större eller lika med elementet på plats i.

Insättningssortering & Urvalssortering

- 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.

Insättningssortering & Urvalssortering

- 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.
- Urvalsssortering **lösningsidé**: **Välj ut** det minsta kvarvarande elementet i den osorterade listan och placera det **sist** i den sorterade listan och upprepa till det inte finns fler osorterade element.

Tidskomplexitet, sortering, medeltal

Urvalssortering, insättningssortering:	$O(n^2)$
”Bra” metoder, tex Quicksort, Timsort:	$O(n \log n)$

Vi har en vektor med 1000 element. Vi har mätt tiden för att sortera elementen många gånger och funnit att det tar ungefär 1 ms både med urvalssortering (eller någon annan ”dålig” metod) och en ”bra” metod.

Hur lång tid tar det om vi har fler element i vektorn?

	1,000	10,000	100,000	1,000,000	10,000,000
dålig	1	100	10^4	10^6	10^8
bra	1	13.3	167	2000	23000

Bogo sort

```
def bogoSort(xs: Vector[Int]) = {
  var result = xs
  while(result != result.sorted) {
    result = scala.util.Random.shuffle(result)
  }
  result
}
```

När blir denna färdig?

Bogo sort

```
def bogoSort(xs: Vector[Int]) = {
  var result = xs
  while(result != result.sorted) {
    result = scala.util.Random.shuffle(result)
  }
  result
}
```

När blir denna färdig?
`https://en.wikipedia.org/wiki/Bogosort`
Antal jämförelser i medeltal vid många mätningar: $O(n \cdot n!)$

Insättnings- och urvalssortering

Insertion sort

- Wikipedia:
  `https://sv.wikipedia.org/wiki/InsÃd'ttningssortering`
  `https://en.wikipedia.org/wiki/Insertion_sort`
- AlgoRythmics: Insert-sort with Romanian folk dance
  `https://www.youtube.com/watch?v=R0alU379l3U`

Selection sort

- Wikipedia:
  `https://sv.wikipedia.org/wiki/Urvalssortering`
  `https://en.wikipedia.org/wiki/Selection_sort`
- AlgoRythmics: Select-sort with Gypsy folk dance
  `https://www.youtube.com/watch?v=Ns4TPTC8whw`

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 */) {
     /* linjärsök rätt position i sorted */
     /* sätt in element på rätt plats i sorted */
  }
  sorted.toVector
}
```

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) {
     // linjärsök rätt position i sorted:
     var pos = 0
     while (pos < sorted.length && sorted(pos) < elem) {
       pos += 1
     }
     // sätt in element på rätt plats i sorted:
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

Sortera till ny vektor med urvalssortering: pseudo-kod

Det är nog lättare att förstå **selection 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 sorterad vektor med heltal

```
def selectionSort(xs: Vector[Int]): Vector[Int] = {
  val unsorted = xs.toBuffer
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  while (/* unsorted inte är tom */) {
    var indexOfMin = /* index för minsta element i unsorted */
    /* flytta elementet unsorted(indexOfMin) till sist i sorted */
  }
}
```

Sortera till ny vektor med urvalssortering: implementation

```
def selectionSort(xs: Vector[Int]): Vector[Int] = {
  val unsorted = xs.toBuffer
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  while (unsorted.nonEmpty) {
    var indexOfMin = 0
    // index för minsta element i unsorted:
    for (i <- 1 until unsorted.length) {
      if (unsorted(i) < unsorted(indexOfMin)) indexOfMin = i
    }
    val elem = unsorted.remove(indexOfMin)  // ta bort ur unsorted
    sorted.append(elem)  // lägg sist i sekvensen med sorterade
  }
  sorted.toVector
}
```

Sortera till ny vektor med urvalssortering: implementation

```
def selectionSort(xs: Vector[Int]): Vector[Int] = {
  val unsorted = xs.toBuffer
  val sorted = scala.collection.mutable.ArrayBuffer.empty[Int]
  while (unsorted.nonEmpty) {
    var indexOfMin = 0
    // index för minsta element i unsorted:
    for (i <- 1 until unsorted.length) {
      if (unsorted(i) < unsorted(indexOfMin)) indexOfMin = i
    }
    val elem = unsorted.remove(indexOfMin)  // ta bort ur unsorted
    sorted.append(elem)  // lägg sist i sekvensen med sorterade
  }
  sorted.toVector
}
```

- Funkar tom sekvens?
- Funkar en sekvens med ett element?
- Funkar det för osorterad sekvens med (minst) två element?
- Vad händer om sekvensen är sorterad?

Urvalssortering på plats: pseudo-kod

Indata: en array med heltal
Utdata: samma array, men nu sorterad

```
def selectionSortInPlace(xs: Array[Int]): Unit = {
  def minIndex(fromIndex: Int): Int = {
    /* index för minsta element från fromIndex  */
  }

  for (i <- /* från första till NÄST sista index */) {
    /* byt plats mellan xs[i] och xs[minIndex(i)] */
  }
}
```

Urvalssortering på plats: pseudo-kod

Indata: en array med heltal
Utdata: samma array, men nu sorterad

```
def selectionSortInPlace(xs: Array[Int]): Unit = {
  def minIndex(fromIndex: Int): Int = {
    /* index för minsta element från fromIndex  */
  }

  for (i <- /* från första till NÄST sista index */) {
    /* byt plats mellan xs[i] och xs[minIndex(i)] */
  }
}
```

Se animering här: Urvalssortering på Wikipedia

Urvalssortering på plats: implementation

Indata: en array med heltal
Utdata: samma array, men nu sorterad

```
def selectionSortInPlace(xs: Array[Int]): Unit = {
  def minIndex(fromIndex: Int): Int = {
    var result = fromIndex
    for (i <- fromIndex + 1 until xs.length) {
      if (xs(i) < xs(result)) result = i
    }
    result
  }

  def swap(i: Int, j: Int): Unit = {
    val temp = xs(i)
    xs(i) = xs(j)
    xs(j) = temp
  }

  for (i <- 0 until xs.length - 1) { // till NÄST sista
    swap(i, minIndex(i))
  }
}
```

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) {  //från ANDRA till sista
    var j = i
    while (j > 0 && xs(j - 1) > xs(j)) {
      /* byt plats på xs(j) och xs(j - 1) */
      j -= 1;  // stega bakåt
    }
  }
}
```

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) {  //från ANDRA till sista
    var j = i
    while (j > 0 && xs(j - 1) > xs(j)) {
      /* 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!

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
  }
  for (i <- 1 until xs.length) {  //från ANDRA till sista
    var j = i
    while (j > 0 && xs(j - 1) > xs(j)) {
      swap(j, j - 1)
      j -= 1;  // stega bakåt
    }
  }
}
```

Sortera samlingar 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) {  // insertion sort using lt as "less than"
     var pos = 0
     while (pos < sorted.length && lt(sorted(pos), elem)) {
       pos += 1
     }
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

Sortera samlingar 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) {  // insertion sort using lt as "less than"
     var pos = 0
     while (pos < sorted.length && lt(sorted(pos), elem)) {
       pos += 1
     }
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

```
scala> val xs = Vector(1,2,1,2,12,42,1)
xs: scala.collection.immutable.Vector[Int] = Vector(1, 2, 1, 2, 12, 42, 1)

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

scala> sortWith(xs)(_ > _)
res1: Vector[Int] = Vector(42, 12, 2, 2, 1, 1, 1)
```

Fördjupning: Sortera samlingar av godtycklig typ

```
def sortWith[T](xs: Vector[T])(lt: (T, T) => Boolean ): Vector[T] = {
  val sorted = scala.collection.mutable.ArrayBuffer.empty[T]
  for (elem <- xs) {
     var pos = 0
     while (pos < sorted.length && lt(sorted(pos), elem)) {
       pos += 1
     }
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

Fördjupning: Sortera samlingar av godtycklig typ

```scala
def sortWith[T](xs: Vector[T])(lt: (T, T) => Boolean ): Vector[T] = {
  val sorted = scala.collection.mutable.ArrayBuffer.empty[T]
  for (elem <- xs) {
     var pos = 0
     while (pos < sorted.length && lt(sorted(pos), elem)) {
       pos += 1
     }
     sorted.insert(pos, elem)
  }
  sorted.toVector
}
```

```
scala> case class Gurka(namn: String, vikt: Int)

scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))

scala> sortWith(xs)(_.vikt > _.vikt)
res0: Vector[Gurka] = Vector(Gurka(c,100), Gurka(a,100), Gurka(b,50))

scala> sortWith(xs)(_.vikt >= _.vikt)
res1: Vector[Gurka] = Vector(Gurka(a,100), Gurka(c,100), Gurka(b,50))
```

Vad menas med att en sorteringsalgoritm är stabil?

En sorteringsalgoritm är **stabil** om **ordningen** mellan element som anses **lika** enligt sorteringsordningsrelationen **bevaras**.

Fördjupning: en.wikipedia.org/wiki/Sorting_algorithm#Stability

Sortera samlingar med inbyggda sortBy och sortWith

```
scala> case class Gurka(namn: String, vikt: Int)
defined class Gurka

scala> val xs = Vector(Gurka("a", 100), Gurka("b", 50), Gurka("c", 100))

scala> xs.sortBy(_.vikt)
res0: scala.collection.immutable.Vector[Gurka] =
        Vector(Gurka(b,50), Gurka(a,100), Gurka(c,100))

scala> xs sortWith (_.vikt > _.vikt)
res1: scala.collection.immutable.Vector[Gurka] =
        Vector(Gurka(a,100), Gurka(c,100), Gurka(b,50))
```

13 Repetition

- Repetition på begäran
- Tentatips
- Utblick och avslutning

Repetition på begäran

På begäran 2024

Grumligt

1. När är det bra/dåligt att använda anonyma funktioner? w03
2. Klasser och kompanjonsobjekt: vad passar bäst var? w05
3. Hur göra felhantering med `Option` och `Try`? w06
4. Skillnaden mellan sats & uttryck, tex **`if`**, **`for`**? w01

Nyfiken-på

1. Flertrådad programmering
2. Fönsterhantering i introprog under huven
3. Generiska typgränser **`<: >:`**

Repetition: Tumregler/tips vid val av abstraktion

Extensionsmetod, singelobjekt, case-klass, klass, trait, eller enum?

- Om du vill lägga till en metod på befintlig typ utan behov av nya attribut etc., använd `extension`.
- Använd `object` om du behöver samla metoder (och variabler) i en modul som bara finns i en upplaga. Du får lokal namnrymd och punktnotation på köpet.
- Behöver du modellera **oföränderlig data**, använd en `case class` eller `enum`.
- Om du vill ha uppräknade värden som du vill kunna iterera över och matcha på i förseglad struktur, med värden i egen namnrymd, använd `enum`.
- Med `case class` och `enum` får du även innehållslikhet och en massa annat godis på köpet!
- Behöver du **förändringsbart tillstånd** (eng. *mutable state*) använd en vanlig `class`. Det normala är att det föränderliga tillståndet (de attribut som är föränderliga) är `private` eller `protected` och att all uppdatering och avläsning av tillståndet sker indirekt genom metoder (getters/setters/...).
- Behöver du en abstrakt bastyp använd en `trait`, speciellt om du vill ha möjlighet till inmixning. Om du vill förhindra inmixning eller underlätta användning från Java, använd `abstract class`.

Repetition: Tips om val av samling

Det är ofta enklare med oföränderliga samlingar med oföränderliga element och skapa nya samlingar vid förändring. Men för vissa algoritmer blir det enklare eller effektivare om du ändrar på plats i förändringsbar samling.

- Behöver du hantera värden i sekvens?
  - Om du klarar dig utan förändring av innehållet efter konstruktion: `val`-referens till `Vector`
  - Om du behöver ändra innehåll men **inte** antal element: `val`-referens till `Array`
  - Om du behöver ändra innehåll **och** antal element: `var`-referens till `Vector` och t.ex. metoden `patch`, eller `val`-referens till `ArrayBuffer` och t.ex. metoden `insert`
- Behöver du hantera värden x som ska vara unika?
  - Oföränderlig: `Set`
  - Förändringsbar: `val`-referens till `scala.collection.mutable.Set`
- Behöver du leta upp värden `x:Int` utifrån en nyckel av t.ex. String?
  - Oföränderlig: `Map[String, Int]`
  - Förändringsbar: `val`-referens till `scala.collection.mutable.Map[String, Int]`

Tentatips

Före tentan:

1. Repetera övningar och labbar i kompendiet.
2. Läs igenom föreläsningsanteckningar.
3. Studera **snabbref** **mycket noga** så att du vet vad som är givet och var det står, så att du kan hitta det du behöver snabbt.
4. Skapa och **memorera** en personlig **checklista** med programmeringsfel du brukar göra, som även inkluderar småfel, så som glömda parenteser och semikolon, och annat som en kompilator/IDE normalt hittar.
5. Tänk igenom hur du ska disponera dina 5 timmar på tentan.
6. Gör minst en extenta som om det vore **skarpt läge**:
    1. Avsätt 5 ostörda timmar (stäng av telefon, dator etc).
    2. Inga hjälpmedel. Bara snabbref.
    3. Förbered dryck och tilltugg.

På tentan:

1. Läs igenom **hela** tentan först.
   Varför? Förstå helheten. Delarna hänger ihop.
2. Notera och begrunda specifika begrepp och definitioner.
   Varför? Begreppen är avgörande för förståelsen av uppgiften.
3. Notera förenklingar, antaganden och specialfall.
   Varför? Uppgiften blir mkt enklare om du inte behöver hantera dessa.
4. **Fråga** tentamensansvarig om du inte förstår uppgiften – speciellt om det finns misstänkta felaktigheter eller förmodat oavsiktliga oklarheter.
   Varför? Det är inte lätt att konstruera en ”perfekt” tenta.
   Du får fråga vad du vill, men det är inte säkert du får svar...
5. Läs specifikationskommentarerna och metodsignaturerna i alla givna klass-specifikationer **mycket noga**.
   Varför? Det är ett vanligt misstag att förbise de ledtrådar som ges där.
6. Återskapa din memorerade personliga checklista för vanliga fel som du brukar göra och avsätt tid till att gå igenom den på tentan. Varje fix plockar poäng!
7. Lämna in ett försök även om du vet att lösningen inte är fullständig. Det gäller att plocka så många poäng det går. En ofullständig lösning kan ändå ge poäng.
8. Om du har svårigheter kan det bli kamp mot klockan. Försök hålla huvudet kallt och prioritera utifrån var du kan plocka flest poäng. Ge inte upp! Ta en kort äta-dricka-paus för att få mer energi!

Planeringstips

Exempel på saker som du kan lägga in tid för i din julpluggkalender:

1. Ta reda på vad just **du** behöver träna på!
2. Välja ut övningar att repetera.
3. Repetera övning X, Y, Z, ... Både läsa och skriva kod. Fundera på typ och värde.
4. Välja ut labbar att repetera.
5. Repetera labb X, Y, Z, ... Lär dig ”trick” och ”mönster”.
6. Träna på att skriva program med papper och penna.
7. Gör så många extentor du orkar, simulera ”skarp läge”.
8. Gör en checklista med vanliga fel och misstag som du brukar göra.
9. Läsa igenom alla de extentor som du väljer att inte göra ”i fiktivt skarpt läge” och studera generella mönster och typiska trick.

Tentans struktur

- Del A 20%:
  Evaluera uttryck där du ska **ange typ och värde**
  - Testar förståelse av variabler, uttryck, samlingar, algoritmer, arv, etc.
  - Det är bra/nödvändigt att anteckna delsteg och variablers värden, då det är mycket svårt att tänka ut svaren direkt i huvudet.
- Del B 80%:
  Skriva kod som uppfyller **krav och design**
  - Testar att du själv kan skapa kod med delar som samverkar
  - Testar förmåga att gå från indata-utdata till algoritm
    givet: ledtrådar, design, ev. skiss på lösning, ev. pseudokod etc.
- Blanka inlämningar ger 0 poäng; det är alltid bättre att försöka än att lämna in blankt. Lämna inte in kladdpapper eller dubbla lösningar.

Vad kommer på tentan?

- Grundläggande begrepp och det som tränas på grundövningar och labbar är basen för att bli godkänd.
- Begrepp, föreläsningsbilder och övningar som är markerade **”fördjupning”** krävs ej för att klara tentan men ökar förståelsen och hjälper dig att nå högre betyg.
- Det är helt ok på tentan om du väljer en **enkel lösning med basala begrepp som fungerar bra**, i stället för en kortare/elegantare/mer avancerad lösning.
- Extra-övningarna i läsvecka 12 ingår ej på tentan.

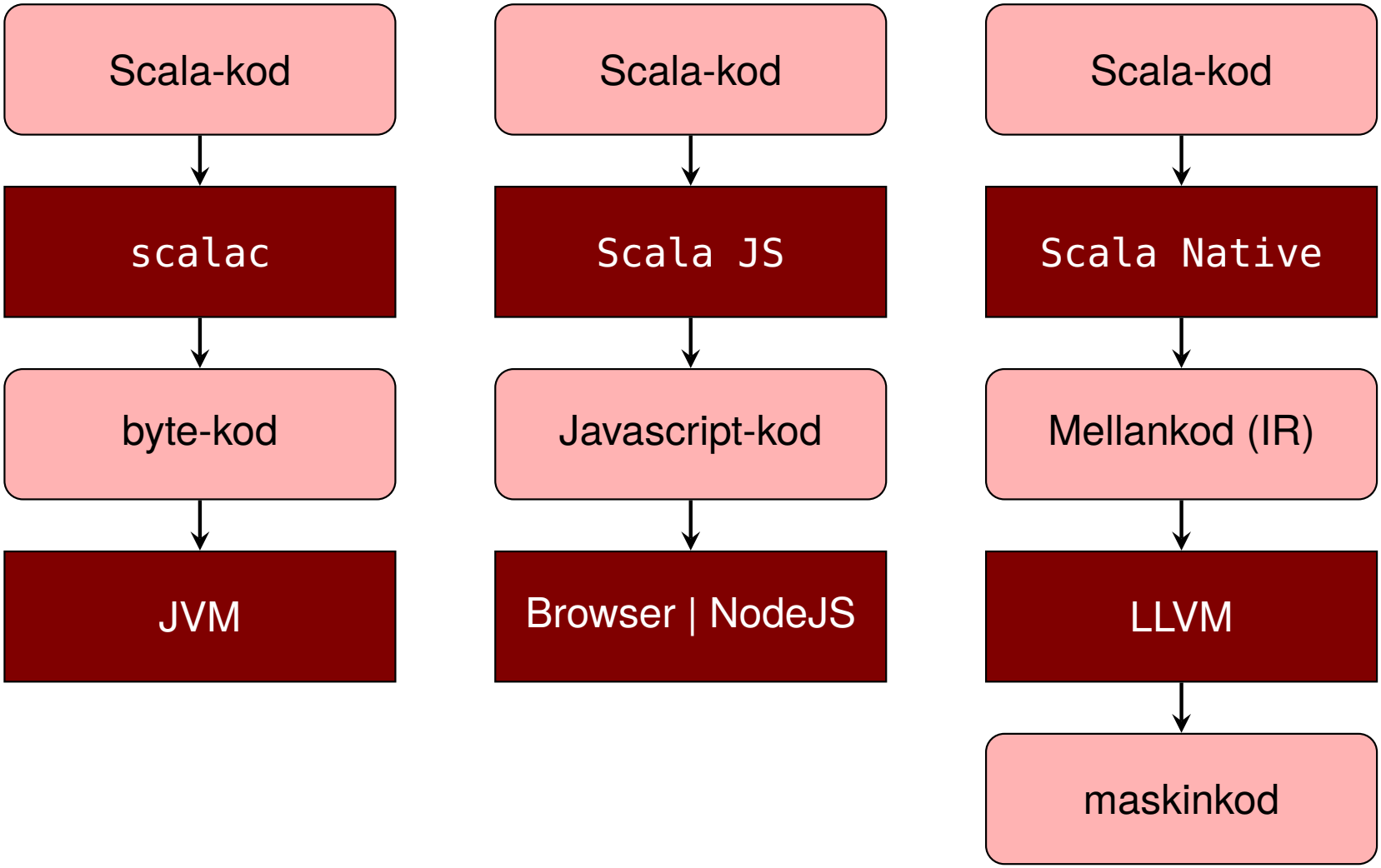
Utblick och avslutning

Scala då, nu och i framtiden

- Scala 1.0 (2003) första pre-release
- Scala 2.0-2.9 (2006-2011) pionjärer: Twitter, LinkedIn, The Guardian, ...
- Scala 2.10 (2013) brett genombrott, viktiga språkutvidgningar
- Scala 2.11 (2014) allmän industriell spridning, stabilitet, prestanda,
- Scala 2.12 (2016) fokus på prestanda, snabbare bytekod med lambda i JVM Java 8
- Scala 2.13 (2019) fokus på standardbiblioteket och `scala.collection`, migreringsverktyg för Scala 3
- Scala 3 (2021): **stort tekniksprång** med många nya språkkonstruktioner

Läs mer om historik här:
`https://en.wikipedia.org/wiki/Scala_(programming_language)`

Scala på JVM, Scala JS, Scala Native



Hur håller jag mig uppdaterad om Scalas utveckling?

- Officiell blog: `https://www.scala-lang.org/blog/`
- Scala-nyheter: `http://scalatimes.com/`
- User-forum: `https://users.scala-lang.org/`
- Tjatt: `https://discord.gg/h9452YPJ`
- Scala Center: `https://scala.epfl.ch/`
- Scala-bibliotek: `https://index.scala-lang.org/`
- Contributors: `https://contributors.scala-lang.org/`
- Scala-språkets pågående förbättring: `https://docs.scala-lang.org/sips`

CEQ – Course Experience Questionnaire

- Görs på hela LTH på samma sätt. Alla får länkar via mejl.
- Snälla fyll i CEQ! Jag är **mycket tacksam** för all konstruktiv feedback! Hög svarsfrekvens är viktigt för att kunna dra slutsatser om variationen i svaren och signifikansen i sammanställningen.
- Del 1: Generella påståenden, alla med 5-gradig skala:
  tar helt avstånd ... instämmer helt
- Del 2: **Fritextfrågor**:
  ”Vad tycker du var det bästa med den här kursen?”
  ”Vad tycker du främst behöver förbättras?”
- Mer om CEQ här: `https://www.ceq.lth.se/`
- **Fördel** med CEQ: Samma alla kurser alla år medger jämförelse över tid.
- **Begränsning**: Saknar frågor kopplat till specifika kursmoment.

Kursspecifik utvärdering om specifika kursmoment

- Jag vill gärna att **alla** gör den LTH-gemensamma, anonyma kursutvärderingsenkäten CEQ. Dina fritext-kommentarer om vad som är det bästa med kursen och vad som främst behöver förbättra emottages mycket tacksamt i CEQ-utvärderingen!
- Jag kommer att komplettera CEQ med en **kursspecifik utvärdering** av specifika kursmoment i denna kurs och jag är därför **mycket tacksam** om alla fyller enkäten när länk kommer via anslag i Canvas.
- Jag behandlar dina svar **konfidentiellt**, men sparar din email så att jag kan återkomma om jag mot förmodan undrar något mer.
- Din input är **mycket värdefull** vid framtida kursutveckling!

Intresserad av att arbeta som handledare?

- Vi har ständigt behov av nya handledare i våra kurser
- Det är lärorikt att jobba som handledare
- Kontakta `bjorn.regnell@cs.lth.se`
  eller annan kursansvarig i den kurs du vill jobba
- Jag påbörjar intervjuer av kandidater pgk+dod **tidig vår**

`https://cs.lth.se/utbildning/`
”Arbeta som Övningsassistent”

Ett stort TACK...

- ... till alla **handledare** som jobbat hårt för att ni ska lära er så mycket som möjligt!
- ... till alla **studenter** som gått kursen för:
  - ... att ni kämpat så hårt!
  - ... att ni ställt massor med frågor!
  - ... att det har varit så hög närvaro på föreläsningarna!
  - ... att ni hjälp till med värdefull återkoppling!
  - ... att ni är så konstruktiva och verkligen vill lära er!

Ett stort TACK...

- ... till alla **handledare** som jobbat hårt för att ni ska lära er så mycket som möjligt!
- ... till alla **studenter** som gått kursen för:
  - ... att ni kämpat så hårt!
  - ... att ni ställt massor med frågor!
  - ... att det har varit så hög närvaro på föreläsningarna!
  - ... att ni hjälp till med värdefull återkoppling!
  - ... att ni är så konstruktiva och verkligen vill lära er!

Ett stort LYCKA TILL på vägen till att bli en kompetent och innovativ systemutvecklare!

Hoppas att pgk-kursen varit givande!

14 MUNTLIGT PROV

- Sista läsveckan

Sista läsveckan

Sista läsveckan

- Det är inga föreläsningar denna vecka.
- Endast resurstider schemalagda sista veckan, se `https://cs.lth.se/pgk/schema/timeedit/`
- Se instruktioner för **muntligt prov** här: `https://fileadmin.cs.lth.se/pgk/lect-w12.pdf`
- Gör klart och redovisa **alla** labbar och projektet
- Senaste datum för anmälning till valfria tentamen är i mitten av december, se: www.student.lth.se/mina-studier/tentamen/

15 Java

- Jämförelse Scala och Java
- Array i Java
- Autoboxning i Java
- Equals i Java

Jämförelse Scala och Java

Övning `scalajava` och labb `javatext`

Valfria uppgifter som ger dig en **flygande start** i efterföljande kurs:

- Övning `scalajava`:
  - Översätta från Java till Scala och från Scala till Java
  - Undersöka autoboxning (eng. *autoboxing*)
  - Använda `import scala.jdk.CollectionConverters.*`
- Laboration `javatext`:
  - Gör ett textspel för terminalen huvudsakligen i Java men vissa delar i Scala, enligt krav, tips och inspiration i labb-instruktionerna.
- OBS! Scala CLI och sbt kan blanda `.scala` och `.java` i samma projekt.

```
> scala run .
```

Ovan kommando kommer också kompilera `.java`-filer.
Observera Javas filnamnsregler: klass == filnamn, paket == katalog
Bra plats att lägga `.java`-filer: `src/main/java/`
Bra plats att lägga `.scala`-filer: `src/main/scala/`

”Hello world!” i Java.

Ett minimalt huvudprogram i Java:

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Testa Java i `jshell`

- Java har en motsvarighet till Scalas REPL: kommandot `jshell`

```
> jshell
|  Welcome to JShell -- Version 11.0.11
|  For an introduction type: /help intro

jshell> /help intro
|
|                                    intro
|                                    =====
|
|  The jshell tool allows you to execute Java code, getting immediate results.
|  You can enter a Java definition (variable, method, class, etc), like:  int x = 8
|  or a Java expression, like:  x + x
|  or a Java statement or import.
|  These little chunks of Java code are called 'snippets'.
|
|  There are also the jshell tool commands that allow you to understand and
|  control what you are doing, like:  /list
|
|  For a list of commands: /help

jshell>
```

Grundläggande likheter och skillnader Java–Scala

Några likheter:

- Kompilerar till bytekod som kör på JVM på många olika plattformar.
- Statisk typning: ger snabb maskinkod, kompilatorn kan ge stöd vid förändring av kod (s.k. refactoring) och hittar många buggar redan vid kompilering.

Liknande men viss skillnad:

Java

- **Objektorientering**, men inte ”äkta” (eng. *pure*) eftersom inte alla värden är objekt
- Primitiva typer är inte objekt; representeras effektivt, normalt **utan boxning**
- Visst stöd för **funktionsprogrammering**

Scala

- **Äkta objektorienterat** eftersom alla värden är objekt, även funktioner
- `AnyVal`-instanser är äkta objekt men representeras ändå effektivt, normalt **utan boxning**
- Omfattande stöd för **funktionsprogrammering**

Huvudprogram i Scala och Java

Scala 2

```scala
object Main {
  def main(args: Array[String]): Unit = {
    println("Hello!")
  }
}
```

Java

```java
public class JMain {
  public static void main(String[] args){
    System.out.println("Hello!");
  }
}
```

Huvudprogram i Scala och Java

Scala 2

```scala
object Main {
  def main(args: Array[String]): Unit = {
    println("Hello!")
  }
}
```

Java

```java
public class JMain {
  public static void main(String[] args){
    System.out.println("Hello!");
  }
}
```

Scala 3

```scala
@main
def sumFirst(n: Int, xs: Int*): Unit = println(xs.take(n).sum)
```

Loopa genom argumenten i ett Java-huvudprogram

```
> code HelloJavaArgs.java
```

```java
public class HelloJavaArgs {
    public static void main(String[] args) {
    int i = 0;
    while (i < args.length) {
      System.out.println(args[i]);
      i = i + 1;
    }
  }
}
```

Kompilera och kör:

```
> javac HelloJavaArgs.java
> java HelloJavaArgs hej gurka tomat
hej
gurka
tomat
```

HIGHSCORE implementerad i Java

```
import java.util.Scanner;

public class HighScore {
    public static void main(String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println("Hur många poäng fick du?");
        int points =  scan.nextInt();
        System.out.println("Vad var highscore före senaste spelet?");
        int highscore = scan.nextInt();
        if (points > highscore) {
            System.out.println("GRATTIS!");
        } else {
            System.out.println("Försök igen!");
        }
    }
}
```

Några saker som finns i Scala men inte i Java

- `case`-klasser
- Lokala funktioner
- Metoder som operatorer
- Infix operatornotation
- Defaultargument
- Namngivna argument
- Engångsinitialisering: `val`
- Fördröjd initialisering: `lazy val`
- Enhetlig access för `def`, `val`, `var`
- Egna setters med `def namn_=`
- Namnanrop, fördröjd evaluering
- Matchning, mönster och garder
- Klassparametrar, primärkonstruktor
- Singelobjekt: `object`
- Kompanjonsobjekt
- Inmixning: `trait`
- `for`-`yield`-uttryck
- Block är uttryck; slipper `return`
- Tomma värdet () av typen `Unit`
- `Option`, `Some`, `None` (Java har `Optional` som ger en del, men inte allt...)
- `Try`, `Success`, `Failure`
- Samlingarna i Scalas standardbibliotek, speciellt de **oföränderliga** samlingarna `Vector`, `Map`, `Set`, `List`, etc.
- Innehållslikhet med == för oföränderliga strukturer, inkl. `<` `<=` `>` `>=` på strängar
- **Enhetlig** användning av samlingar **inkl. Array** (förutom innehållslikhet för Array)
- Kontextuella abstraktioner `given` `using`
- Mer precis synlighet `private[mypack]`
- Namnändring vid `import`
- Flexibel filstruktur och filnamngivning
- Flexibel nästling av klasser, objekt, traits
- Typ-alias och abstrakta typer med `type`
- Extensionsmetoder `extension`
- ...

Några saker som finns i Java men inte i Scala

- \+ Snabbare kompilering
- \+ Mognare verktygsstöd
- Variabeldeklaration utan initialisering
- Förändringsbara parametrar
- C-liknande prefix- och postfix-inkrementering och -dekrementering: `i++ ++i i-- --i`
- C-liknande **`for`**-sats
- Semikolon krävs efter alla satser
- **`return`** krävs i alla metoder som har returvärde

- Nyckelordet **`void`**
- Parenteser efter alla metoder
- Specialsyntax för indexering av array `[]` ej som i andra samlingar
- Hoppa ut ur loop med **`break`**
  docs.oracle.com/javase/tutorial/
- **`switch`** ”faller igenom” om du glömmer skriva **`break`**
- Kontrollerade undantag (eng. *checked exceptions*) och **`throws`**
- ...

Begränsningar med funktionsprogrammering i Java

Av alla dessa funktionsprogrammeringskoncept i Scala...

- **överlagring**
- **anonyma funktioner**
- **mönstermatchning**
- funktioner etc. på toppnivå
- utelämna tom parameterlista (enhetlig access)
- defaultargument
- namngivna argument
- lokala funktioner
- funktioner som äkta värden
- klammerparentes vid ensam parameter
- multipla parameterlistor
- egendefinierade kontrollstrukturer
- namnanrop (fördröjd evaluering)
- stegade funktioner (”Curry-funktioner”)
- fångad variablelrymd i funktionsobjekt (”closure”)
- ad hoc polymorfism (”typklasser”)
- kontextuella abstraktioner

...kan man i Java endast göra: **överlagring** (”overloading”), **anonyma funktioner** (”lambda”), **mönstermatchning** (de senare två har starka begränsningar)

Läs mer om Java här:
`https://en.wikipedia.org/wiki/Java_version_history`
`https://en.wikipedia.org/wiki/Anonymous_function#Java_Limitations`

Grundtyper i Scala och primitiva typer Java

Grundtyp i **Scala**	Antal bitar	Omfång minsta/största värde	**primitiv typ** i **Java**
`Byte`	8	$-2^7 \ldots 2^7 - 1$	`byte`
`Short`	16	$-2^{15} \ldots 2^{15} - 1$	`short`
`Char`	16	$0 \ldots 2^{16} - 1$	`char`
`Int`	32	$-2^{31} \ldots 2^{31} - 1$	`int`
`Long`	64	$-2^{63} \ldots 2^{63} - 1$	`long`
`Float`	32	$\pm\, 3.4028235 \cdot 10^{38}$	`float`
`Double`	64	$\pm\, 1.7976931348623157 \cdot 10^{308}$	`double`

Javas switch-sats

De flesta C-liknande språk (men inte Scala) har en **`switch`**-sats som man kan använda istället för (vissa) nästlade if-else-satser:

```
import java.util.Scanner;

public class Switch {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv grönsak:");
        String g = scan.next();
        switch (g) {
        case "gurka":
            System.out.println("gott!");
            break;
        case "tomat":
            System.out.println("gott!");
            break;
        case "broccoli":
            System.out.println("ganska gott...");
            break;
        default:
            System.out.println("mindre gott...");
            break;
        }
    }
}
```

`switch` från Java 21 har begränsad mönstermatchning `https://docs.oracle.com/en/java/javase/21/language/pattern-matching.html`

Javas switch-sats utan break

Saknad **break**-sats ”faller igenom” till efterföljande gren:

```
import java.util.Scanner;

public class SwitchNoBreak {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv grönsak:");
        String g = scan.next();
        switch (g) {
        case "gurka":
        case "tomat":
            System.out.println("gott!");
            break;
        case "broccoli":
            System.out.println("ganska gott...");
            break;
        default:
            System.out.println(" mindre gott...");
            break;
        }
    }
}
```

En glömd **break** kan ge svårhittad bugg...

Javas switch-sats med glömd break

```java
import java.util.Scanner;

public class SwitchForgotBreak {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv grönsak:");
        String g = scan.next();
        switch (g) {
        case "gurka":
            System.out.println("gott!");
        case "tomat":
            System.out.println("mycket gott!");
            break;
        case "broccoli":
            System.out.println("ganska gott...");
            break;
        default:
            System.out.println("mindre gott...");
            break;
        }
    }
}
```

Javas switch-sats med glömd break

```java
import java.util.Scanner;

public class SwitchForgotBreak {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv grönsak:");
        String g = scan.next();
        switch (g) {
        case "gurka":
            System.out.println("gott!");
        case "tomat":
            System.out.println("mycket gott!");
            break;
        case "broccoli":
            System.out.println("ganska gott...");
            break;
        default:
            System.out.println("mindre gott...");
            break;
        }
    }
}
```

```
> java SwitchForgotBreak
Skriv grönsak:
gurka
gott!
mycket gott!
```

Syntax för variabeldeklaration i Scala och Java

Exempel på variabeldeklarationer i

Scala

```
var i1: Int = 0
var i2 = 0
var i3 = (i2: Int) + 0
var p1: Point = new Point(0, 0)
var p2 = new Point(0, 0)
var (x, y) = (0, 0)
val a = 0
final val Constant = 42
```

- i2 härledd typ; går ej i Java 8,9 men finns med `var` i Java 10
- i3 typ varhelst i uttryck; går ej i Java
- (x, y) mönster i init; går ej i Java
- `val` ger ”engångsinit”; ingen exakt motsvarighet i Java men `final` kan ofta användas i stället

Java

```
int i1 = 0;
var i2 = 0; // från Java 10
int i4;
Point p1 = new Point(0, 0);
var p2 = new Point(0, 0);
final int CONSTANT = 42;
```

- i4 ej explicit init; går ej i Scala

For-sats i Scala och Java

Scala

```scala
val s = "Abbasillen"

// Loopa över index framlänges:

for i <- 0 until s.length do
  println(s(i))


// Loopa över index baklänges:

for i <- s.length-1 to 0 by -1 do
  println(s(i))
```

Java

```java
String s = "Abbasillen";

// Loopa över index framlänges:

for (int i = 0; i < s.length(); i++) {
    System.out.println(s.charAt(i));
}

// Loopa över index baklänges:

for (int i = s.length()-1; i >= 0; i--) {
    System.out.println(s.charAt(i));
}
```

I Scala är `s.indices` att föredra!

For-sats i Scala med indices

Scala

```scala
val s = "Abbasillen"

// Loopa över index framlänges:

for i <- s.indices do
  println(s(i))


// Loopa över index baklänges:

for i <- s.indices.reverse do
  println(s(i))
```

Java

```java
String s = "Abbasillen";

// Loopa över index framlänges:

for (int i = 0; i < s.length(); i++) {
    System.out.println(s.charAt(i));
}

// Loopa över index baklänges:

for (int i = s.length()-1; i >= 0; i--) {
    System.out.println(s.charAt(i));
}
```

For-satser och arrayer i Java

En for-sats i Java har följande struktur:

```
for (initialisering; slutvillkor; inkrementering) {
    sats1;
    sats2;
    ...
}
```

En primitiv heltals-array deklareras så här i Java:

```
int[] xs = new int[42];  // rymmer 42 st heltal, init 0:or
int[] ys = {10, 42, -1}; // initera med 3 st heltal
```

Exempel på for-sats: fyll en array med 1:or

```
for (int i = 0; i < xs.length; i++){
  xs[i] = 1;    // indexera med [i]
}
```

Implementation av SEQ-COPY i Java med `for`-sats

```
1  public class SeqCopyForJava {
2
3      public static int[] arrayCopy(int[] xs){
4          int[] result = new int[xs.length];
5          for (int i = 0; i < xs.length; i++){
6              result[i] = xs[i];
7          }
8          return result;
9      }
10
11     public static String test(){
12         int[] xs = {1, 2, 3, 4, 42};
13         int[] ys = arrayCopy(xs);
14         for (int i = 0; i < xs.length; i++){
15             if (xs[i] != ys[i]) {
16                 return "FAILED!";
17             }
18         }
19         return "OK!";
20     }
21
22     public static void main(String[] args) {
23         System.out.println(test());
24     }
25 }
```

Lite syntax och semantik för Java:

- En Java-klass med enbart statiska medlemmar motsvarar ett singelobjekt i Scala.
- Typen kommer **före** namnet.
- Man **måste** skriva `return`.
- Man **måste** ha semikolon efter varje sats.
- Metodnamn **måste** följas av parenteser; om inga parametrar finns används `()`
- En array i Java är inget vanligt objekt, men har ett ”attribut” `length` som ger antal element.
- **Övning**: skriv om med Javas `while`-sats i stället.

Element för element med speciell for-each-sats i Java

Scala

```
val s = "Abbasillen"

// Loopa över alla tecken:

for ch <- s do println(ch)
```

Java

```
String s = "Abbasillen";

// Loopa över alla tecken:

for (char ch: s.toCharArray()) {
  System.out.println(ch);
}
```

Element för element med speciell for-each-sats i Java

Scala

```
val s = "Abbasillen"

// Loopa över alla tecken:

for ch <- s do println(ch)
```

Java

```
String s = "Abbasillen";

// Loopa över alla tecken:

for (char ch: s.toCharArray()) {
  System.out.println(ch);
}
```

`s.foreach(println)` går ej i Java men från Java 8 finns metoden `chars` som ger en `IntStream` och då kan man:

```
str.chars().forEachOrdered(i -> System.out.println((char) i));
```

Typisk utformning av Java-klass

Typisk ”anatomi” hos en Java-klass:

```
public class Klassnamn {
    attribut, normalt privata
    konstruktorer, normalt publika
    metoder: publika getters, och vid förändringsbara objekt även setters
    metoder: privata abstraktioner för internt bruk
    metoder: publika abstraktioner tänkta att användas av klientkoden
}
```

Statiska medlemmar i Java

- Man kan **inte** deklarera explicita singelobjekt i Java och det finns inget nyckelord `object`.
- I stället kan man deklarera **statiska medlemmar** i en klass med Java-nyckelordet `static`.
- Exempel på hur vi ska göra detta inuti en klassen `JPerson`:

```
public static final int ADULT_AGE = 18;
```

- Effekten blir den samma som ett singelobjekt i Scala:
  - Alla statiska medlemmar i en Java-klass allokeras automatiskt och hamnar i en egen singulär ”klassinstans” som existerar oberoende av de dynamiska instanserna.
  - De statiska medlemmarna accessas med punktnotation genom klassnamnet, **utan `new`**:

```
System.out.println(JPerson.ADULT_AGE);
```

Exempel: oföränderlig klass i Scala och Java

```
class Person(val name: String, val age: Int):
  def isAdult = age >= Person.AdultAge

object Person:
  val AdultAge = 18
```

Exempel: oföränderlig klass i Scala och Java

```scala
class Person(val name: String, val age: Int):
  def isAdult = age >= Person.AdultAge

object Person:
  val AdultAge = 18
```

```java
public class JPerson {
    private String name;
    private int age;
    public static final int ADULT_AGE = 18;

    public JPerson(String name, int age) {
      this.name = name;
      this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public boolean isAdult() {
        return age >= ADULT_AGE;
    }
}
```

Lär dig detta mönster för en typisk Java-klass utantill så du snabbt får grejerna på plats!

Exempel: oföränderlig klass i Scala och Java

```
class Person(val name: String, val age: Int):
  def isAdult = age >= Person.AdultAge

object Person:
  val AdultAge = 18
```

```
public class JPerson {
    private String name;
    private int age;
    public static final int ADULT_AGE = 18;

    public JPerson(String name, int age) {
      this.name = name;
      this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public boolean isAdult() {
        return age >= ADULT_AGE;
    }
}
```

Övning:
Gör `Person` + `JPerson` **förändringsbara** så att namnet och åldern går att uppdatera och följande krav uppfylls:

- namnet ska ges vid konstruktion,
- åldern ska initieras till 0 vid konstr.,
- åldern ska aldrig kunna bli negativ.

Lär dig detta mönster för en typisk Java-klass utantill så du snabbt får grejerna på plats!

Exempel: Scala-klassen Complex

```
class Complex(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'
```

Scala: `https://github.com/lunduniversity/introprog/blob/master/compendium/examples/complex7.scala`
Java: `https://github.com/lunduniversity/introprog/blob/master/compendium/examples/JComplex.scala`

Exempel: Motsvarande Java-klassen JComplex

```
1   public class JComplex { // man kan ej deklarera klassparametrar i Java
2       private double re;  // initialiseras i konstruktorn nedan
3       private double im;  // initialiseras i konstruktorn nedan
4       public static char imSymbol = 'i'; // publikt förändringsbart attribut (ovanligt i Java)
5
6       public JComplex(double real, double imag){ // konstruktor, körs vid new
7           re = real;
8           im = imag;
9       }
10
11  // en "getter" som ger attributvärdet, hindrar förändring av re
12      public double getRe(){
13          return re;
14      }
15
16  // ej bruklig formattering i Java, så metoder blir minst 3 rader
17      public double getIm(){ return im; }
18
19      public double getR(){
20          return Math.hypot(re, im);       // Math med stort M i Java
21      }
22
23      public double getFi(){
24          return Math.atan2(re, im);
25      }
26
27  // Javametodnamn får ej ha operatortecken t.ex. +, därav namnet add
28      public JComplex add(JComplex other){
29          return new JComplex(re + other.getRe(), im + other.getIm());
30      }
31
32      @Override public String toString(){
33          return re + " + " + im + imSymbol;
34      }
35  }
```

Exempel: Använda JComplex i Scala-kod

```
$ javac JComplex.java
$ scala
Welcome to Scala 2.12.9 (OpenJDK 64-Bit Server VM, Java 1.8.0_222).
Type in expressions for evaluation. Or try :help.

scala> val jc1 = JComplex(3, 4)
jc1: JComplex = 3.0 + 4.0i

scala> val polarForm = (jc1.getR, jc1.getFi)
polarForm: (Double, Double) = (5.0,0.6435011087932844)

scala> val jc2 = JComplex(1, 2)
jc2: JComplex = 1.0 + 2.0i

scala> jc1 add jc2
res0: JComplex = 4.0 + 6.0i
```

Exempel: Använda JComplex i Java-kod

```
public class JComplexTest {
    public static void main(String[] args){
        JComplex jc1 = new JComplex(3,4);
        String polar = "(" + jc1.getR() + ", " + jc1.getFi() + ")";
        System.out.println("Polär form: " + polar);
        JComplex jc2 = new JComplex(1,2);
        System.out.println(jc1.add(jc2));
    }
}
```

- I Java måste man skriva **`new`**.
- I Java måste man skriva **tomma parentes-par** efter metodnamnet vid **anrop av parameterlösa metoder**.
- **Tupler finns inte** i Java, så det går inte på ett enkelt sätt att skapa par av värden som i Scala; ovan görs polär form till en sträng för utskrift.
- **Operatornotation för metoder finns inte** i Java, så man måste i Java använda punktnotation och skriva: `jc1.add(jc2)`

Exempel: Förändringsbar klass i Scala och Java

```
class MutablePerson(var name: String):
  private var _age = 0

  def age: Int = _age

  def age_=(a: Int): Unit =
    if (a >= 0) _age = a else _age = 0
      // eller hellre kasta undantag?

  def isAdult: Boolean =
    age >= MutablePerson.AdultAge

object MutablePerson:
  val AdultAge = 18
```

Exempel: Förändringsbar klass i Scala och Java

```scala
class MutablePerson(var name: String):
  private var _age = 0

  def age: Int = _age

  def age_=(a: Int): Unit =
    if (a >= 0) _age = a else _age = 0
      // eller hellre kasta undantag?

  def isAdult: Boolean =
    age >= MutablePerson.AdultAge

object MutablePerson:
  val AdultAge = 18
```

```java
public class JMutablePerson {
    private String name;
    private int age = 0;
    public static final int ADULT_AGE = 18;

    public JMutablePerson(String name) {
      this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age >= 0) {
          this.age = age;
        } else {
          this.age = 0;
        }
    }

    public boolean isAdult() {
        return age >= ADULT_AGE;
    }
}
```

Scalas ”case-klass-godis” finns inte i Java

En oföränderlig datatyp implementeras i **Scala** helst som en

Scalas ”case-klass-godis” finns inte i Java

En oföränderlig datatyp implementeras i **Scala** helst som en **`case`**-klass:

```
case class Person(name: String, age: Int):
  def isAdult = age >= Person.AdultAge

object Person:
  val AdultAge = 18
```

Scalas ”case-klass-godis” finns inte i Java

En oföränderlig datatyp implementeras i **Scala** helst som en **`case`**-klass:

```
case class Person(name: String, age: Int):
  def isAdult = age >= Person.AdultAge

object Person:
  val AdultAge = 18
```

En oföränderlig datatyp i **Java** med **motsvarande** funktionalitet kräver egen implementation av dessa metoder:

- en getter för varje attribut
- `equals`
- `hashCode` (förklaras i forts.kurs)
- `apply`
  (men man kallar nog den `create` el. likn.; namnet måste ju skrivas)
- `toString`
- `copy`
  (men det finns ju inte namngivna parametrar och default-argument så denna blir osmidig)
- `unapply`
  (men det finns ju inte mönstermatchning så denna struntar man nog i)

Array i Java

Repetition: Den primitiva typen Array i JVM

- Primitiva arrayer (`Array` i Scala, `[]` i Java) har **fördelar**:[21]
  - Det är den snabbaste indexerbara datastrukturen i JVM: att läsa och uppdatera ett element på en viss plats är mycket effektivt om man vet platsens index.
  - Fungerar lika bra med både primitiva värden och objektreferenser
- ... men också **nackdelar**:
  - Man måste bestämma sig för antalet element som ska allokeras när man gör `new`.
  - Man kan ta i lite extra när man allokerar om man behöver plats för fler senare, men då måste man hålla reda på hur många platser man använder och veta var nästa lediga plats finns.
  - Det är krångligt att stoppa in (eng. *insert*) och ta bort (eng. *delete*) element.
  - Vill man ha fler platser måste man allokera en helt ny, större array och kopiera över alla befintliga element.

[21] stackoverflow.com/questions/2843928/benefits-of-arrays

Syntax för Array i Scala och Java

Scala

```scala
var xs = Array(42, 43, 44)




val n = xs.length

var strings = new Array[String](42)
// eller      Array.ofDim[String](42)
// eller      Array.fill(42)(null: String)

strings(0) = "first"

strings(1) = "second"
```

Java

```java
int[] xs = new int[]{42, 43, 44};

// samma som ovan, men kortare:
int[] xs2 = {42, 43, 44};

int n = xs.length;  // EJ length()

String[] strings = new String[42];




strings[0] = "first";

strings[1] = "second";
```

Exempel: Polygon med primitiv array i Java

```
1   public class Polygon {
2       private Point[] vertices; // array med hörnpunkter
3       private int n;            // antalet hörnpunkter
4
5       /** Skapar en polygon */
6       public Polygon() {
7           vertices = new Point[1];
8           n = 0;
9       }
10
11      ...
```

Polygon med primitiv array i Java: stoppa in sist och vid behov skapa mer plats

Implementera:

```
private void extend()                    // dubbla storleken
public void addVertex(int x, int y)  // lägg till hörnpunkt
```

Polygon med primitiv array i Java: stoppa in sist och vid behov skapa mer plats

Implementera:

```
private void extend()                    // dubbla storleken
public void addVertex(int x, int y)  // lägg till hörnpunkt
```

```
1     private void extend(){
2         Point[] oldVertices = vertices;
3         vertices = new Point[2 * vertices.length]; // skapa dubbel plats
4         for (int i = 0; i < oldVertices.length; i++) {  // kopiera
5             vertices[i] = oldVertices[i];
6         }
7     }
8
9     public void addVertex(int x, int y) {
10        if (n == vertices.length) extend();
11        vertices[n] = new Point(x, y);
12        n++;
13    }
```

Polygon med primitiv array i Java: stoppa in mitt i på angiven plats

Implementera:

```
/** Sätt in hörnpunkt på plats pos */
public void insertVertex(int pos, int x, int y)
```

Polygon med primitiv array i Java: stoppa in mitt i på angiven plats

Implementera:

```
/** Sätt in hörnpunkt på plats pos */
public void insertVertex(int pos, int x, int y)
```

```
1     public void insertVertex(int pos, int x, int y) {
2         if (n == vertices.length) extend();   // utöka vid behov
3         for (int i = n; i > pos; i--) {       // flytta element bakifrån
4             vertices[i] = vertices[i - 1];
5         }
6         vertices[pos] = new Point(x, y);
7         n++;
8     }
```

Scanna filer och strängar med `java.util.Scanner`

- I Scala kan man läsa från fil så här (se quickref sid 3 längst ner):

```
val names = scala.io.Source.fromFile("src/names.txt").getLines.toVector
```

- Klassen `java.util.Scanner` kan också läsa från fil (se Java Snabbref sid 4):

```
def readFromFile(fileName: String): Vector[String] = {
  val file = new java.io.File(fileName)
  val scan = new java.util.Scanner(file)
  val buffer = scala.collection.mutable.ArrayBuffer.empty[String]
  while (scan.hasNext) {
    buffer += scan.next
  }
  scan.close
  buffer.toVector
}
```

- Med `new java.util.Scanner(System.in)` kan man även scanna tangentbordet.
- Med `new java.util.Scanner("hej 42")` kan man även scanna en sträng.
- Scanna `Int` och `Double` med metoderna `nextInt` och `nextDouble`.
  Se doc: docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

Exempel: Scanner

```
scala> val scan = new java.util.Scanner("hej 42 42.0   42 slut")

scala> scan.hasNext
res0: Boolean = true

scala> scan.hasNextInt
res1: Boolean = false

scala> scan.next
res2: String = hej

scala> scan.hasNextInt
res3: Boolean = true

scala> scan.nextInt
res4: Int = 42

scala> while (scan.hasNext) println(scan.next)
42.0
42
slut
```

Använda Java-samlingar i Scala med `CollectionConverters`

Med hjälp av **`import`** `scala.jdk.CollectionConverters.*`
får du smidig **interoperabilitet** med Java och dess standardbibliotek,
speciellt metoderna **`asJava`** och **`asScala`**:

```
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`

Generiska samlingar i Java

- Från och med version 5 av Java (2004) så introducerades **generics** vilket möjliggör skapandet av klasser som kan erbjuda generell behandling av olika typer av objekt.
- Generiska klasser i Java känns igen med syntaxen `Klassnamn<Typ>`, till exempel `ArrayList<Point>`
- Fördjupning: docs.oracle.com/javase/tutorial/extra/generics/intro.html, mer om detta i fördjupningskursen.

Om ArrayList i Java

`java.util.ArrayList` liknar `scala.collection.mutable.ArrayBuffer` som båda har dessa fördelar:

- Lagrar sina element internt i snabbindexerade primitiva arrayer.
- Fungerar för alla typer av objekt.
- Utökar samlingens storlek av sig själv vid behov.

Det finns dock vissa nackdelar med `ArrayList` i Java (som inte gäller för `ArrayBuffer` i Scala):

- Fungerar **inte** rakt av med primitiva typer `int`, `double`, `char`, ... (men det finns sätt komma runt detta, tack vare s.k. wrapper-klasser och autoboxning; mer om detta snart)
- Namnet `ArrayList` är inte helt lyckat, eftersom ordet ”lista” normalt används för länkade snarare än array-liknande strukturer.

Polygon med ArrayList i Java

Klassen `Polygon`, nu med ett attribut av typen `ArrayList<Point>`:

```
public class Polygon {
    private ArrayList<Point> vertices; // lista med hörnpunkter

    /** Skapar en polygon */
    public Polygon() {
        vertices = new ArrayList<Point>();
    }

    ...
```

Det behövs inget attribut n eftersom vi inte själva behöver hålla reda på antalet allokerade platser: allokering, insättning, och utökning av antalet platser sköts helt automatiskt av `ArrayList`-klassen vid behov.

Några viktiga operationer på ArrayList<E>

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html

```
/** Tar reda på elementet på plats pos */
E get(int pos);

/** Lägger in objektet obj sist */
void add(E obj);

/** Lägger in obj på plats pos; efterföljande flyttas */
void add(int pos, E obj);

/** Tar bort elementet på plats pos och returnerar det */
E remove(int pos);

/** Tar reda på antalet element i listan */
int size();
```

Lär dig vad som finns om ArrayList i snabbreferensen för Java
Överkurs för den nyfikne: kolla implementation av ArrayList här:
https://hg.openjdk.org/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/ArrayList.java#l106

Övning ArrayList: new och add

Skriv Java-kod som skapar en lista med element av typen `Point` och lägger in tre punkter i listan med koordinaterna:
(50, 50), (50,10) och (30, 40).

Övning ArrayList: new och add

Skriv Java-kod som skapar en lista med element av typen `Point` och lägger in tre punkter i listan med koordinaterna:
(50, 50), (50,10) och (30, 40).

Lösning:

```
ArrayList<Point> vertices = new ArrayList<Point>();
vertices.add(new Point(50, 50));
vertices.add(new Point(50, 10));
vertices.add(new Point(30, 40));
```

For-each-sats i Java:

- Antag att vi vill gå igenom alla element i en lista.

```
ArrayList<String> words = new ArrayList<String>();
```

- Det finns två olika typer av **for**-satser i Java som kan göra detta:
  - Vanlig **for**-sats:

```
for (int i = 0; i < words.size(); i++) {
    System.out.println(i + ": " + words.get(i));
}
```

  - Så kallad **for-each-sats** med denna syntax:
    `for (Elementtyp element: samling) { ... }`

    Exempel:

```
for (String s: words) {
    System.out.println(s);
}
```

    Men vi får ingen indexvariabel då...

Polygon med ArrayList: metoderna blir enklare

```
    public void addVertex(int x, int y) {
        vertices.add(new Point(x, y));
    }

    public void move(int dx, int dy) {
        for (Point p: vertices){
            p.move(dx, dy);
        }
    }

    public void insertVertex(int pos, int x, int y) {
        vertices.add(pos, new Point(x, y));
    }

    public void removeVertex(int pos) {
        vertices.remove(pos);
    }
```

Se hela lösningen här: compendium/examples/scalajava/list/Polygon.java

Polygon med ArrayList: iterera över alla hörnpunkter i draw med indexering

```
public void draw(SimpleWindow w) {
    if (vertices.size() == 0) {
        return;
    }
    Point start = vertices.get(0);
    w.moveTo(start.getX(), start.getY());
    for (int i = 1; i < vertices.size(); i++) {
        w.lineTo(vertices.get(i).getX(),
                 vertices.get(i).getY());
    }
    w.lineTo(start.getX(), start.getY());
}
```

Övning: Skriv om med for-each-sats.

Polygon med ArrayList: iterera över alla hörnpunkter i draw med foreach-sats

```
    public void draw(SimpleWindow w) {
        if (vertices.size() == 0) {
            return;
        }
        Point start = vertices.get(0);
        w.moveTo(start.getX(), start.getY());
        for (Point p: vertices){
            w.lineTo(p.getX(), p.getY());
        }
        w.lineTo(start.getX(), start.getY());
    }
```

Se hela lösningen här: compendium/examples/scalajava/list/Polygon.java

Övning ArrayList: implementera metoden hasVertex

Skriv kod som implementerar denna metod i klassen `Polygon`:

```
/** Undersöker om polygonen har någon hörnpunkt med koordinaterna x, y. */
public boolean hasVertex(int x, int y) {
    ???
}
```

Lösning ArrayList: implementera metoden hasVertex

```
public boolean hasVertex(int x, int y) {
    for (Point p: vertices) {
        if (p.getX() == x && p.getY() == y) {
            return true;
        }
    }
    return false;
}
```

For-each-sats med array

For-each-sats fungerar även med primitiv array:

```
String[] stringArray = {"hej", "på", "dej"};
for (String s: stringArray) {
    System.out.println(s);
}
```

Autoboxning i Java

Generiska klasser (t.ex. ArrayList) med primitiva typer

Detta går tyvärr **INTE** i Java:

~~`ArrayList<int> list = new ArrayList<int>();`~~

Generiska klasser (t.ex. ArrayList) med primitiva typer

Detta går tyvärr **INTE** i Java:

~~`ArrayList<int> list = new ArrayList<int>();`~~

- Hur gör man om man vill ha heltalselement (eller andra primitiva värden) i en generisk samling?
- Javas lösning på problemet består av två delar:
  - Klasser som packar in primitiva typer, (eng. *wrapper classes*)
  - Speciella regler för implicita konverteringar, s.k. ”auto-boxing” (eng. *Boxing / Unboxing conversions*)

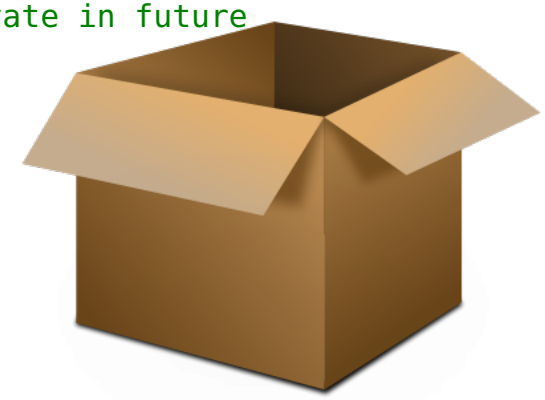
Ofta fungerar det fint, men det finns fallgropar.

(Om du är nyfiken på alla intrikata detaljer, se Java tutorial och Javaspecifikationen.)

Wrapper-klassen `Integer`

En skiss av klassen `Integer`
(ligger i paketet `java.lang` och importeras därmed implicit):

```
public class Integer {
    private int value;

    public static final MIN_VALUE = -2147483648;
    public static final MAX_VALUE = 2147483647;

    public Integer(int value) { // deprecated; will be private in future
        this.value = value;
    }

    public static valueOf(int value) {
      return new Integer(value)
    }

    public int intValue() {
        return value;
    }
    ...
}
```



Javadoc för klasen `Integer` finns här:
`http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html`

Wrapper-klasser i `java.lang`

Primitiv typ	Inpackad typ
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

Övning: primitiva versus inpackade typer

- Deklarera en variabel med namnet `gurka` av den primitiva heltalstypen och initiera den till värdet 42.
- Deklarera en referensvariabel med namnet `tomat` av den inpackade (”wrappade”) heltalstypen och initiera den till värdet 43.
- Rita hur det ser ut i minnet.

Exempel: Lista med heltal utan autoboxning

```
import java.util.ArrayList;
import java.util.Scanner;

public class TestIntegerList {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv heltal med blank emellan. Avsluta med <CTRL+D>");
        while (scan.hasNextInt()) {
            int nbr = scan.nextInt();
            Integer obj = Integer.valueOf(nbr);
            list.add(obj);
        }
        System.out.println("Dina heltal i omvänd ordning:");
        for (int i = list.size() - 1; i >= 0; i--) {
            Integer obj = list.get(i);
            int nbr = obj.intValue();
            System.out.println(nbr);
        }
    }
}
```

Koden finns här: compendium/examples/scalajava/TestIntegerList.java

Specialregler för wrapper-klasser

- Om ett `int`-värde förekommer där det behövs ett `Integer`-objekt, så lägger kompilatorn **automatiskt** ut kod som skapar ett `Integer`-objekt som packar in värdet.
- Om ett `Integer`-objekt förekommer där det behövs ett `int`-värde, lägger kompilatorn **automatiskt** ut kod som anropar metoden `intValue()`.

Samma gäller mellan alla primitiva typer och dess wrapper-klasser:

`boolean`	⇔	`Boolean`
`byte`	⇔	`Byte`
`short`	⇔	`Short`
`char`	⇔	`Character`
`int`	⇔	`Integer`
`long`	⇔	`Long`
`float`	⇔	`Float`
`double`	⇔	`Double`

Exempel: Lista med heltal och autoboxning

```
import java.util.ArrayList;
import java.util.Scanner;

public class TestIntegerListAutoboxing {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv heltal med blank emellan. Avsluta med <CTRL+D>");
        while (scan.hasNextInt()) {
            int nbr = scan.nextInt();
            list.add(nbr);   // motsvarar: list.add(Integer.valueOf(nbr));
        }
        System.out.println("Dina heltal i omvänd ordning:");
        for (int i = list.size() - 1; i >= 0; i--) {
            int nbr = list.get(i);  // motsvarar: int nbr = list.get(i).intValue();
            System.out.println(nbr);
        }
    }
}
```

Koden finns här: scalajava/generics/TestIntegerListAutoboxing.java

Fallgropar vid autoboxning

- Jämförelser med == och `!=`
  compendium/examples/scalajava/generics/TestPitfall1.java
- Kompilatorn hittar inte förväxlad parameterordning, t.ex. `add(pos, item)` i fel ordning: ~~`add(item, pos)`~~
  compendium/examples/scalajava/generics/TestPitfall2.java

Equals i Java

Referenslikhet eller innehållslikhet i Scala och Java

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturlikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.

Referenslikhet eller innehållslikhet i Scala och Java

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturlikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.
- I Scala finns flera metoder som testar likhet:
  - metoden `eq` testar **referenslikhet** och `r1.eq(r2)` ger **`true`** om `r1` och `r2` refererar till **samma** instans.
  - metoden `ne` testar 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 men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturlikhet.

Referenslikhet eller innehållslikhet i Scala och Java

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturlikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.
- I Scala finns flera metoder som testar likhet:
  - metoden `eq` testar **referenslikhet** och `r1.eq(r2)` ger **`true`** om `r1` och `r2` refererar till **samma** instans.
  - metoden `ne` testar 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 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 ==

Referenslikhet eller innehållslikhet i Scala och Java

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturlikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.
- I Scala finns flera metoder som testar likhet:
  - metoden `eq` testar **referenslikhet** och `r1.eq(r2)` ger **`true`** om `r1` och `r2` refererar till **samma** instans.
  - metoden `ne` testar 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 men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturlikhet.
- Scalas **standardbibliotek** och **grundtyperna** `Int`, `String` etc. testar **innehållslikhet** genom metoden ==
- I **Java** är det **annorlunda**: symbolen == är ingen metod i Java utan **specialsyntax** som vid instansjämförelse alltid testar **referenslikhet**, medan metoden `equals` kan överskuggas med valfri likhetstest.

Fallgrop med samlingar: metoden contains kräver implementation av equals

Antag att vi vill implementera `hasVertex()` i klassen `Polygon` genom att använda metoden `contains` på en lista. Hur gör vi då?

Fallgrop med samlingar: metoden contains kräver implementation av equals

Antag att vi vill implementera `hasVertex()` i klassen `Polygon` genom att använda metoden `contains` på en lista. Hur gör vi då?

```
public boolean hasVertex(int x, int y) {
    return vertices.contains(new Point(x, y)); // FUNKAR INTE om ...
    // ... inte Point har en equals som kollar innehållslikhet
}
```

Vi behöver implementera metoden `equals(Object obj)` i klassen `Point` som kollar innehållslikhet och ersätter den `equals` som finns i `Object` som kollar referenslikhet, eftersom metoden `contains` i klassen `ArrayList` anropar `equals` när den letar igenom listan efter lika objekt.
Se exempel här: compendium/examples/scalajava/generics/TestPitfall3.java

Det krävs ofta även att man även ersätter `hashCode`, mer om det i fortsättningskursen.

Fullständigt recept för `equals`

För den nyfikne inför fortsättningskursen efter jul:

Läs om fallgropar för att implementera equals i **Java** här:
www.artima.com/lejava/articles/equality.html

Läs receptet för att implementera equals i **Scala** här:
www.artima.com/pins1ed/object-equality.html#28.4

Villkorsuttryck i Java

Det går att använda villkorsuttryck i Java, men med syntax från språket C:

Scala

```
var r = math.random()
var answer = if r > 0.5 then 42 else 0
```

Java

```
double r = Math.random();
int answer = (r > 0.5) ? 42 : 0;
```

Typtest och typkonvertering

Scala

```scala
var x = "hej"

var isString = x.isInstanceOf[String]

var y = 42

var z = y.asInstanceOf[Double]
```

Java

```java
String x = "hej";

boolean isString = x instanceof String;

int y = 42;

double z = (double) y;
```

Typtest och typkonvertering

Scala

```scala
var x = "hej"

var isString = x.isInstanceOf[String]

var y = 42

var z = y.asInstanceOf[Double]
```

Java

```java
String x = "hej";

boolean isString = x instanceof String;

int y = 42;

double z = (double) y;
```

Detta görs ju i Scala bäst med **`match`** och typmönster!

Regler för överskuggning i Java

http://docs.oracle.com/javase/tutorial/java/IandI/override.html

Fånga undantag i Scala och Java

Typisk skillnad mellan Scala och Java:
konstruktioner som är **uttryck** i Scala är ofta **satser** i Java.

Scala

```scala
val a = try 2 / 0 catch
  case e: ArithmeticException => 0

val b = try 4 / 2 catch
  case e: ArithmeticException => 0
```

Java

```java
int a;
try {
    a = 2 / 0;
} catch (ArithmeticException e) {
    a = 0;
}

int b;
try {
    b = 4 / 2;
} catch (ArithmeticException e) {
    b = 0;
}
```

Gränssnittet `List` i Java

- I Java finns inte **`trait`** och inmixning.
- I stället finns **`interface`** som liknar **`trait`** men är mer begränsad vad gäller vilka medlemmar som får finnas.
- Man kan bara göra **`extends`** på exakt en annan klass, men man kan i Java göra **`implements`** på flera **`interface`**.
- Exempel:

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

- Att implementera ett gränssnitt innebär att uppfylla ett kontrakt som utlovar att vissa speciella metoder finns tillgängliga.
- Gränssninttet `List` uppfylls av en av dess implementationer `ArrayList` på liknande sätt i Scala där gränssnittet `Seq` uppfylls av `Vector` etc.
  `List<String> xs = new ArrayList<String>();`
- Liknande exempel från övningen Hangman:
  `Set<Character> found = new HashSet<Character>();`
- En Scala-trait med enbart abstrakta medlemmar kompileras till ett Java-interface i JVM bytekode.
- Mer om gränssnitt i Java i fördjupningskursen.

Det går inte att skapa generisk Array i Java

- I Java kan man **inte** skapa en primitiv array av godtycklig typ enligt generisk typparameter: ~~`T[] xs = new T[42]`~~
- Man måste istället skapa en array av den mest generella referenstypen:
  `Object[] xs = new Object[42]`
  och sedan typtesta och typkonvertera under körtid; se t.ex. implementationen av `ArrayList` på rad 119:
  http://developer.classpath.org/doc/java/util/ArrayList-source.html

Det går inte att skapa generisk Array i Java

- I Java kan man **inte** skapa en primitiv array av godtycklig typ enligt generisk typparameter: ~~`T[] xs = new T[42]`~~
- Man måste istället skapa en array av den mest generella referenstypen:
  `Object[] xs = new Object[42]`
  och sedan typtesta och typkonvertera under körtid; se t.ex. implementationen av `ArrayList` på rad 119:
  http://developer.classpath.org/doc/java/util/ArrayList-source.html

- Detta går faktiskt att göra i Scala med hjälp av `reflect.ClassTag`

Det går inte att skapa generisk Array i Java

- I Java kan man **inte** skapa en primitiv array av godtycklig typ enligt generisk typparameter: ~~`T[] xs = new T[42]`~~
- Man måste istället skapa en array av den mest generella referenstypen:
  `Object[] xs = new Object[42]`
  och sedan typtesta och typkonvertera under körtid; se t.ex. implementationen av `ArrayList` på rad 119:
  http://developer.classpath.org/doc/java/util/ArrayList-source.html

- Detta går faktiskt att göra i Scala med hjälp av `reflect.ClassTag` så här:

```
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, h
```

Jämföra strängar i Java

- I Java kan man **inte** jämföra strängar med operatorerna <, <=, >, och >=
- Dessutom ger operatorerna == och `!=` *inte* innehålls(o)likhet utan **referens(o)likhet** : (
- För innehållslikhet måste du använda metoderna `equals` och `compareTo`.

Jämföra strängar i Java

- I Java kan man **inte** jämföra strängar med operatorerna <, <=, >, och >=
- Dessutom ger operatorerna == och `!=` *inte* innehålls(o)likhet utan **referens(o)likhet** :(
- För innehållslikhet måste du använda metoderna `equals` och `compareTo`.
- `s1.compareTo(s2)` ger ett heltal som är:
  - `0` om s1 och s2 har samma innehåll
  - **negativt** om s1 < s2 i lexikografisk mening, alltså s1 ska sorteras **före**
  - **positivt** om s1 > s2 i lexikografisk mening, alltså s1 ska sorteras **efter**

Jämföra strängar i Java

- I Java kan man **inte** jämföra strängar med operatorerna <, <=, >, och >=
- Dessutom ger operatorerna == och != *inte* innehålls(o)likhet utan **referens(o)likhet** :(
- För innehållslikhet måste du använda metoderna `equals` och `compareTo`.
- `s1.compareTo(s2)` ger ett heltal som är:
  - `0` om s1 och s2 har samma innehåll
  - **negativt** om s1 < s2 i lexikografisk mening, alltså s1 ska sorteras **före**
  - **positivt** om s1 > s2 i lexikografisk mening, alltså s1 ska sorteras **efter**
- Undersök följande:

```
scala> new String("hej") eq new String("hej") // motsvarar == i Java
scala> "hej".equals("hej")                    // samma som == i Scala
scala> "hej".compareTo("hej")
scala> "hej".compareTo("HEJ")         // alla stora är 'före' alla små
scala> "HEJ".compareTo("hej")
```

- Ibland ger `"hej" == "hej"` förvånande nog värdet **`true`** i Java. Detta beror på en minnesoptimering som kallas stränginternalisering (eng. *string interning*) som ofta sker, men inte alltid. Det är därför en svårhittad bug!

docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareTo-java.lang.String-

Jämföra strängar i Java: exempel

Vad skriver detta Java-program ut?

```
public class StringEqTest {
    public static void main(String[] args){
        boolean eqTest1 =
          (new String("hej")) == (new String("hej"));
        boolean eqTest2 =
          (new String("hej")).equals(new String("hej"));
        int eqTest3 =
          (new String("hej")).compareTo(new String("hej"));
        System.out.println(eqTest1);
        System.out.println(eqTest2);
        System.out.println(eqTest3);
    }
}
```

Jämföra strängar i Java: exempel

Vad skriver detta Java-program ut?

```java
public class StringEqTest {
    public static void main(String[] args){
        boolean eqTest1 =
          (new String("hej")) == (new String("hej"));
        boolean eqTest2 =
          (new String("hej")).equals(new String("hej"));
        int eqTest3 =
          (new String("hej")).compareTo(new String("hej"));
        System.out.println(eqTest1);
        System.out.println(eqTest2);
        System.out.println(eqTest3);
    }
}
```

```
$ javac StringEqTest.java
$ java StringEqTest
false
true
0
```