Programmering, grundkurs pgk

Föreläsningsanteckningar pgk (EDAA45)

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

1 Introduktion

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

Om kursen

Nytt för i år 2016

- **Scala** införs som förstaspråk på Datateknikprogrammet.
- Den **största förnyelsen** av den inledande programmeringskursen sedan vi införde **Java 1997**.
  - Nya föreläsningar
  - Nya övningar
  - Nya laborationer
  - Nya skrivningar
- Allt kursmaterial är **öppen källkod**.
- **Studentermedverkan** i kursutvecklingen.

www.lth.se/nyheter-och-press/nyheter/visa-nyhet/article/scala-blir-foerstaspraak-paa-datateknikprogrammet/

Veckoöversikt

W	*Modul*	*Övn*	*Lab*
W01	Introduktion	expressions	kojo
W02	Kodstrukturer	programs	–
W03	Funktioner, objekt	functions	blockmole
W04	Datastrukturer	data	pirates
W05	Sekvensalgoritmer	sequences	shuffle
W06	Klasser	classes	turtlegraphics
W07	Arv	traits	turtlerace-team
KS	KONTROLLSKRIVN.	–	–
W08	Mönster, undantag	matching	chords-team
W09	Matriser, typparametrar	matrices	maze
W10	Sökning, sortering	sorting	survey
W11	Scala och Java	scalajava	Ithopoly-team
W12	Trådar, webb	threads	life
W13	Design, api	Uppsamling	Projekt
W14	Tentaträning	Extenta	–
T	TENTAMEN	–	–

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**
- Programspråken **Scala** och **Java**
- Utvecklingsverktyg (editor, kompilator, utvecklingsmiljö)
- Implementera, testa, felsöka

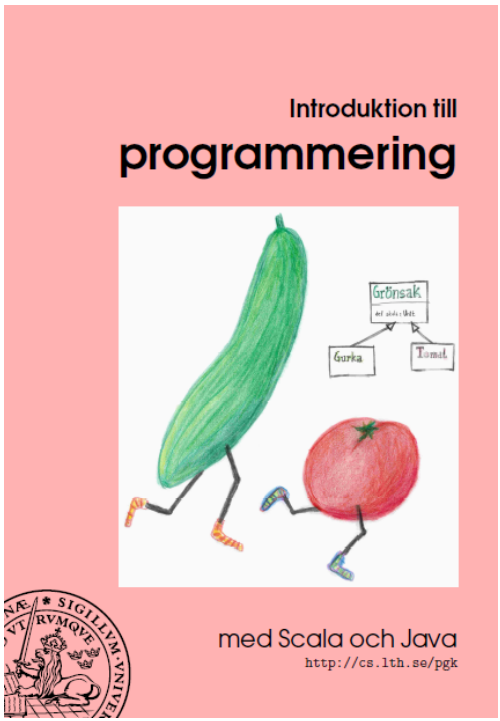
Varför Scala + Java som förstaspråk?

- Varför Scala?
  - Enkel och enhetlig syntax => lätt att skriva
  - Enkel och enhetlig semantik => lätt att fatta
  - Kombinerar flera angreppsätt => lätt att visa olika lösningar
  - Statisk typning + typhärledning => färre buggar + koncis kod
  - Scala Read-Evaluate-Print-Loop => lätt att experimentera
- Varför Java?
  - Det mest spridda språket
  - Massor av fritt tillgängliga kodbibliotek
  - Kompabilitet: fungerar på många platformar
  - Effektivitet: avancerad & mogen teknik ger snabba program
- Java och Scala fungerar utmärkt tillsammans
- Illustrera likheter och skillnader mellan olika språk
  => Djupare lärande

Hur lär du dig?

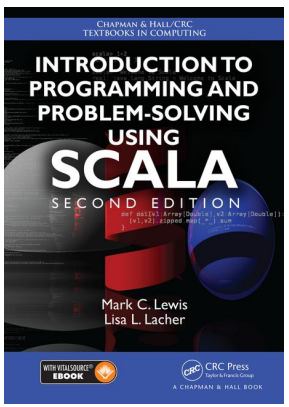
- Genom praktiskt **eget arbete**: **Lära genom att göra!**
  - Övningar: applicera koncept på olika sätt
  - Laborationer: kombinera flera koncept till en helhet
- Genom studier av kursens teori: **Skapa förståelse!**
- Genom samarbete med dina kurskamrater: **Gå djupare!**

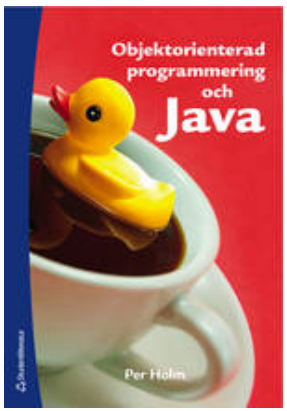
Kurslitteratur



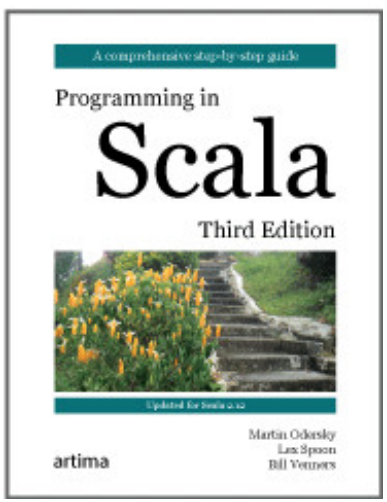
- **Kompendium** med övningar & laborationer, trycks & säljs av inst. på beställning
- Föreläsningsbilder
- Nätresurser enl. länkar

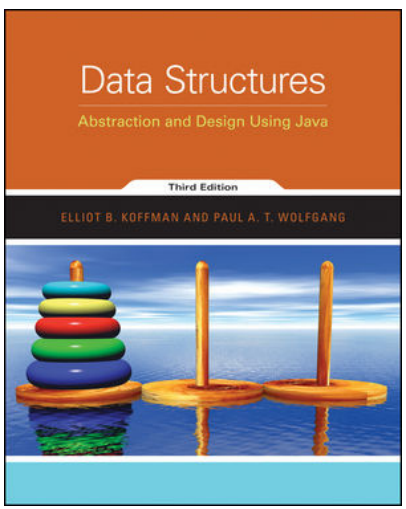
Bra, men ej nödvändig, **bredvidläsning**:
– för **nybörjare**:





– för de som **redan kodat** en del:





Beställning av kompendium och snabbreferens

- **Kompendiet** finns i pdf för fri nedladdning enl. CC-BY-SA, men det **rekommenderas starkt** att du köper den tryckta bokversionen.
- Det är mycket lättare 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.
- Kompendiet och snabbreferens trycks här i E-huset och säljs av institutionen till **självkostnadspris**.
- Pris för kompendium **beror på hur många som beställer**.
- Snabbreferens kostar 10 kr.
- Kryssa i **BOK** på listan som snart skickas runt – tryckning enligt denna beställning.
- Du betalar **kontant** med **jämna pengar** på cs expedition, våning 2.

Föreläsningsanteckningar

- Föreläsningbilder utvecklas under kursens gång.
- Alla bilder läggs ut här: github.com/lunduniversity/introprog/tree/master/slides och uppdateras kontinuerligt allt eftersom de utvecklas.
- Förslag på innehåll välkomna!

Personal

Kursansvarig:

Björn Regnell, bjorn.regnell@cs.lth.se

Kurssekreterare:

Lena Ohlsson
Exp.tid 09.30 – 11.30 samt 12.45 – 13.30

Handledare:

Doktorander:
MSc. Gustav Cedersjö, Tekn. Lic. Maj Stenmark
Teknologer:
Anders Buhl, Anna Palmqvist Sjövall, Anton Andersson, Cecilia Lindskog, Emil Wihlander, Erik Bjäreholt, Erik Grampp, Filip Stjernström, Fredrik Danebjer, Henrik Olsson, Jakob Hök, Jonas Danebjer, Måns Magnusson, Oscar Sigurdsson, Oskar Berg, Oskar Widmark, Sebastian Hegardt, Stefan Jonsson, Tom Postema, Valthor Halldorsson

Kursmoment — varför?

- **Föreläsningar**: skapa översikt, ge struktur, förklara teori, svara på frågor, motivera varför.
- **Övningar**: bearbeta teorins 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 tentan.
- **Individuell projektuppgift**: **obligatorisk**, du visar att du kan skapa ett större program självständigt; redovisas för handledare.
- **Tentamen**: **obligatorisk**, skriftlig, enda hjälpmedel: snabbreferensen. `http://cs.lth.se/pgk/quickref`

Detta är bara början...

Exempel på efterföljande kurser som bygger vidare på denna:

- **Årskurs 1**
  - Programmeringsteknik – fördjupningskurs
  - Utvärdering av programvarusystem
  - Diskreta strukturer
- **Årskurs 2**
  - Objektorienterad modellering och design
  - Programvaruutveckling i grupp
  - Algoritmer, datastrukturer och komplexitet
  - Funktionsprogrammering

Registrering

- Fyll i listan **REGISTRERING EDAA45** som skickas runt.
- Kryssa i kolumnen **ÅBEROPAR PLATS** om vill gå kursen[1][2]
- Kryssa i kolumnen **BESTÄLLER BOK**
- Kryssa i kolumnen **KAN VARA KURSOMBUD** om du kan tänka dig att vara kursombud under kursens gång:
  - Alla LTH-kurser ska utvärderas under kursens gång och efter kursens slut.
  - Till det behövs kursombud – ungefär **2 D-are** och **2 W-are**.
  - Ni kommer att bli kontaktade av studierådet.

[1] D1:a som redan gått motsvarande högskolekurs? Uppsök studievägledningen

[2] D2:a eller äldre som redan påbörjad EDA016/EDA011/EDA017 el likn.? Övergångsregler: Alla labbar gk: tenta EDA011/017; annars kom och prata på rasten

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

Förkunskapsenkät

- Om du inte redan gjort det fyll i förkunskapsenkäten **snarast**: `http://cs.lth.se/pgk/survey`
- Dina svar behandlas internt och all redovisad statistik anonymiseras.
- Enkäten ligger till grund för randomiserad gruppindelning i samarbetsgrupper, så att det blir en spridning av förkunskaper inom gruppen.
- Gruppindelnig publiceras här: `http://cs.lth.se/pgk/grupper/`

Samarbetgrupper

- Ni delas in i **samarbetsgrupper** om ca 5 personer baserat på förkunskapsenkäten, så att olika förkunskapsnivåer sammanförs
- Några av laborationerna är mer omfattande **grupplabbar** och kommer att göras i samarbetsgrupperna
- Kontrollskrivningen i halvtid kan ge **samarbetsbonus** (max 5p) som adderas till ordinarie 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: **Bra 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 vi **når mycket längre**
- 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** 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.
Ta med samarbetskontraktet och visa för handledare på labb 1.

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

Bestraffa inte frågor!

- 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 genom **bra samarbete**. Samarbete gör att man lär sig bättre, men man lär sig inte av att bara kopiera andras lösningar. **Plagiering är förbjuden** och kan medföra **disciplinärende och avstängning**.

En typisk kursvecka

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

Se detaljerna och undantagen i schemat: cs.lth.se/pgk/schema

Laborationer

- **Programmering lär man sig bäst genom att programmera...**
- Labbarna är **individuella** (utom 3) och **obligatoriska**
- Gör **övningarna** och **labbförberedelserna** noga **innan** själva labben – detta är ofta helt nödvändigt för att du ska hinna klart. Dina labbförberedelserna 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 din närvaro**, och fortsätt på resurstid och ev. uppsamlingstider.
- Läs noga kapitel noll ”**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. **Gruppindelning** meddelas på hemsidan senast onsdag morgon.

Resurstider

- På resurstiderna får du **hjälp** med **övningar** och labb**förberedelser**.
- Kom till minst en resurstid per vecka, se TimeEdit.
- Handledare gör ibland **genomgångar** för alla under resurstiderna. Tipsa om handledare om vad du finner svårt!
- Du får i mån av plats gå på flera resurstider per vecka. Om det blir fullt i ett rum prioriteras schemagrupper för att minimera krockar:

Tid Lp1	Sal	Grupper med prio
Ons 10-12 v1-7	Falk	09
Ons 10-12 v1-7	Val	10
Ons 13-15 v1-7	Falk	03
Ons 13-15 v1-7	Val	04
Ons 15-17 v1-7	Falk	11
Ons 15-17 v1-7	Val	12
Tor 10-12 v1-7	Falk	01
Tor 10-12 v1-7	Val	02
Tor 13-15 v1-7	Falk	05
Tor 13-15 v1-7	Val	06
Tor 15-17 v1-7	Falk	07
Tor 15-17 v1-7	Val	08

Att lära denna läsvecka w01

Att lära denna läsvecka w01

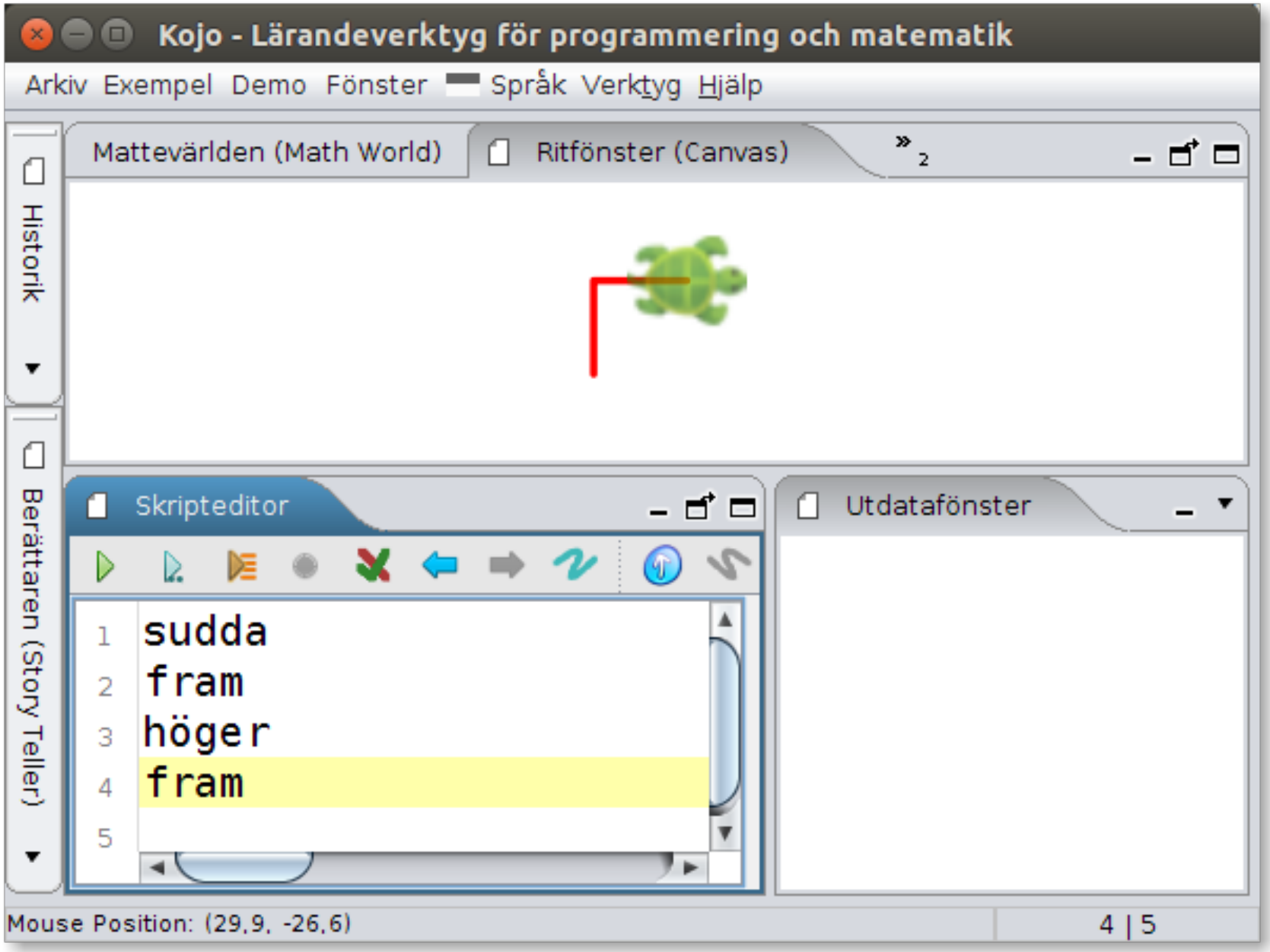
Modul **Introduktion**: Övn **expressions** → Labb **kojo**

- ☐ sekvens
- ☐ alternativ
- ☐ repetition
- ☐ abstraktion
- ☐ programmeringsspråk
- ☐ programmer-ingsparadigmer
- ☐ editera-kompilera-exekvera
- ☐ datorns delar
- ☐ virtuell maskin
- ☐ REPL
- ☐ literal
- ☐ värde
- ☐ uttryck
- ☐ identifierare
- ☐ variabel
- ☐ typ
- ☐ tilldelning
- ☐ namn
- ☐ val
- ☐ var
- ☐ def
- ☐ inbyggda grundtyper
- ☐ Int
- ☐ Long
- ☐ Short
- ☐ Double
- ☐ Float
- ☐ Byte
- ☐ Char
- ☐ String
- ☐ println
- ☐ typen Unit
- ☐ enhetsvärdet ()
- ☐ stränginterpolatorn s
- ☐ if
- ☐ else
- ☐ true
- ☐ false
- ☐ MinValue
- ☐ MaxValue
- ☐ aritmetik
- ☐ slumptal
- ☐ math.random
- ☐ logiska uttryck
- ☐ de Morgans lagar
- ☐ while-sats
- ☐ for-sats

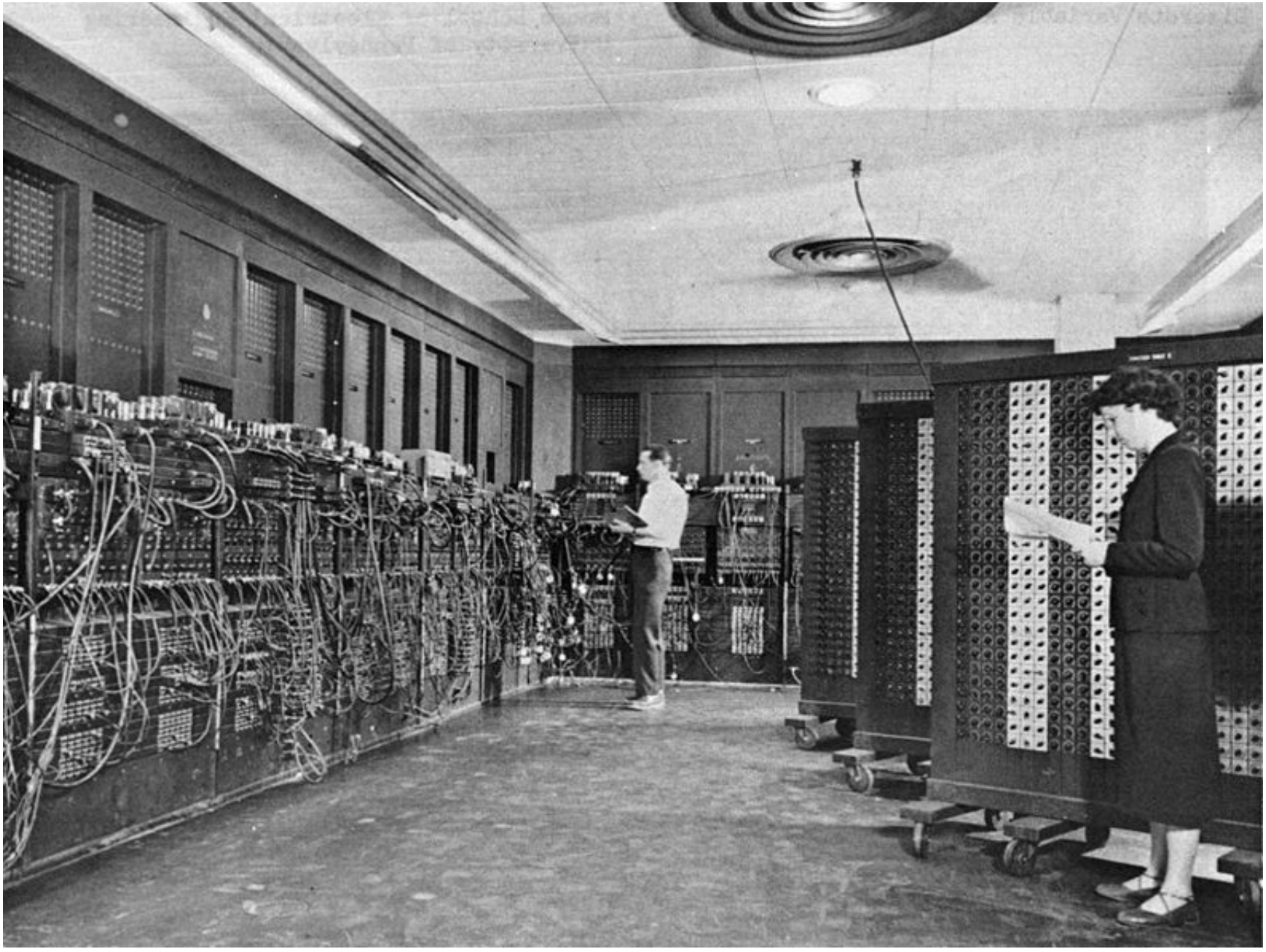
Om programmering

Programming unplugged: Två frivilliga?

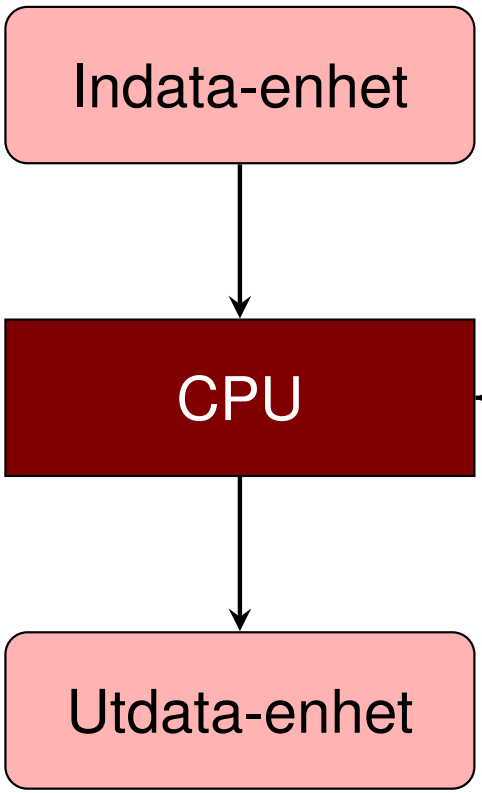
Editera och exekvera ett program



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?

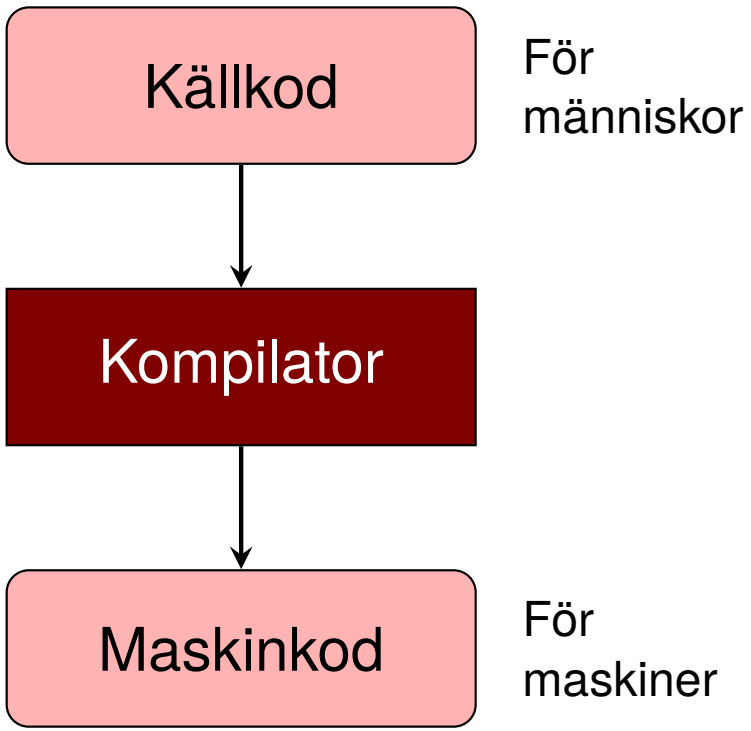
- Programmering innebär att ge instruktioner till en maskin.
- Ett **programmeringsspråk** används av människor för att skriva **källkod** som kan översättas av en **kompilator** till **maskinspråk** som i sin tur **exekveras** av en dator.
- Ada Lovelace skrev det första programmet redan på 1800-talet 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 första kompilatorn 1952.

en.wikipedia.org/wiki/Grace_Hopper

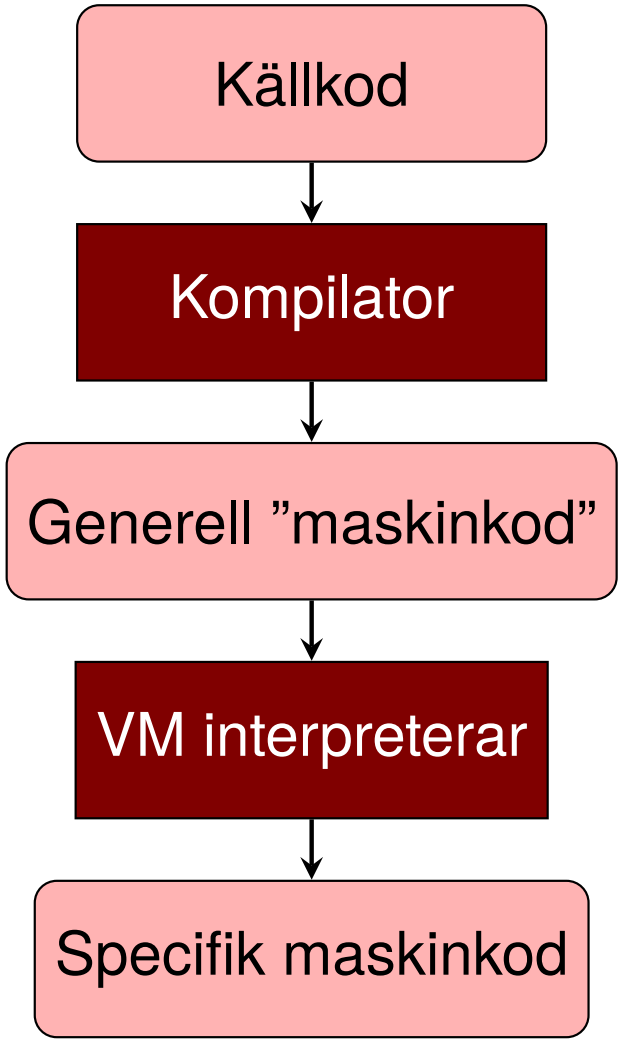


Virtuell maskin (VM) == abstrakt hårdvara

En VM är en ”dator” implementerad i mjukvara som kan tolka en generell ”maskinkod” som **översätts under körning** till den verkliga maskinens kod.

Med en VM blir källkoden **plattformsoberoende** och fungerar på många olika maskiner.

Exempel:
Java Virtual Machine



Vad består ett program av?

- Text som följer entydiga språkregler (gramatik):
  - **Syntax**: textens konkreta utseende
  - **Semantik**: textens betydelse (vad maskinen gör/beräknar)
- **Nyckelord**: ord med speciell betydelse, t.ex. `if`, `else`
- **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 saker händer beroende på uttrycks värde
  - **Repetition**: satser upprepas många gånger
  - **Abstraktion**: nya byggblock skapas för att återanvändas

Exempel på programmeringsspråk

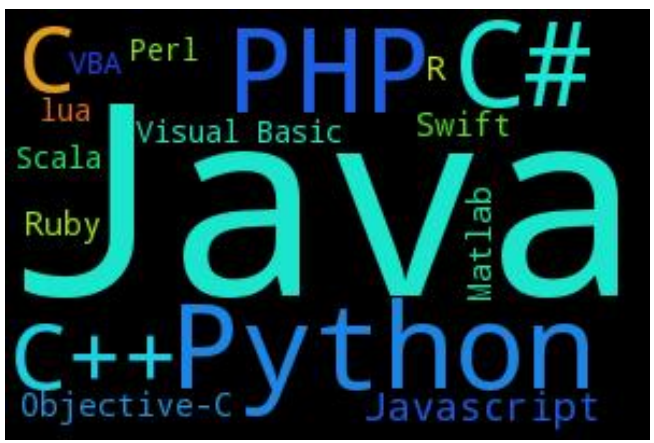
Det finns massor med olika språk och det kommer ständigt nya.

Exempel:

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

Topplistor:

- TIOBE Index
- PYPL Index



Olika programmeringsparadigm

- Det finns många olika programmeringsparadigm (sätt att programmera på), till exempel:
  - **imperativ programmering:** programmet är uppbyggt av sekvenser av olika satser som påverkar systemets tillstånd
  - **objektorienterad programmering:** en sorts imperativ programmering där programmet består av objekt som sammanför data och operationer på dessa data
  - **funktionsprogrammering:** programmet är uppbyggt av samverkande (matematiska) funktioner som undviker föränderlig data och tillståndsändringar
  - **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

Hello world

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

```scala
// this is Scala

object Hello {
  def main(args: Array[String]): Unit = {
    println("Hejsan scala-appen!")
  }
}
```

```java
// this is Java

public class Hi {
    public static void main(String[] args) {
        System.out.println("Hejsan Java-appen!");
    }
}
```

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 handgrep.
- Verktyg vi använder i kursen:
  - Scala **REPL**: från övn 1
  - **Texteditor** för kod, t.ex `gedit` eller `atom`: från övn 2
  - Kompilera med **`scalac`** och **`javac`**: från övn 2
  - Integrerad utvecklingsmiljö (IDE)
    - **Kojo**: från lab 1
    - **Eclipse+ScalaIDE** eller **IntelliJ IDEA** med Scala-plugin: från lab 3 i vecka 4
  - **jar** för att packa ihop och distribuera klassfiler
  - **javadoc** och **scaladoc** för dokumentation av kodbibliotek
- Andra verktyg som är bra att lära sig:
  - git för versionshantering
  - GitHub för kodlagring – men **inte** av lösningar till labbar!

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 bristen blir bara värre och värre...
  CS 2016-08-23
- Störst brist är det på **kvinnliga** utvecklare:
  DN 2015-04-02
- Global kompetensmarknad
  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.
- **Livslångt lärande:** Lär nytt och dela med dig hela tiden. Exempel på pedagogisk utmaning: hjälp andra förstå och använda ditt API $\Longrightarrow$ **Samarbetskultur**

De enklaste beståndsdelarna: litteraler, uttryck, variabler

Literaler

- Literaler 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 **all** typinformation måste finnas redan 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 (booelsk typ)	truth value type	`Boolean`

Grundtypernas implementation i JVM

Grundtyp i **Scala**	Antal bitar	Omfång minsta/största värde	**primitiv typ** i **Java** & **JVM**
`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`

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 blir en helhet.
- Delar i ett uttryck kan t.ex. vara:
  literaler (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 & Java.
- **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 (men inte Java) 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)
res1: Int = 8
```

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

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

- **Moduloräkning** med restoperatorn %

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

Flyttalsaritmetik

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

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

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

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

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

```
1 scala> 0.0000000000001
2 res6: Double = 1.0E-13
3
4 scala> 1E10 + 0.0000000000001
5 res7: Double = 1.0E10
```

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
```

Funktioner kan ha parametrar

- I en parameterlista inom parenteser kan en eller flera **parametrar** till funktionen anges.
- Exempel på deklaration av funktion med en parameter:

```
def tomatvikt(x: Int) = 42 + x
```

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

```
def tomatvikt(x: Int): Int = 42 + 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.

Färdiga matte-funktioner i paketet `scala.math`

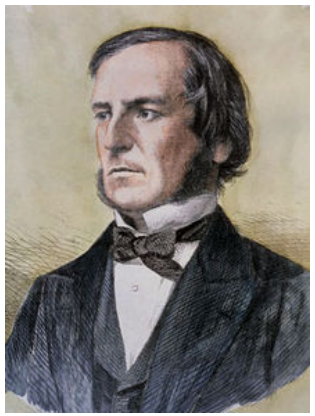
- I paketet **`scala.math`** finns många användbara funktioner: t.ex. `math.random` ger slumptal mellan `0.0` och `0.9999999999999999`

```
scala> val x = math.random
x: Double = 0.2774919174988963 5

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

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

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 all 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 inom parentes och två grenar.

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

- Den ena grenen evalueras om uttrycket är **`true`**
- Den andra **`else`**-grenen evalueras om uttrycket är **`false`**

```
scala> slumpgrönsak
res13: String = gurka

scala> slumpgrönsak
res14: String = gurka

scala> slumpgrönsak
res15: String = tomat
```

Satser

Tilldelningssatser

- En variabeldeklaration medför att **plats i datorns minne reserveras** så att värden av den typ som variabeln kan referera till får plats där.

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 applicera en **tilldelningsoperator** på variabeln själv, 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
  ```

- Ovan expanderas av kompilatorn till `x = x + 1`

Exempel på förkortade tilldelningssatser

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

scala> x *= 2

scala> x
res0: Int = 84


scala> x /= 3

scala> x
res2: 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

- Variabler som **förändras** över tid kan vara svåra att resonera kring.
- Många buggar beror på att variabler förändras på felaktiga och oanade sätt.
- **Föränderliga** värden blir speciellt svåra i kod som körs jämlöpande (parallelt).
- I ”verklig” s.k. **produktionskod** används därför `val` överallt där det går och `var` bara om det **verkligen** behövs.

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) 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) println("gurka")
```

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

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

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!")
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.
- Funktioner kan också ha sidoeffekter. De kallas då **oäkta** funktioner.

Abstraktion: Problemlösning genom nedbrytning i enkla funktioner och procedurer som 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.

Om veckans övning: `expressions`

- Förstå vad som händer när satser exekveras och uttryck evalueras.
- Förstå sekvens, alternativ och repetition.
- Känna till literalerna 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.

Om veckans labb: `kojo`

- Kunna 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 tillämpa principerna sekvens, alternativ, repetition, och abstraktion i enkla algoritmer.
- Kunna formatera egna program så att de blir lätta att läsa och förstå.
- Kunna förklara vad en variabel är och kunna skriva deklarationer och göra tilldelningar.
- Kunna genomföra upprepade varv i cykeln *editera-exekvera-felsöka/förbättra* för att successivt bygga upp allt mer utvecklade program.

2 Kodstruktur

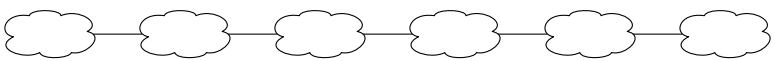
- Datastrukturer och kontrollstrukturer
- Huvudprogram med `main` i Scala och Java
- Algoritmer: stegvisa lösningar
- Funktioner skapar struktur
- Katalogstruktur för kodfiler med paket
- Dokumentation
- Att göra denna vecka

Datastrukturer och kontrollstrukturer

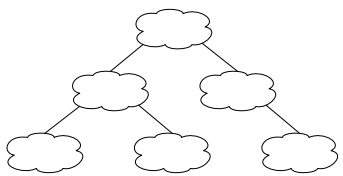
Vad är en datastruktur?

- 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:

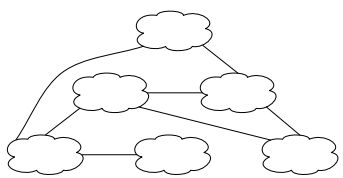
Lista



Träd



Graf



Mer om listor & träd fördjupningskursen. Mer om träd, grafer i Diskreta strukturer.

Vad är en vektor?

En **vektor**[4] (eng. *vector, array*) är en **samling** som är **snabb** att **indexera** i. Åtkomst av element sker med `apply(platsnummer)`:

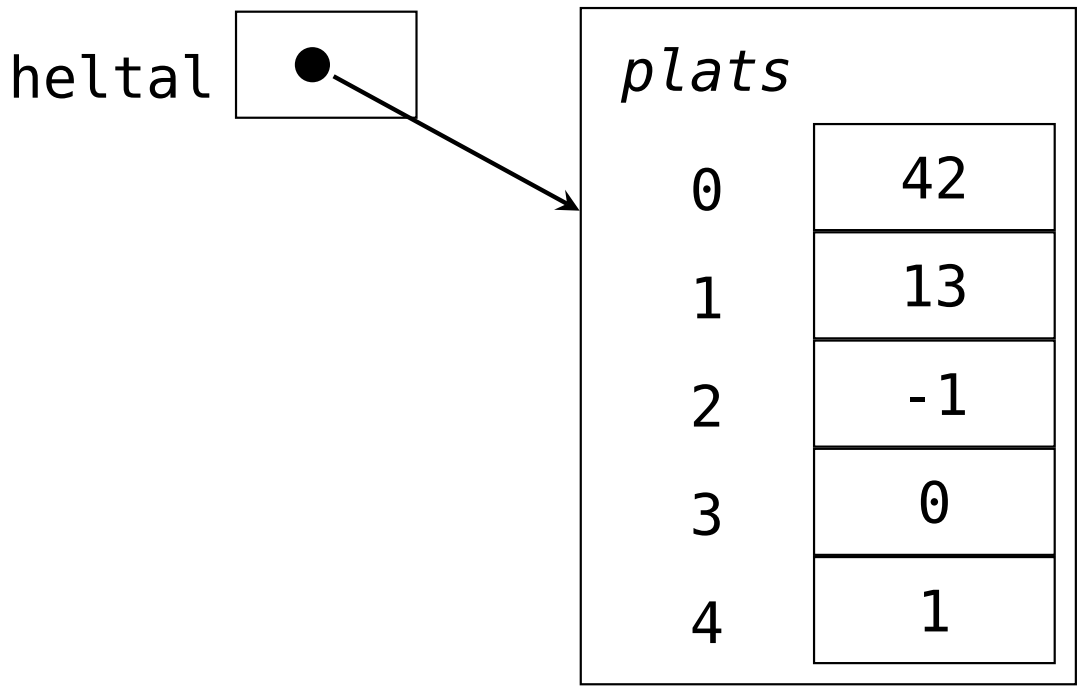
```
1  scala> val heltal = Vector(42, 13, -1, 0 , 1)
2  heltal: scala.collection.immutable.Vector[Int] = Vector(42, 13, -1, 0, 1)
3
4  scala> heltal.apply(0)
5  res0: Int = 42
6
7  scala> heltal(1)    // man kan skippa .apply
8  res1: Int = 13
9
10 scala> heltal(5)
11 java.lang.IndexOutOfBoundsException: 5
12   at scala.collection.immutable.Vector.checkRangeConvert(Vector.scala:132)
```

Utelämnar du `.apply` så gör kompilatorn anrop av `apply` ändå om det går.

[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)
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.

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

Vad är en kontrollstruktur?

- En **kontrollstruktur** påverkar **sekvensen**.

  Exempel på inbyggda kontrollstrukturer:

  `for`-sats, `while`-sats

- 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
```

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

Loopa genom elementen i en vektor

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

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

Bygga en ny samling från en befintlig med for-uttryck

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

```
for (g <- grönsaker) yield "god " + g
```

```
1  scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")
2
3  scala> for (g <- grönsaker) yield "god " + g
4  res0: scala.collection.immutable.Vector[String] =
5    Vector(god gurka, god tomat, god paprika, god selleri)
6
7  scala> val åsikter = for (g <- grönsaker) yield s"god $g"
8  åsikter: scala.collection.immutable.Vector[String] =
9    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)
res0: scala.collection.immutable.Range =
  Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
    15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
    29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41)
```

- Men alla värden däremellan skapas inte förrän de behövs:

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

Loopa med Range

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

```
scala> for (i <- Range(0, 6)) print(" 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
res1: scala.collection.immutable.Range =
  Range(1, 2, 3, 4, 5, 6)

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

Loopa med Range skapad med `to`

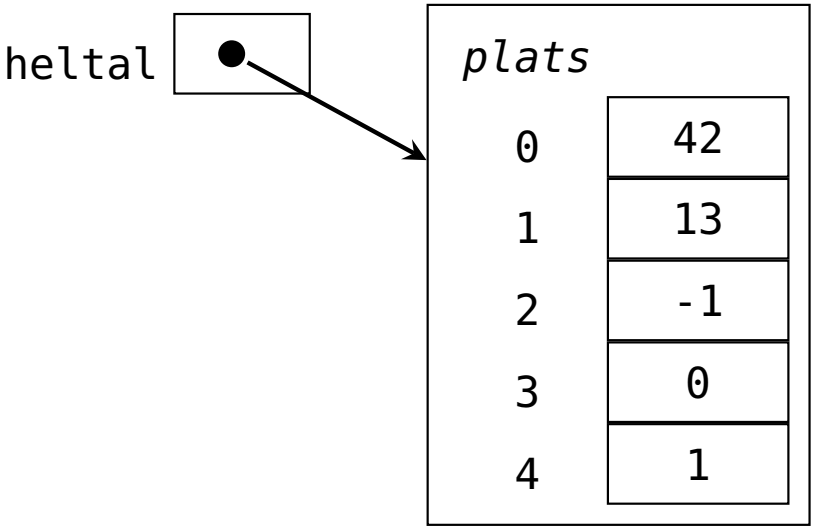
Med `to` efter ett heltal får du en Range till och **med** sista:

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

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

Vad är en `Array` i JVM?

- 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)`
- Båda har en fix storlek efter allokering.

Några viktiga skillnader:

Vector

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

Array

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

Huvudprogram med `main` i Scala och Java

Ett minimalt fristående program i Scala och Java

Nedan Scala-kod skrivs i en editor, spara med valfritt filnamn:

```
// this is Scala

object Hello {
  def main(args: Array[String]): Unit = {
    println("Hejsan scala-appen!")
  }
}
```

Nedan Java-kod skrivs i en editor, filen **måste** heta `Hi.java`

```
// this is Java

public class Hi {
    public static void main(String[] args) {
        System.out.println("Hejsan Java-appen!");
    }
}
```

Loopa genom en samling med en `while`-sats

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

scala> xs.size
res0: Int = 4

scala> var i = 0
i: Int = 0

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

Loopa genom argumenten i ett Scala-huvudprogram

Skriv denna kod och spara i filen `helloargs.scala`

```
$ gedit helloargs.scala
```

```
object HelloScalaArgs {
  def main(args: Array[String]): Unit = {
    var i = 0
    while (i < args.size) {
      println(args(i))
      i = i + 1
    }
  }
}
```

Kompilera och kör:

```
$ scalac helloargs.scala
$ scala HelloScalaArgs hej gurka tomat
hej
gurka
tomat
```

Loopa genom argumenten i ett Java-huvudprogram

```
$ gedit HelloJavaArgs.java
```

```java
// this is 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.scala
$ java HelloJavaArgs hej gurka tomat
hej
gurka
tomat
```

Scala-skript

- Skala-kod kan köras som ett **skript**.[5]
- Ett skript kompileras varje gång innan det körs och maskinkoden sparas inte som vid vanlig kompilering.
- Då behövs ingen `main` och inget **`object`**

```
// spara nedan i filen 'myscript.scala'

println("Hejsan argumnet!")
for (arg <- args) println(arg)
```

```
$ scala myscript.scala
```

[5] Du får prova detta på övningen. Vi kommer mest att köra kompilerat i kursen, då Scala-skript saknar mekanism för inkludering av andra skript. Men det finns ett öppenkällkodsprojekt som löser det: `http://www.lihaoyi.com/Ammonite/`

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
- räkna ut din pensionsprognos
- köra bil
- kolla om highscore i ett spel
- ...

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*
   Skriv ”Försök igen!”
   annars
   Skriv ”Grattis!”

Hittar du buggen?

HIGHSCORE implementerad i Scala

```
import scala.io.StdIn.readLine

object HighScore {
  def main(args: Array[String]): Unit = {
    val points = readLine("Hur mång poäng fick du?").toInt
    val highscore = readLine("Vad var highscore före senaste spelet?").toInt
    val msg = if (points > highscore) "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 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 higscore före senaste spelet?");
        int highscore = scan.nextInt();
        if (points > highscore) {
            System.out.println("GRATTIS!");
        } else {
            System.out.println("Försök igen!");
        }
    }
}
```

Algoritmexempel: N-FAKULTET

Indata : heltalet n
Resultat: utskrift av produkten av de första n heltalen

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

- 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
Resultat: utskrift av 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
skriv ut *min*

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

Funktioner skapar struktur

Mall för funktionsdefinitioner

def funktionsnamn(parameterdeklarationer): returtyp = block

Exempel:

```
def öka(i: Int): Int = { i + 1 }
```

Om ett enda uttryck: behövs inga {}. Returtypen kan härledas.

```
def öka(i: Int) = i + 1
```

Om flera parametrar, separera dem med kommatecken:

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

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

`isHigscore` är en **äkta funktion** som alltid ger samma svar för samma inparametrar och saknar **sidoeffekter**.

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**.

```
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.

```
1  scala> { val lokaltNamn = 42; println(lokaltNamn) }
2  42
3
4  scala> println(lokaltNamn)
5  <console>:12: error: not found: value lokaltNamn
6         println(lokaltNamn)
7
8  scala> { val x = 42; { val x = 76; println(x) }; println(x) }
9  76
10 42
11
12 scala> { val x = 42; { val y = x + 1; println(y) } }
13 43
```

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. **namngiven parameter**

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

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”.

```
1  scala> def hej(x: String): Unit = println(s"Hej på dej $x!")
2  hej: (x: String)Unit
3
4  scala> hej("Herr Gurka")
5  Hej på dej Herr Gurka!
6
7  scala> val x = hej("Fru Tomat")
8  Hej på dej Fru Tomat!
9  x: Unit = ()
```

- 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`

Abstraktion: Problemlösning genom nedbrytning i enkla funktioner och procedurer som 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.

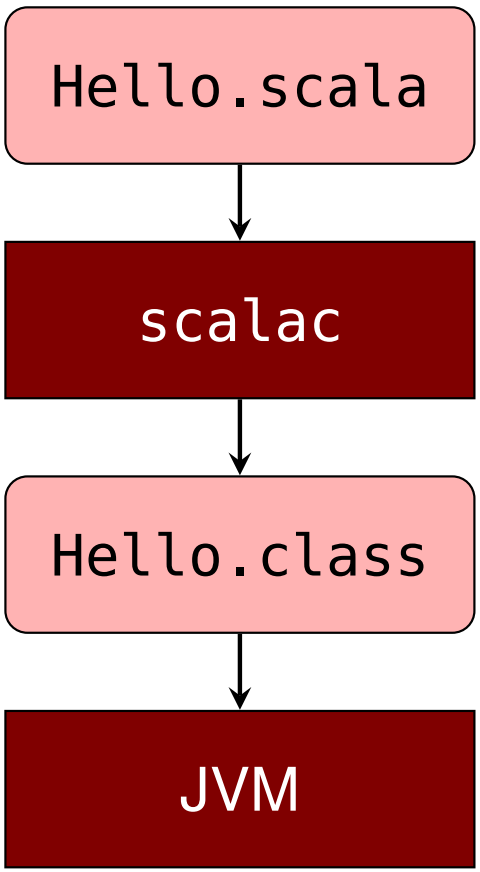
```
// skapa abstraktioner som bygger på varandra

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 och Java:
  - Klasser är ”byggblock” med kod som används för att skapa objekt, innehållande delar som hör ihop.
    Nyckelord: `class` och `object`
  - Metoder är funktioner som finns i klasser/objekt och används för att lösa specifika uppgifter. Nyckelord: `def`
  - Paket används för att organisera kodfiler i en hierarkisk katalogstruktur och skapa namnrymder.
    Nyckelord: `package`

Katalogstruktur för kodfiler med paket

Källkodsfiler och klassfiler



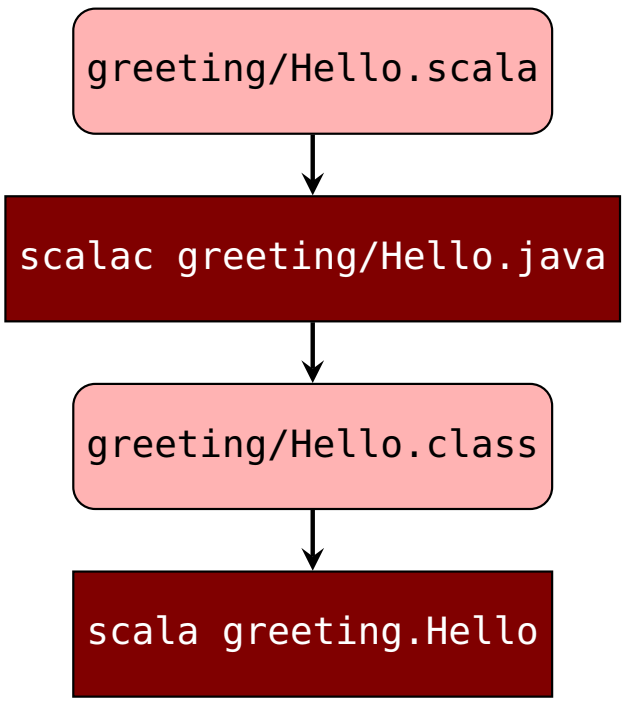
Källkodsfil

`.class`-fil med byte-kod

Java Virtual Machine
Översätter till maskinkod
som passar din specifika CPU
medan programmet kör

Paket

- Paket ger struktur åt kodfilerna. Bra om man har många kodfiler.
- Byte-koden placeras av kompilatorn i kataloger enligt paketstrukturen.




```
package greeting
object Hello { ...
```

Paketens bytekod hamnar i katalog med samma namn som paketnamnet

Katalogstrukturen för källkoden måste i Java motsvara paketstrukturen, men inte i Scala. Dock kräver många IDE att så görs även för Scala.

Import

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

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

En **`import`**-sats...

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

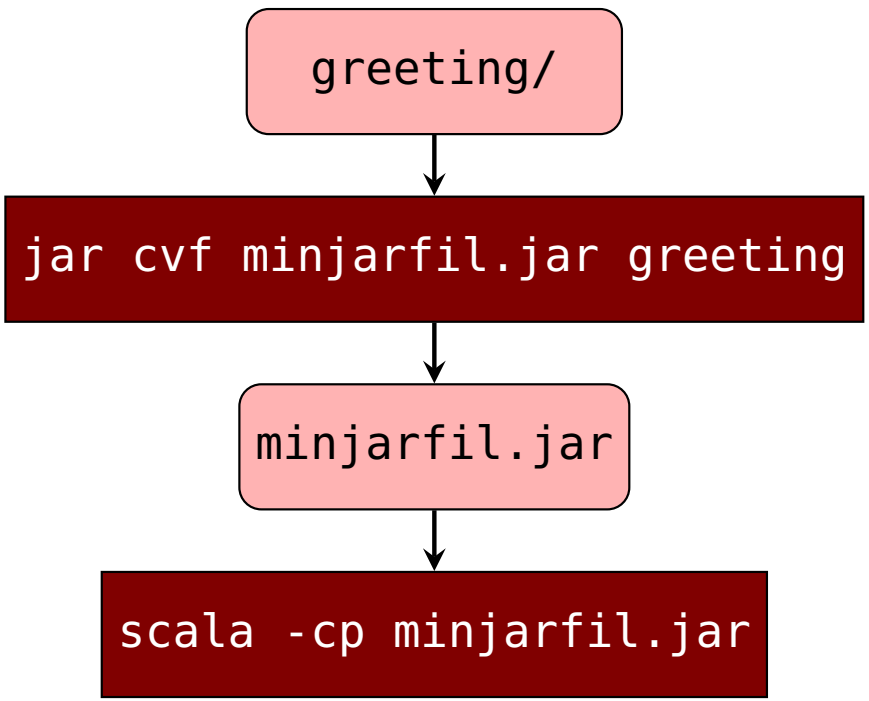
...gör så att kompilatorn ”ser” namnet, och man slipper skriva hela sökvägen till namnet:

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

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

Jar-filer

`jar`-filer liknar `zip`-filer och används för att packa ihop bytekod i en enda fil för enkel distribution och körning.



en katalog med filer

En jar-fil med alla filer inpackade

Lägg jar-filen till ”classpath”

Dokumentation

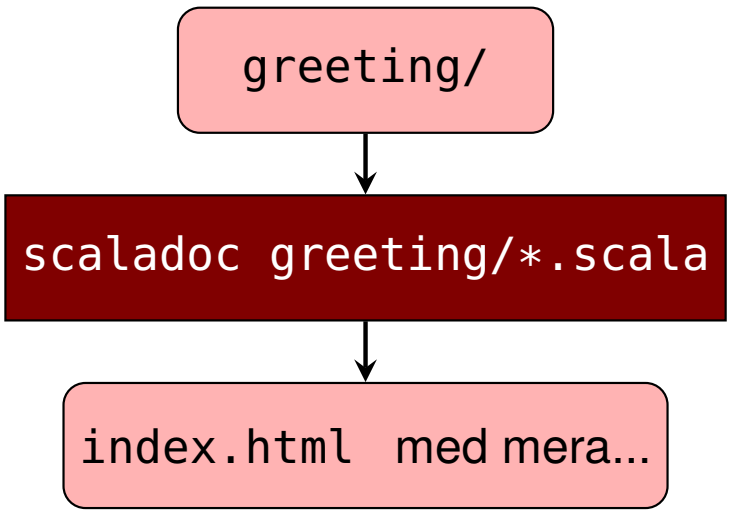
Dokumentation

För att kod ska bli begriplig för människor är det bra att dokumentera vad den gör. Det finns **tre olika sorters kommentarer** som man kan skriva direkt i Scala/Java-koden, **som kompilatorn struntar fullständigt i**:

```
// 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.
 */
```

scaladoc

Programmet `scaladoc`-filer läser källkod och skapar en webbsajt med dokumentation.



en katalog med `.scala`-filer

En webbsajt

Att göra i Vecka 1: Förstå grundläggande kodstrukturer

1. Laborationer är **obligatoriska**.
   Ev. sjukdom måste anmälas **före** via mejl till kursansvarig!
2. Gör övning `programs`
3. OBS! Ingen lab denna vecka w02. Använd tiden att komma ikapp om du ligger efter!
4. Träffas i samarbetsgrupper och hjälp varandra att förstå.
5. Vi har nosat på flera koncept som vi kommer tillbaka till senare: du måste inte fatta alla detaljer redan nu.
6. Om ni inte redan gjort det:
   Visa samarbetskontrakt för handledare på resurstid.
7. **Koda på resurstiderna** och få hjälp och tips!

Veckans övning: `w02-programs`

- Kunna skapa samlingarna Range, Array och Vector med heltals- och strängvärden.
- 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 grundläggande skillnader och likheter mellan samlingarna Range, Array och Vector.
- Förstå skillnaden mellan en for-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 på en indexerbar samling med en `while`-sats.
- Kunna köra igång enkel Scala-kod i REPL, som skript och som applikation.
- Kunna skriva och köra igång ett enkelt Java-program.
- Känna till några grundläggande syntaxskillnader mellan Scala och Java, speciellt variabeldeklarationer och indexering i Array.
- Förstå vad ett block och en lokal variabel är.
- Förstå hur nästlade block påverkar namnsynlighet och namnöverskuggning.
- Förstå kopplingen mellan paketstruktur och kodfilstruktur.
- Kunna skapa en jar-fil.
- Kunna skapa dokumentation med scaladoc.

3 Funktioner, Objekt

- Funktioner
- Objekt
- Funktioner är objekt
- Rekursion
- SimpleWindow
- Veckans övning och laboration

Funktioner

Deklarera funktioner, överlagring

- En parameter, och sedan två parametrar:

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

- Båda funktionerna ovan kan finnas samtidigt! Trots att de har samma namn är de **olika** funktioner; kompilatorn kan skilja dem åt med hjälp av de olika parameterlistorna.
- Detta kallas **överlagring** (eng. *overloading*) av funktioner.

Tom parameterlista och inga parametrar

- Om en funktion deklareras med tom parameterlista ( ) kan den anropas på två sätt: med och utan tomma parenteser.

```
scala> def tomParameterLista() = 42

scala> tomParameterLista()
res2: Int = 42

scala> tomParameterLista
res3: Int = 42
```

Denna flexibilitet är grunden för **enhetlig access**: namnet kan användas enhetligt oavsett om det är en funktion eller en variabel.

- Om parameterlista saknas får man **inte** använda ( ) vid anrop:

```
scala> def ingenParameterLista = 42

scala> ingenParameterLista
res4: Int = 42

scala> ingenParameterLista()
<console>:13: error: Int does not take parameters
```

Funktioner med defaultargument

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

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

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

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

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

- Om argumentet utelämnas och det finns ett defaultargumentet, så är det defaultargumentet som appliceras.

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.

```
1  scala> def namn(förnamn: String,
2                  efternamn: String,
3                  förnamnFörst: Boolean = true,
4                  ledtext: String = ""): String =
5          if (förnamnFörst) s"$ledtext: $förnamn $efternamn"
6          else s"$ledtext: $efternamn, $förnamn"
7
8  scala> namn(ledtext = "Name", efternamn = "Coder", förnamn = "Kim")
9  res0: String = Name: Kim Coder
```

Anropsstacken och objektheapen

Minnet är uppdelat i två delar:

- **Anropsstacken**: På stackminnet 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 heapminnet[7,8] sparas alla objekt (data) som allokeras under körning. Heapen städas vid tillfälle av skräpsamlaren (eng. *garbage collector*), och minne som inte används längre frigörs.

  stackoverflow.com/questions/1565388/increase-heap-size-in-java

[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.

```
1  scala> :paste
2  def f(): Unit = { val n = 5; g(n, 2 * n) }
3  def g(a: Int, b: Int): Unit = { val x = 1; h(x + 1, a + b) }
4  def h(x: Int, y: Int): Unit = { val z = x + y; println(z) }
5
6  scala> f()
```

Stacken

variabel	värde	Vilken aktiveringspost?
n	5	f
a	5	g
b	10	
x	1	
x	2	h
y	15	
z	17	

Lokala funktioner

Med lokala funktioner kan delproblem lösas med nästlade abstraktioner.

```
def gissaTalet(max: Int): Unit = {
  def gissat = io.StdIn.readLine(s"Gissa talet mellan [1, $max]: ").toInt
  val hemlis = (math.random * max + 1).toInt
  def skrivLedtrådOmEjRätt(gissning: Int): Unit =
    if (gissning > hemlis) println(s"$gissning är för stort :(")
    else if (gissning < hemlis) println(s"$gissning är för litet :(")
  def inteRätt(gissning: Int): Boolean = {
    skrivLedtrådOmEjRätt(gissning)
    gissning != hemlis
  }
  def loop: Int = { var i = 1; while(inteRätt(gissat)){ i += 1 }; i }

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

Lokala, nästlade funktionsdeklarationer är tyvärr inte tillåtna i Java.[9]

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

Värdeanrop och namnanrop

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

```
1  scala> def byValue(n: Int): Unit = for (i <- 1 to n) print(" " + n)
2
3  scala> byValue(21 + 21)
4   42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
5
6  scala> byValue({print(" hej"); 21 + 21})
7   hej 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**):

```
1  scala> def byName(n: => Int): Unit = for (i <- 1 to n) print(" " + n)
2
3  scala> byName({print(" hej"); 21 + 21})
4   hej hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej
```

Klammerparenteser vid ensam parameter

Så här har vi sett nyss att man man göra:

```
1  scala> def loop(n: => Int): Unit = for (i <- 1 to n) print(" " + n)
2
3  scala> loop(21 + 21)
4
5  scala> loop({print(" hej"); 21 + 21})
```

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.

Men man kan alltså göra så här också:

```
scala> loop{ 21 + 21 }

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

Uppdelad parameterlista

- 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
```

- Detta kallas även **multipla parameterlistor** (eng. *multiple parameter lists*)

docs.scala-lang.org/style/declarations.html#multiple-parameter-lists

Skapa din egen kontrollstruktur

- Genom att **kombinera uppdelad parameterlista** med **namnanrop** med **klammerparentes vid ensam parameter** kan vi skapa vår egen kontrollstruktur:

```
scala> def upprepa(n: Int)(block: => Unit) = {
         var i = 0
         while (i < n) { block; i += 1 }
       }

scala> upprepa(42){
         if (math.random < 0.5) {
           print(" gurka")
         } else {
           print(" tomat")
         }
       }
 gurka gurka gurka tomat tomat gurka gurka gurka gurka t
```

Funktioner är äkta värden i Scala

- En funktioner är ett äkta värde.
- Vi kan till exempel tilldela en variabel ett funktionsvärde.
- Med hjälp av blank+understreck efter funktionsnamnet får vi funktionen som ett **värde** (inga argument appliceras än):

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

scala> val f = add _

scala> f
f: (Int, Int) => Int = <function2>

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

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

Funktionsvärden kan vara argument

- En funktion kan ha en annan funktion som parameter:

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

scala> def öka(x: Int) = x + 1

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

scala> tvåGånger(42, öka _)
res1: Int = 43

scala> tvåGånger(42, minska _)
res1: Int = 41
```

- Om argumentets funktionstyp **kan härledas** av kompilatorn och **passar** med parametertypen så behövs ej understreck:

```
scala> tvåGånger(42, öka)
res1: Int = 43
```

Applicera funktioner på element i samlingar med `map`

```
1  scala> def öka(x: Int) = x + 1
2
3  scala> def minska(x: Int) = x - 1
4
5  scala> val xs = Vector(1, 2, 3)
6
7  scala> xs.map(öka)
8  res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
9
10 scala> xs.map(minska)
11 res1: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
12
13 scala> xs map öka
14 res2: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
15
16 scala> xs map minska
17 res3: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
```

Funktioner som tar andra funktioner som parametrar kallas **högre ordningens funktioner**.

Anonyma funktioner

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

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

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

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

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

Applicera anonyma funktioner på element i samlingar

- Anonym funktion skapad med funktionsliteral direkt i anropet:

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

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

- Eftersom kompilatorn här kan härleda typerna så behövs de inte:

```
scala> xs.map(x => x - 1)
res1: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)

scala> xs map (x => x - 1)
res2: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
```

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

```
scala> xs.map(_ + 1)
res3: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

Platshållarsyntax för anonyma funktioner

- Understreck i funktionsliteraler 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 funktionsliteral (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 samlingsmetoden `reduceLeft`

Metoden `reduceLeft` applerar en funktion på de två första elementen och tar sedan på resultatet som första argument och nästa element som andra argument och upprepar detta genom hela samlingen.

```
1  scala> def summa(x: Int, y: Int) = x + y
2
3  scala> val xs = Vector(1, 2, 3, 4, 5)
4
5  scala> xs.reduceLeft(summa)
6  res20: Int = 15
7
8  scala> xs.reduceLeft((x, y) => x + y)
9  res21: Int = 15
10
11 scala> xs.reduceLeft(_ + _)
12 res22: Int = 15
13
14 scala> xs.reduceLeft(_ * _)
15 res23: Int = 120
```

Stegade funktioner, ”Curry-funktioner”

Om en funktion har en uppdelad parameterlista 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) _
öka: Int => Int = <function1>

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

scala> Vector(1,2,3).map(add(2))
res1:scala.collection.immutable.Vector[Int]= Vector(3, 4, 5)
```

Översikt begrepp vi gått igenom hittills

- överlagring
- utelämna tom parameterlista (enhetlig access)
- defaultargument
- namngivna argument
- lokala funktioner
- namnanrop (fördröjd evaluering)
- klammerparentes vid ensam paramenter
- uppdelad parameterlista
- egendefinierade kontrollstrukturer
- funktioner som äkta värden
- anonyma funktioner
- stegade funktioner (”Curry-funktioner”)

Begränsningar i Java

- Av alla dessa funktionskoncept...
  - överlagring
  - utelämna tom parameterlista (principen om enhetlig access)
  - defaultargument
  - namngivna argument
  - lokala funktioner
  - namnanrop (fördröjd evaluering)
  - klammerparentes vid ensam paramenter
  - uppdelad parameterlista
  - egendefinierade kontrollstrukturer
  - funktioner som äkta värden
  - anonyma funktioner
  - stegade funktioner (”Curry-funktioner”)
- ...kan man endast göra **överlagring** i Java 7,
- medan även **anonyma funktioner** (”lambda”) går att göra (med vissa begränsningar) i Java 8. en.wikipedia.org/wiki/Anonymous_function#Java_Limitations
- En av de saker jag saknar mest i Java: **lokala funktioner**!

Det är **kombinationen** av alla koncept som **skapar uttryckskraften** i Scala.

Objekt

Objekt som modul

- Ett **`object`** användas ofta för att samla **medlemmar** (eng. *members*) som **hör ihop** och ge dem en egen **namnrymd** (eng. *name space*).
- Medlemmarna kan vara t.ex.:
  - **`val`**
  - **`var`**
  - **`def`**
- Ett sådant objekt kallas även för **modul**.[11]

[11] Även paket som skapas med **`package`** har en egen namnrymd och är därmed också en slags modul. Objekt kan alltså i Scala användas som ett alternativ till paket; en skillnad är att objekt kan ha tillstånd och att objekt inte skapar underkataloger vid kompilering (det finns iofs s.k. **`package object`**) en.wikipedia.org/wiki/Modular_programming

Singelobjekt och metod

Ett Scala-**`object`** är ett s.k. **singelobjekt** (eng. *singleton object*) och finns bara i **en** enda upplaga.
Minne för objektets variabler allokeras första gången objektet refereras.
En funktion som finns i ett objekt kallas en **metod** (eng. *method*).

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

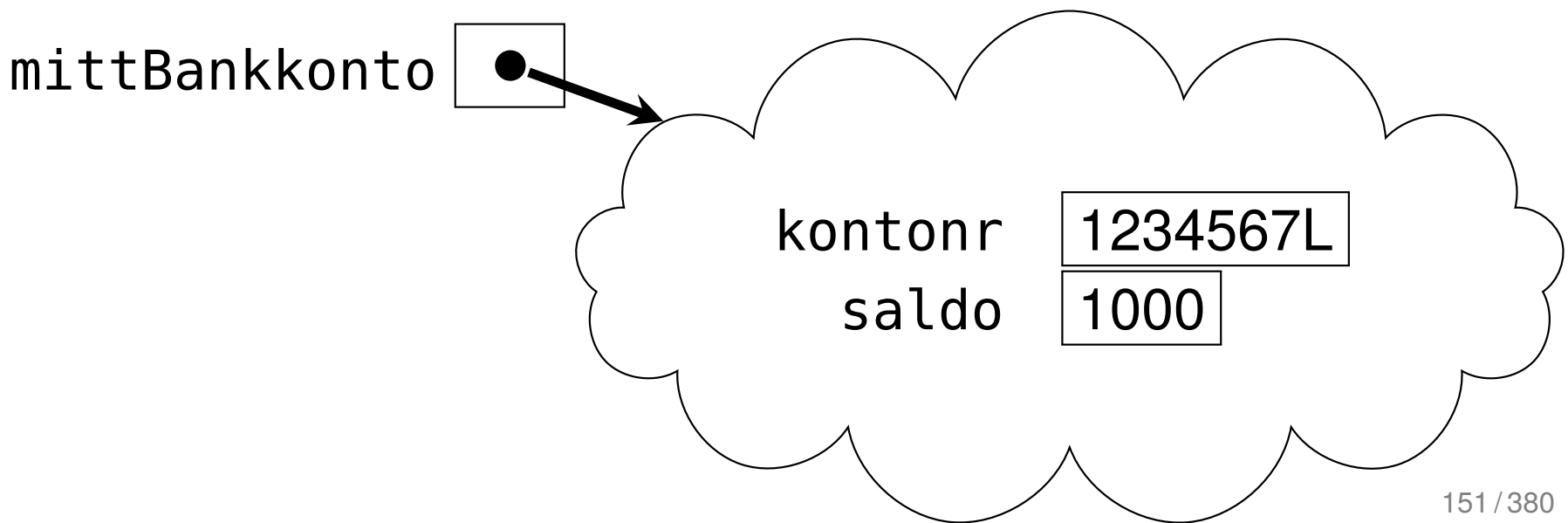
```
scala> mittBankkonto.saldo -= 25000

scala> mittBankkonto.ärSkuldsatt
res0: Boolean = true
```

(Vi ska i nästa vecka se hur man med s.k. klasser kan skapa många upplagor av samma typ av objekt, så att vi kan ha flera olika bankkonto.)

Vad är ett tillstånd?

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

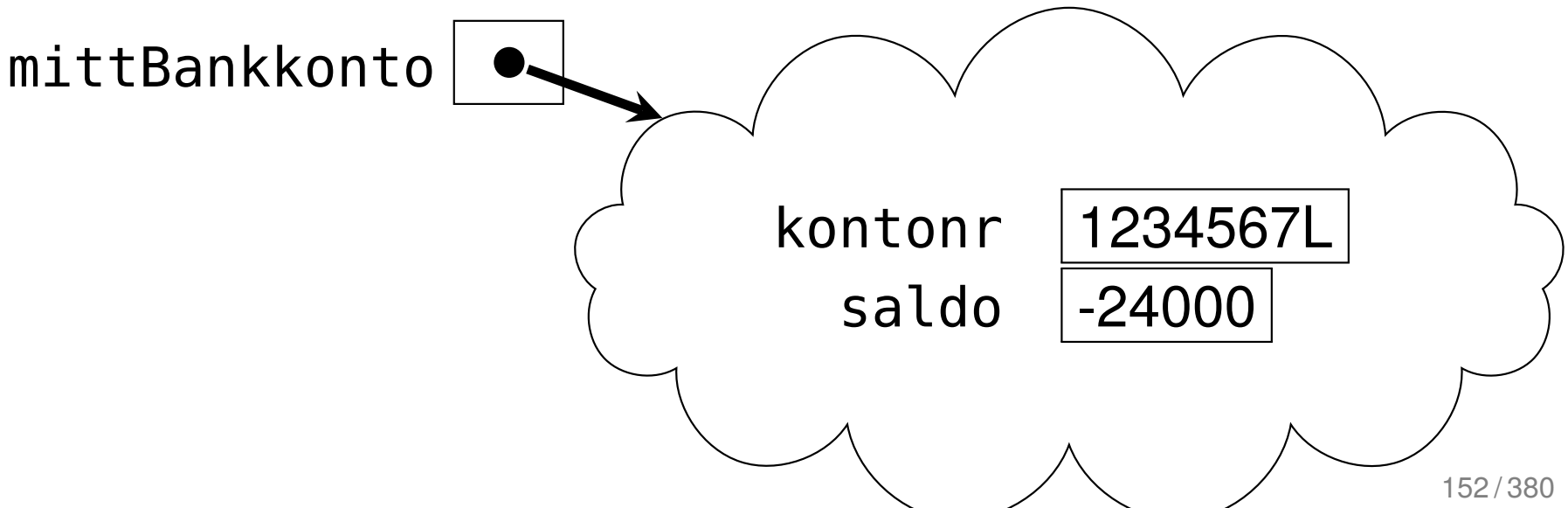
```
object mittBankkonto {
  val kontonr: Long         = 1234567L
  var saldo: Int            = 1000
  def ärSkuldsatt: Boolean = saldo < 0
}
```



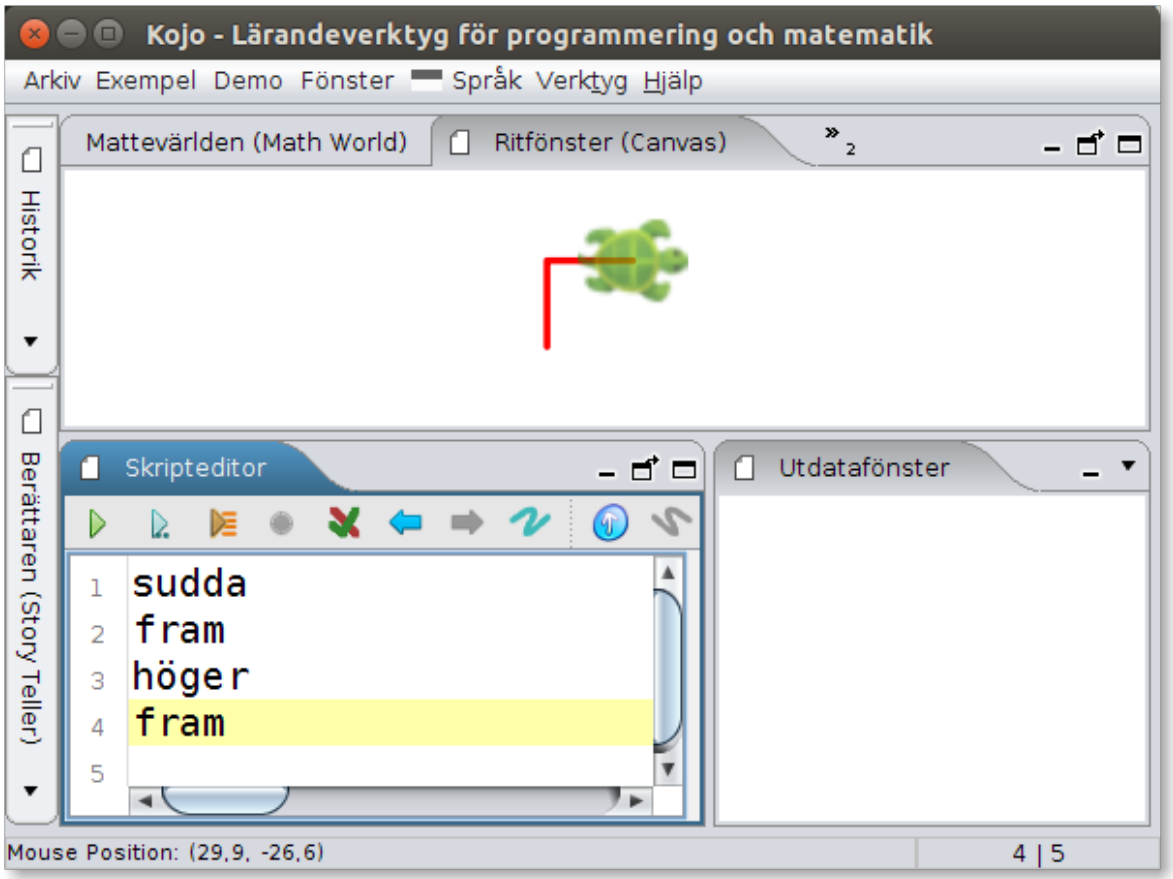
Tillståndsändring

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
```



Vad rymmer sköldpaddan i Kojo i sitt tillstånd?



position, rikting, pennfärg, pennbredd, penna uppe/nere, fyllfärg

Lata variabler och fördröjd evaluering

Med nyckelordet **`lazy`** före **`val`** skapas en s.k. ”lat” (eng. *lazy*) variabel.

```
1  scala> val striktVektor = Vector.fill(1000000)(math.random)
2  striktVektor: scala.collection.immutable.Vector[Double] =
3   Vector(0.7583305221813246, 0.9016192590993339, 0.770022134260162, 0.156677181
4
5  scala> lazy val latVektor = Vector.fill(1000000)(math.random)
6  latVektor: scala.collection.immutable.Vector[Double] = <lazy>
7
8  scala> latVektor
9  res0: scala.collection.immutable.Vector[Double] =
10   Vector(0.5391685014341797, 0.14759775960530275, 0.722606095900537, 0.9025572
```

En **`lazy val`** initialiseras **inte** vid deklarationen utan när den **refereras första gången**. Yttrycket som anges i deklarationen evalueras med s.k. **fördröjd evaluering** (även ”lat” evaluering).

Vad är egentligen skillnaden mellan `val`, `var`, `def` och `lazy val`?

```
object slump {
  val förAlltidSammaReferens  = math.random
  var kanÄndrasMedTilldelning = math.random
  def evaluerasVidVarjeAnrop  = math.random
  lazy val fördröjdInit       = Vector.fill(1000000)(math.random)
}
```

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

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 (matematiska) 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*) **objekt-orientering** och **funktionsprogrammering**:

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)
res0: Int = 2

scala> öka.   // tryck TAB
andThen   apply   compose   toString
```

Mer om **`extends`** senare i kursen...

Rekursion

Rekursiva funktioner

- Funktioner som **anropar sig själv** kallas **rekursiva**.

```
scala> def fakultet(n: Int): Int =
         if (n < 2) 1 else n * fakultet(n - 1)

scala> fakultet(5)
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): Unit = {
  def gissat = io.StdIn.readLine(s"Gissa talet mellan [1, $max]: ").toInt
  val hemlis = (math.random * max + 1).toInt
  def skrivLedtrådOmEjRätt(gissning: Int): Unit =
    if (gissning > hemlis) println(s"$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)) 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.

SimpleWindow

Färdiga, enkla funktioner för att rita finns i klassen `cslib.window.SimpleWindow`

På labben ska du använda `cslib.window.SimpleWindow`

- Paketet `cslib` innehåller paketet `window` som innehåller Java-klassen `SimpleWindow`.
- Med `SimpleWindow` kan man skapa ritfönster.
- Ladda ner `http://cs.lth.se/pgk/cslib` och lägg sedan jar-filen den katalog där du startar REPL med: `scala -cp cslib.jar`

```
$ scala -cp cslib.jar
scala> val w = new SimpleWindow(200,200,"hejsan")
```

Studera dokumentationen för `cslib.window.SimpleWindow` här: `http://cs.lth.se/pgk/api/`

Veckans övning och laboration

Övning `functions`

- Kunna skapa och använda funktioner med en eller flera parametrar, default-argument, namngivna argument, och uppdelad parameterlista.
- Kunna använda funktioner som äkta värden.
- Kunna skapa och använda anonyma funktioner (s.k. lambda-funktioner).
- Kunna applicera en funktion på element i en samling.
- Förstå skillnader och likheter mellan en funktion och en procedur.
- Förstå skillnader och likheter mellan en värde-anrop och namnanrop.
- Kunna skapa en procedur i form av en enkel kontrollstruktur med fördröjd evaluering av ett block.
- Kunna skapa och använda objekt som moduler.
- Förstå skillnaden mellan äkta funktioner och funktioner med sidoeffekter.
- Kunna skapa och använda variabler med fördröjd initialisering och förstå när de är användbara.
- Kunna förklara hur nästlade funktionsanrop fungerar med hjälp av begreppet aktiveringspost.
- Kunna skapa och använda lokala funktioner, samt förstå nyttan med lokala funktioner.
- Känna till att funktioner är objekt med en `apply`-metod.
- Känna till stegade funktioner och kunna använda partiellt applicerade argument.
- Känna till rekursion och kunna förklara hur rekursiva funktioner fungerar.

Lab `blockmole`

- Kunna kompilera Scalaprogram med `scalac`.
- Kunna köra Scalaprogram med `scala`.
- Kunna definiera och anropa funktioner.
- Kunna använda och förstå default-argument.
- Kunna ange argument med parameternamn.
- Kunna definiera objekt med medlemmar.
- Förstå kvalificerade namn och import.
- Förstå synlighet och skuggning.

4 Datastrukturer

- Vad är en datastruktur?
- Tupler
- Klasser
- Case-klasser
- Samlingar
- Integrerad utvecklingsmiljö (IDE)

Vad är en datastruktur?

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 **färdiga samlingar** i Scalas standardbibliotek där elementen är organiserade på olika vis så att samlingen får olika egenskaper som passar **olika användningsområden**:
  - `scala.collection.immutable.Vector`, snabb access **överallt**.
  - `scala.collection.immutable.List`, snabb access **i början**.
  - `scala.collection.immutable.Set`, `scala.collection.mutable.Set`, unika element, snabb innehållstest.
  - `scala.collection.immutable.Map` `scala.collection.mutable.Map`, nyckel-värde-par, snabb access via nyckel.
  - `scala.collection.mutable.ArrayBuffer`, kan ändra storlek.
  - `scala.Array`, motsvarar primitiva, föränderliga Java-arrayer, fix storlek.

Olika sätt att skapa datastrukturer

- **Tupler**
  - samla *n* st datavärden i element **_1**, **_2**, ... _*n*
  - elementen kan vara av **olika** typ
- **Klasser**
  - 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)
- **Färdiga samlingar**
  - 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

Tupler

Vad är en tupel?

- En tupel samlar *n* st objekt i en enkel struktur, med koncis syntax.
- Elementen kan vara av **olika** typ.
- `("hej", 42, math.Pi)` är en **3-tupel** av typen: `(String, Int, Double)`
- Du kan komma åt det enskilda elementen med **_1**, **_2**, ... *_n*

```
1  scala> val t = ("hej", 42, math.Pi)
2  t: (String, Int, Double) = (hej,42,3.141592653589793)
3
4  scala> t._1
5  res0: String = hej
6
7  scala> t._2
8  res1: Int = 42
```

- Tupler är praktiska när man inte vill ta det lite större arbetet att skapa en egen klass. (Men med klasser kan man göra mycket mer än med tupler.)
- I Scala kan du skapa tupler upp till en storlek av 22 element. (Behöver du fler element, använd i stället en samling, t.ex. `Vector`.)

Tupler som parametrar och returvärde.

- 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)) = math.hypot(p._1, p._2)

scala> def vinkel(p: (Double, Double)) = math.atan2(p._1, p._2)

scala> def polär(p: (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**:

```
1  scala> ("Ålder", 42)
2  res0: (String, Int) = (Ålder,42)
3
4  scala> "Ålder".->(42)
5  res1: (String, Int) = (Ålder,42)
6
7  scala> "Ålder" -> 42
8  res2: (String, Int) = (Ålder,42)
9
10 scala> Vector("Ålder" -> 42, "Längd" -> 178, "Vikt" -> 65)
11 res3: scala.collection.immutable.Vector[(String, Int)] =
12         Vector((Ålder,42), (Längd,178), (Vikt, 65))
```

Klasser

Vad är en klass?

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

```
scala> object Björn { var ålder = 49; val längd = 178 }
```

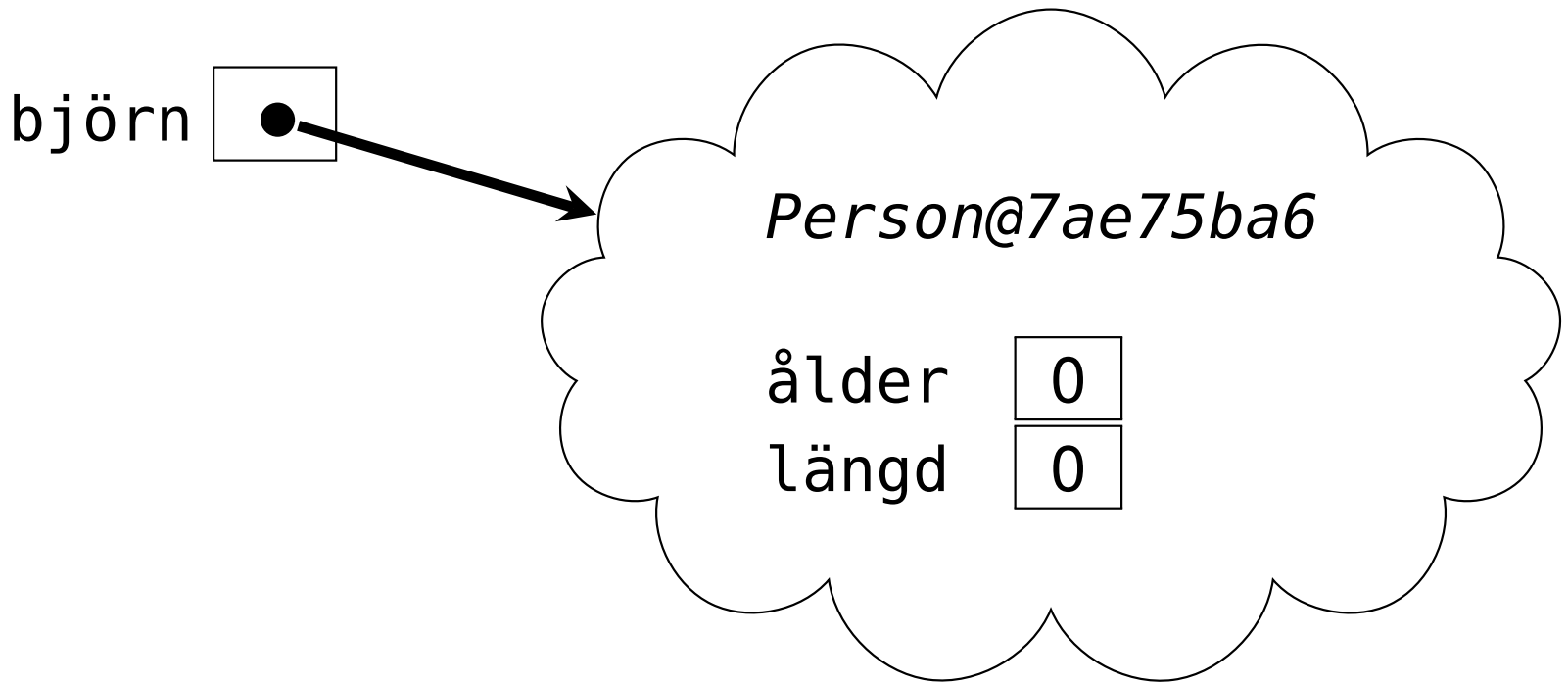
Med en **klass** kan man skapa **godtyckligt många instanser av klassen** med hjälp av nyckelordet **new** följt av klassens namn:

```
scala> class Person { var ålder = 0; var längd = 0 }

scala> val björn = new Person
björn: Person = Person@7ae75ba6

scala> björn.ålder = 49

scala> björn.längd = 178
```

- En klass kan ha **medlemmar** (i likhet med singelobjekt).
- Funktioner som är medlemmar kallas **metoder**.
- Variabler som är medlemmar kallas **attribut**.

Vid new allokeras plats i minnet för objektet

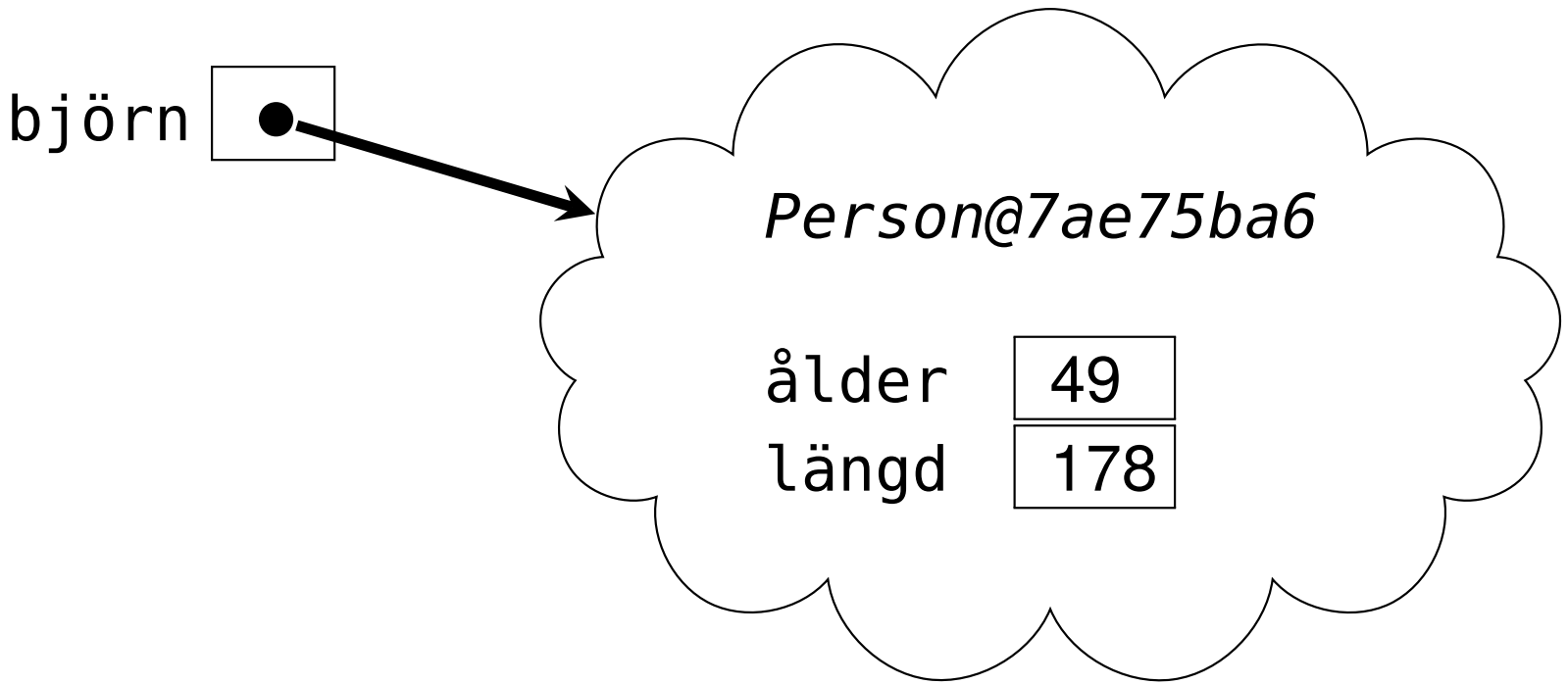
```
scala> class Person { var ålder = 0; var längd = 0 }

scala> val björn = new Person
björn: Person = Person@7ae75ba6
```



`Person@7ae75ba6` är en unik idenfierare för instansen, så att JVM hittar den i heapen.

Med punktnotation kan förändringsbara variabler tilldelas nya värden och objektets tillstånd uppdateras.

```
scala> björn.ålder = 49
scala> björn.längd = 178
```



En klass kan ha parametrar som initialiserar attribut

- Med en parameterlista efter klassnamnet får man en så kallad **primärkonstruktor** för initialisering av attribut.
- Argumenten för initialiseringen ges vid **new**.

```
scala> class Person(var ålder: Int, var längd: Int)

scala> val björn = new Person(49, 178)
björn: Person = Person@354baab2

scala> println(s"Björn är ${björn.ålder} år gammal.")
Björn är 49 år gammal.

scala> björn.ålder = 18

scala> println(s"Björn är ${björn.ålder} år gammal.")
Björn är 18 år gammal.
```

En klass kan ha privata medlemmar

Med **`private`** blir en medlem **privat**: access utifrån **medges ej**.

```
1  scala> class Person(private var minÅlder: Int, private var minLängd: Int){
2           def ålder = minÅlder
3         }
4
5  scala> val björn = new Person(42, 178)
6  björn: Person = Person@4b682e71
7
8  scala> println(s"Björn är ${björn.ålder} år gammal.")
9  Björn är 42 år gammal.
10
11 scala> björn.minÅlder = 18
12 error: variable minÅlder in class Person cannot be accessed in Person
13
14 scala> björn.längd
15 error: value längd is not a member of Person
```

Med **`private`** kan man förhindra tokiga förändringar.

Privata förändringsbara attribut och publika metoder

```
class Människa(val födelseLängd: Double, val födelseVikt: Double){
  private var minLängd = födelseLängd
  private var minVikt  = födelseVikt
  private var ålder    = 0

  def längd = minLängd  // en sådan här metod kallas "getter"
  def vikt  = minVikt   // vi förhindrar attributändring "utanför" klassen

  val slutaVäxaÅlder      = 18
  val tillväxtfaktorLängd = 0.00001
  val tillväxtfaktorVikt  = 0.0002

  def ät(mat: Double): Unit = {
    if (ålder < slutaVäxaÅlder) minLängd += tillväxtfaktorLängd * mat
    minVikt += tillväxtfaktorVikt * mat
  }

  def fyllÅr: Unit = ålder += 1

  def tillstånd: String = s"Tillstånd: $minVikt kg, $minLängd cm, $ålder år"
}
```

Tillstånd kan förändras indirekt genom metodanrop

```
1  scala> val björn = new Människa(födelseVikt=3.5, födelseLängd=52.1)
2  björn: Människa = Människa3e52
3
4  scala> björn.tillstånd
5  res0: String = Tillstånd: 3.5 kg, 52.1 cm, 0 år
6
7  scala> for (i <- 1 to 42) björn.fyllÅr
8
9  scala> björn.tillstånd
10 res2: String = Tillstånd: 3.5 kg, 52.1 cm, 42 år
11
12 scala> björn.ät(mat=5000)
13
14 scala> björn.tillstånd
15 res3: String = Tillstånd: 4.5 kg, 52.1 cm, 42 år
```

Metoden `isInstanceOf` och rot-typen `Any`

```
scala> class X(val i: Int)

scala> val a = new X(42)
a: X = X@117b2cc6

scala> a.isInstanceOf[X]
res0: Boolean = true

scala> val b = new X(42)
b: X = X@61ab6521

scala> b.isInstanceOf[X]
res1: Boolean = true

scala> a == b
res2: Boolean = false

scala> a.i == b.i
res3: Boolean = true
```

- Ett objekt skapat med **new** X är en instans av **typen** X.
- Detta kan testas med metoden `isInstanceOf[X]: Boolean`
- Typen **Any** är sypertyp till **alla** typer och kallas för **rot-typ** i Scalas typhierarki.

```
scala> a.isInstanceOf[Any]
res4: Boolean = true

scala> b.isInstanceOf[Any]
res5: Boolean = true

scala> 42.isInstanceOf[Any]
res6: Boolean = true
```

- Se quickref sid 4. (Mer i w07.)
- I klassen Any finns bl.a. `toString`

Överskugga `toString`

Alla objekt får automatiskt en metod `toString` som ger en sträng med objektets unika identifierare, här `Gurka@3830f1c0`:

```
1  scala> class Gurka(val vikt: Int)
2
3  scala> val g = new Gurka(42)
4  g: Gurka = Gurka@3830f1c0
5
6  scala> g.toString
7  res0: String = Gurka@3830f1c0
```

Man kan **överskugga** den automatiska `toString` med en **egen implementation**. Observera nyckerordet **`override`**.

```
1  scala> class Tomat(val vikt: Int){override def toString = s"Tomat($vikt g)"}
2
3  scala> val t = new Tomat(142)
4  t: Tomat = Tomat(142 g)
5
6  scala> t.toString
7  res1: String = Tomat(142 g)
```

Objektfabrik i kompanjonsobjekt

- Om det finns ett objekt i samma kodfil med samma namn som klassen blir det objektet ett s.k. **kompanjonsobjekt** (eng. *companion object*).
- Ett kompanjonsobjekt får **accessa privata medelmmar** i den klass till vilken objektet är kompanjon.
- Kompanjonsobjekt är en bra plats för s.k. **fabriksmetoder** som skapar instanser. Då slipper vi skriva **new**.

```
1  scala> :paste   // måste skrivas tillsammans annars ingen kompanjon
2
3  class Broccoli(var vikt: Int)
4
5  object Broccoli {
6    def apply(vikt: Int) = new Broccoli(vikt)
7  }
8
9  scala> val b = Broccoli(420)
10 b: Broccoli = Broccoli@32e8d5a4
```

Kompanjonsobjekt kan accessa privata medlemmar

```
class Gurka(startVikt: Double) {
  private var vikt = startVikt
  def ät(tugga: Int): Unit = if (vikt > tugga) vikt -= tugga else vikt = 0
  override def toString = s"Gurka($vikt)"
}
object Gurka {
  private var totalVikt = 0.0
  def apply(): Gurka = {
    val g = new Gurka(math.random * 0.42 + 0.1)
    totalVikt += g.vikt  // hade blivit kompileringsfel om ej vore kompanjon
    g
  }
  def rapport: String = s"Du har skapat ${totalVikt.toInt} kg gurka."
}
```

```
1  scala> val gs = Vector.fill(1000)(Gurka())
2  gs: scala.collection.immutable.Vector[Gurka] =
3    Vector(Gurka(0.49018400799506734), Gurka(0.2462822679714138), Gurka(0.173913
4
5  scala> println(Gurka.rapport)
6  Du har skapat 305 kg gurka.
```

Förändringsbara och oföränderliga objekt

Ett **oföränderligt objekt** där nya instanser skapas i stället för tillståndsändring ”på plats”.

```
class Point(val x: Int, val y: Int) {
  def moved(dx: Int, dy: Int): Point = new Point(x + dx, y + dy)

  override def toString: String = s"Point($x, $y)"
}
```

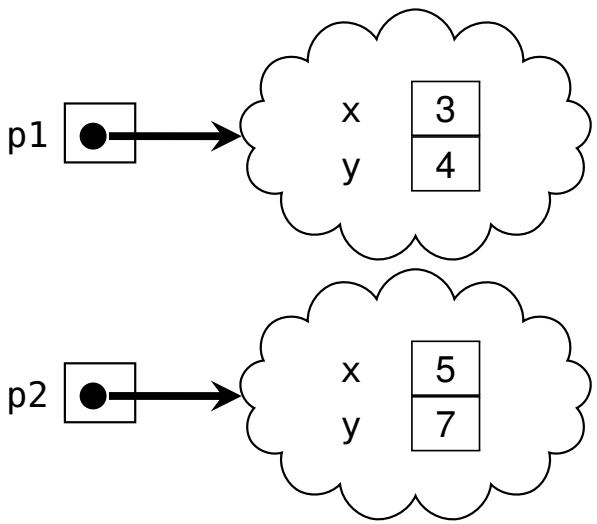
Ett **förändringsbart** objekt där **tillståndet uppdateras**.

```
class MutablePoint(private var x: Int, private var y: Int) {
  def move(dx: Int, dy: Int): Unit = {x += dx; y += dy}  // Mutation!!!

  override def toString: String = s"MutablePoint($x, $y)"
}
```

Oföränderliga objekt

```
1  scala> var p1 = new Point(3, 4)
2  p1: Point = Point(3, 4)
3
4  scala> val p2 = p1.moved(2, 3)
5  p2: Point = Point(5, 7)
6
7  scala> println(p1)
8  Point(3, 4)
9
10 scala> p1 = new Point(0, 0)
11 p1: Point = Point(0, 0)
```

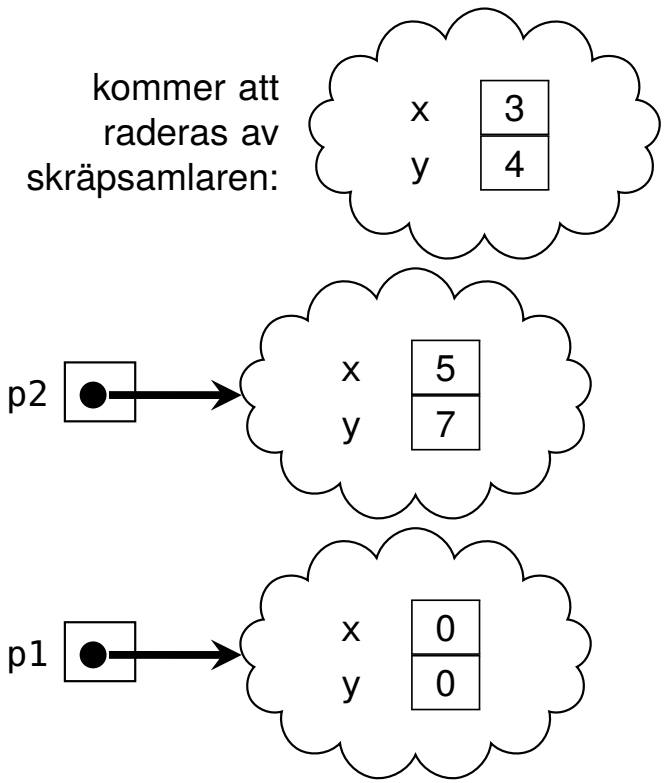
Minnessituationen efter rad 7:



Oföränderliga objekt

Minnessituationen efter rad 10:

```
1  scala> var p1 = new Point(3, 4)
2  p1: Point = Point(3, 4)
3
4  scala> val p2 = p1.moved(2, 3)
5  p2: Point = Point(5, 7)
6
7  scala> println(p1)
8  Point(3, 4)
9
10 scala> p1 = new Point(0, 0)
11 p1: Point = Point(0, 0)
```

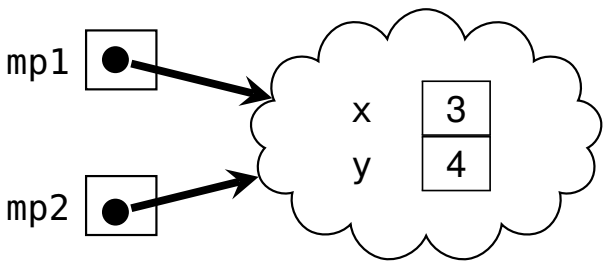


Vi kan **lugnt dela referenser** till vårt oföränderliga objekt eftersom det **aldrig** kommer att ändras.

Förändringsbara objekt

```
1  scala> val mp1 = new MutablePoint(3, 4)
2  mp1: MutablePoint = MutablePoint(3, 4)
3
4  scala> val mp2 = mp1
5  mp2: MutablePoint = MutablePoint(3, 4)
6
7  scala> mp1.move(2,3)
8
9  scala> println(mp2)
10 MutablePoint(5, 7)
```

Minnessituationen efter rad 4:

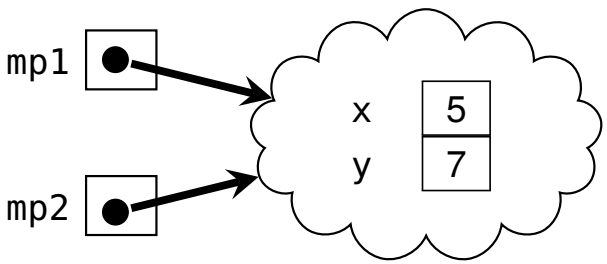


Varning! Vem som helst som har tillgång till en referens till ditt förändringsbara objekt kan **manipulera** det, vilket ibland ger överaskande och **problematiska** konsekvenser!

Förändringsbara objekt

```
1  scala> val mp1 = new MutablePoint(3, 4)
2  mp1: MutablePoint = MutablePoint(3, 4)
3
4  scala> val mp2 = mp1
5  mp2: MutablePoint = MutablePoint(3, 4)
6
7  scala> mp1.move(2,3)
8
9  scala> println(mp2)
10 MutablePoint(5, 7)
```

Minnessituationen efter **rad 7**:



Varning! Vem som helst som har tillgång till en referens till ditt förändringsbara objekt kan **manipulera** det, vilket ibland ger överaskande och **problematiska** konsekvenser!

Case-klasser

Vad är en case-klass?

- En **case**-klass är ett smidigt sätt att skapa **oföränderliga objekt**.
- Kompilatorn ger dig **en massa ”godis”** på köpet (ca 50-100 rader kod), inkl.:
  - klassparametrar blir automatiskt **val**-attribut, alltså **publika** och **oföränderliga**,
  - en automatisk **toString** som visar klassparametrarnas värde,
  - ett automatiskt **kompanjonsobjekt** med **fabriksmetod** så du slipper skriva **new**,
  - automatiska metoden **copy** för att skapa kopior med andra attributvärden, m.m...
    (Mer om detta i w06 & w11, men är du nyfiken kolla på uppgift 2d) på sid 261.)
- Det **enda** du behöver göra är att lägga till nyckelordet **case** före **class**...

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

scala> val p = Point(3, 5)
p: Point = Point(3,5)

scala> p.  // tryck TAB och se lite av allt case-klass-godis
scala> Point.  // tryck TAB och se ännu mer godis

scala> val p2 = p.copy(y= 30)
p2: Point = Point(3,30)
```

Exempel på case-klasser

```
case class Person(namn: String, ålder: Int) {
  def fyllerJämt: Boolean = ålder % 10 == 0
  def hyllning = if (fyllerJämt) "Extra grattis!" else "Vi gratulerar!"
  def ärLikaGammalSom(annan: Person) = ålder == annan.ålder
}

case class Point(x: Int = 0, y: Int = 0) {
  def distanceTo(other: Point) = math.hypot(x - other.x, y - other.y)
  def dx(d: Int): Point = copy(x + d, y)
  def dy(d: Int): Point = copy(y= y + d)  //namngivet arg. och defaultarg.
}
object Point {
  def origin = new Point()
}
```

```
1  scala> Point().dx(10).dy(10).dx(32)
2  res0: Point = Point(42,10)
3
4  scala> Point(3,4) distanceTo Point.origin
5  res1: Double = 5.0
```

Synlighet av klassparametrar i klasser & case-klasser

`private[this]` är ännu mer privat än `private`

```
class Hemlis(private val hemlis: Int) {
  def ärSammaSom(annan: Hemlis) = hemlis == annan.hemlis   // Funkar!
}

class Hemligare(private[this] val hemlis: Int) {
  def ärSammaSom(annan: Hemligare) = hemlis == annan.hemlis //KOMPILERINGSFEL
}
```

Vad händer om man inte skriver något? Olika för klass och case-klass:

```
class Hemligare(hemlis: Int) { // motsvarar private[this] val
  def ärSammaSom(annan: Hemligare) = hemlis == annan.hemlis //KOMPILERINGSFEL
}

case class InteHemlig(seMenInteRöra: Int) { // blir automatiskt val
  def ärSammaSom(annan: InteHemlig): Boolean =
    seMenInteRöra == annan.seMenInteRöra
}
```

Samlingar

Vad är en samling?

En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.

Exempel:
Heltalsvektor: `val xs = Vector(2, -1, 3, 42, 0)`

Samlingar implementeras med hjälp av klasser.
I standardbiblioteken `scala.collection` och `java.util` finns **många färdiga samlingar**, så man behöver sällan implementera egna.

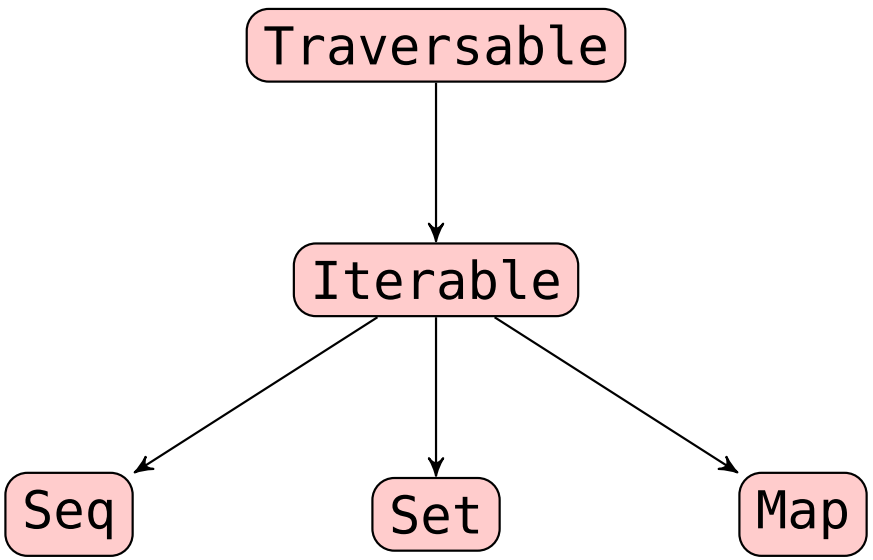
Om man behöver en egen, speciell datastruktur är det ofta lämpligt att skapa en klass som *innehåller* en *färdig* samling och utgå från dess färdiga metoder.

Typparameter möjliggör generiska samlingar

Funktioner och klasser kan, förutom vanliga parametrar, även ha **typparametrar** som skrivs i en egen parameterlista med **hakparenteser**. En typparameter gör så att funktioner och datastrukturer blir **generiska** och kan hantera element av **godtycklig** typ på ett typsäkert sätt. (Mer om detta i w09.)

```
scala> def strängLängd[T](x: T): Int = x.toString.length
strängLängd: [T](x: T)Int

scala> strängLängd[Double](42.0)  //Double är typargument
res0: Int = 4

scala> strängLängd(42.0) //Kompilatorn härleder T=Double
res1: Int = 4

scala> Vector.empty[Int] //Här kan den ej härleda typen...
res2: scala.collection.immutable.Vector[Int] = Vector()

scala> strängLängd[Vector[Int]](Vector.empty) //...men här
res3: Int = 8
```

Hierarki av samlingstyper i `scala.collection`



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

```
def foreach[U](f: Elem => U): Unit
```

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

```
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` som är en `Traversable`.

Använda `iterator`

Med en `iterator` kan man **iterera** med **`while`** över alla element, men endast **en gång**; sedan är iteratorn ”förbrukad”. (Men man kan be om en ny.)

```
1  scala> val xs = Vector(1,2,3,4)
2  xs: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4)
3
4  scala> val it = xs.iterator
5  it: scala.collection.immutable.VectorIterator[Int] = non-empty iterator
6
7  scala> while (it.hasNext) print(it.next)
8  1234
9
10 scala> it.hasNext
11 res1: Boolean = false
12
13 scala> it.next
14 java.util.NoSuchElementException: reached iterator end
15   at scala.collection.immutable.VectorIterator.next(Vector.scala:674)
```

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

Några användbara metoder på samlingar

Traversable	`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
Iterable	`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

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

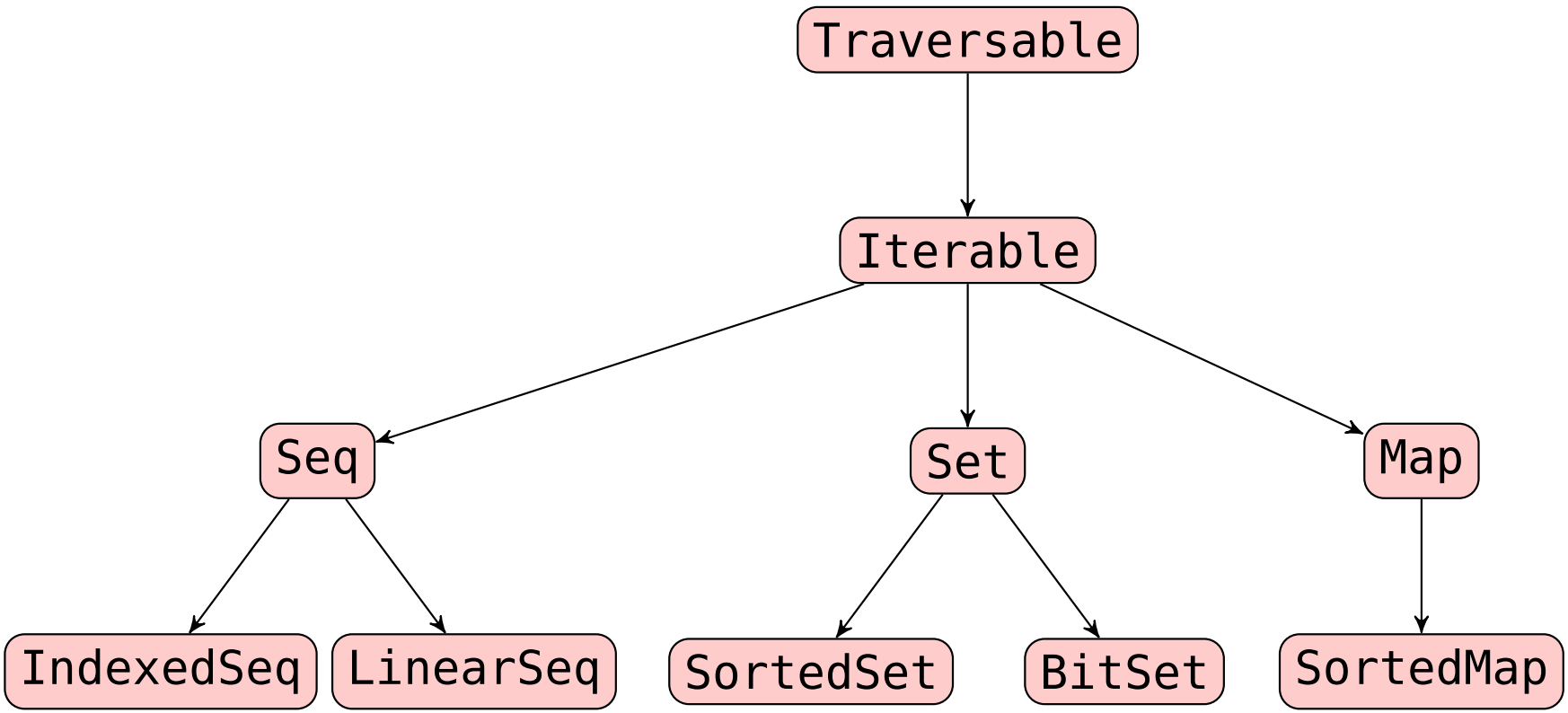
Prova fler samlingsmetoder ur snabbreferensen: `http://cs.lth.se/quickref`

Använda samlingsmetoder

```
scala> val tal = Vector(1,4,7,9,42)
tal: scala.collection.immutable.Vector[Int] = Vector(1, 4, 7, 9, 42)

scala> val jämna = tal.filter(_ % 2 == 0)
jämna: scala.collection.immutable.Vector[Int] = Vector(4, 42)

scala> val xs = Vector(("Kim","Smith"), ("Kim", "Jones"), ("Robin", "Smith"))
xs: scala.collection.immutable.Vector[(String, String)] = Vector((Kim,Smith),

scala> val grupperaEfterFörnamn = xs.groupBy(_._1)
grupperaEfterFörnamn: Map[String,Vector[(String, String)]] =
Map(Kim -> Vector((Kim,Smith), (Kim,Jones)), Robin -> Vector((Robin,Smith)))

scala> val grupperaEfterEfternamn = xs.groupBy(_._2)
grupperaEfterEfternamn: Map[String,Vector[(String, String)]] =
Map(Jones -> Vector((Kim,Jones)), Smith -> Vector((Kim,Smith), (Robin,Smith)))
```

Mer specifik samlingstyper i `scala.collection`

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



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

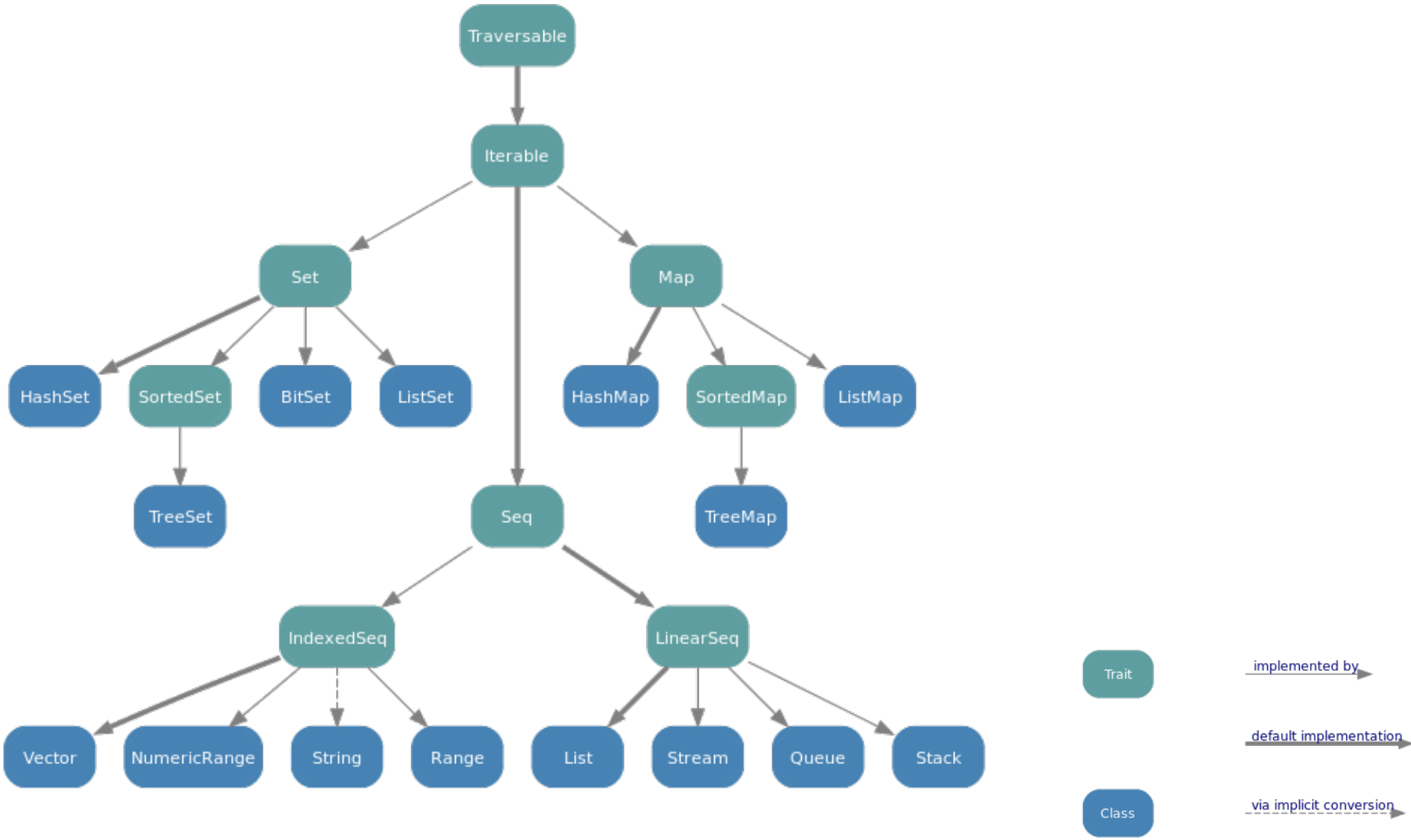
Några oföränderliga och förändringsbara sekvenssamlingar

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

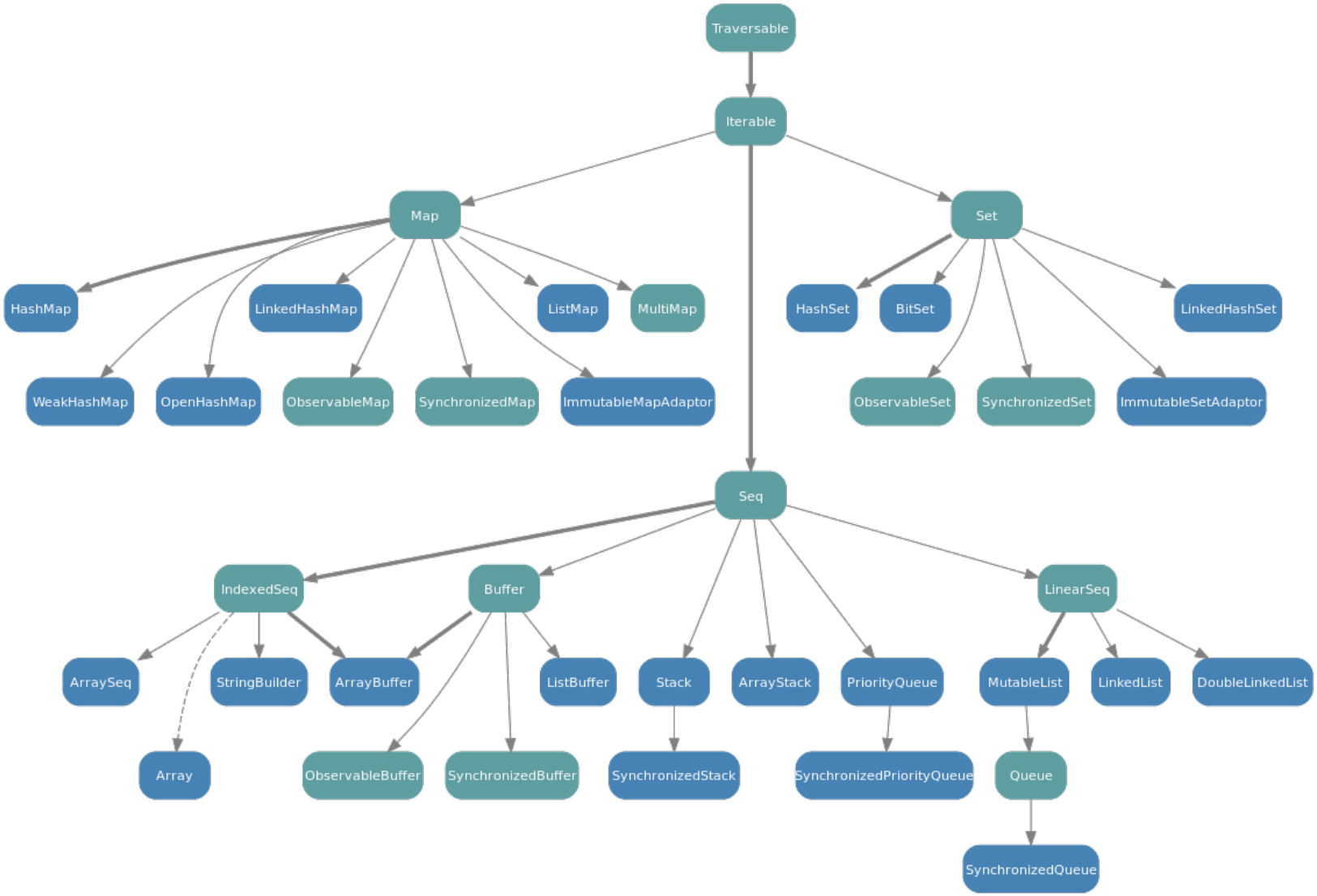
Studera samlingars egenskaper här:

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

scala.collection.immutable



scala.collection.mutable



Strängar är implicit en `IndexedSeq[Char]`

Det finns en så kallad **implicit konvertering** mellan `String` och `IndexedSeq[Char]` vilket gör att **alla samlingsmetoder på Seq funkar även 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
```

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

`Vector` eller `List`?

stackoverflow.com/questions/6928327/when-should-i-choose-vector-in-scala

1. If we only need to transform sequences by operations like map, filter, fold etc: basically it does not matter, we should program our algorithm generically and might even benefit from accepting parallel sequences. For sequential operations List is probably a bit faster. But you should benchmark it if you have to optimize.
2. If we need a lot of random access and different updates, so we should use vector, list will be prohibitively slow.
3. If we operate on lists in a classical functional way, building them by prepending and iterating by recursive decomposition: use list, vector will be slower by a factor 10-100 or more.
4. If we have an performance critical algorithm that is basically imperative and does a lot of random access on a list, something like in place quick-sort: use an imperative data structure, e.g. ArrayBuffer, locally and copy your data from and to it.

stackoverflow.com/questions/20612729/how-does-scalas-vector-work

Mer om tids- och minneskomplexitet i fördjupningskursen och senare kurser.

Mängd: snabb innehållstest, garanterat dubblettfri

En **mängd** (eng. *set*) är en samling som **inte** kan innehålla **dubbletter** och som är snabb på att avgöra om ett element **finns eller inte** i mängden.

```
1  scala> var veg = Set.empty[String]
2  veg: scala.collection.immutable.Set[String] = Set()
3
4  scala> veg = veg + "Gurka"
5  veg: scala.collection.immutable.Set[String] = Set(Gurka)
6
7  scala> veg = veg ++ Set("Broccoli", "Tomat", "Gurka")
8  veg: scala.collection.immutable.Set[String] = Set(Gurka, Broccoli, Tomat)
9
10 scala> veg.contains("Gurka")
11 res0: Boolean = true
12
13 scala> veg.apply("Gurka")   // samma som contains
14 res1: Boolean = true
15
16 scala> veg("Morot")
17 res2: Boolean = false
```

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 kort och gott **mapp** (eng. *map*),
- En hashtabell är en **samling av par**, där varje par består av en **unik nyckel** och ett tillhörande **värde**.
- Om man vet nyckeln kan man få fram värdet **snabbt**, på liknande sätt som indexering sker i en vektor om man vet heltalsindex.
- Denna datastruktur är **mycket användbar** och liknar en enkel databas.

```
1  scala> val födelse = Map("C" -> 1972,  "C++" -> 1983, "C#" -> 2000,
2    "Scala" -> 2004, "Java" -> 1995, "Javascript" -> 1995, "Python" -> 1991)
3
4  födelse: scala.collection.immutable.Map[String,Int] = Map(Scala -> 2004, C# ->
5
6  scala> födelse.apply("Scala")
7  res0: Int = 2004
8
9  scala> födelse("Java")
10 res1: Int = 1995
```

Exempel nyckel-värde-tabell

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

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

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

scala> val ärGrönSak = färg.map(elem => (elem._1, elem._2 == "grön"))
ärGrönSak: Map[String,Boolean] = Map(gurka -> true, tomat -> false, aubergine

scala> val baklängesFärg = färg.mapValues(s => s.reverse)
baklängesFärg: Map[String,String] = Map(gurka -> nörg, tomat -> dör, aubergine
```

- `xs.keySet` ger en mängd av alla nycklar
- `xs.map(f)` mappar funktionen f på alla par av (key, value)
- `xs.mapValues(f)` mappar funktionen f på alla värden

Metoderna zipWithIndex, groupBy och mapValues

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

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

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

scala> val xs = Vector(("Kim","Smith"), ("Kim", "Jones"), ("Robin", "Smith"))
xs: Vector[(String, String)] = Vector((Kim,Smith), (Kim,Jones), (Robin,Smith))

scala> val grupperaEfterFörnamn = xs.groupBy(_._1)
grupperaEfterFörnamn: Map[String,Vector[(String, String)]] =
Map(Kim -> Vector((Kim,Smith), (Kim,Jones)), Robin -> Vector((Robin,Smith)))

scala> val grupperaEfterEfternamn = xs.groupBy(_._2)
grupperaEfterEfternamn: Map[String,Vector[(String, String)]] =
Map(Jones -> Vector((Kim,Jones)), Smith -> Vector((Kim,Smith), (Robin,Smith)))

scala> val frekvens = xs.groupBy(_._1).mapValues(_.size)
frekvens: Map[String,Int] = Map(Kim -> 2, Robin -> 1)
```

Speciella metoder på förändringsbara samlingar

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

```
1  scala> import scala.collection.mutable
2
3  scala> val ms = mutable.Set.empty[Int]
4  ms: scala.collection.mutable.Set[Int] = Set()
5
6  scala> ms += 42
7  res0: ms.type = Set(42)
8
9  scala> ms += (1, 2, 3, 1, 2, 3); ms -= 1
10 res1: ms.type = Set(2, 42, 3)
11
12 scala> ms.mkString("Mängd: ", ", ", s" Antal: ${ms.size}")
13 res2: String = Mängd: 1, 2, 42, 3 Antal: 4
14
15 scala> val ordpar = mutable.Map.empty[String, String]
16 scala> ordpar += ("hej" -> "svejs", "abra" -> "kadabra", "ada" -> "lovelace")
17 scala> println(ordpar("abra"))
18 kadabra
```

Fler exempel på samlingsmetoder

Exempel: 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 ordlängMap = 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(kv => (kv._1, kv._2.size))
val sorterat = antal.toVector.sortBy(_._2)
val vanligast = antal.maxBy(_._2)
```

Jobba med föränderlig samling lokalt; returnera oföränderlig samling när du är klar

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`.

```
1  scala> :paste
2  def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6) = {
3    val s = scala.collection.mutable.Set.empty[Int]
4    var n = 0
5    while (s.size < sidor - 1) {
6      s += (math.random * sidor + 1).toInt
7      n += 1
8    }
9    (n, s.toSet)
10 }
11 scala> kastaTärningTillsAllaUtfallUtomEtt()
12 res0: (Int, scala.collection.immutable.Set[Int]) = (13,Set(5, 1, 6, 2, 3))
```

Integrerad utvecklingsmiljö (IDE)

Välja IDE

- En **integrerad utvecklingsmiljö** (eng. *Integrated Development Environment, IDE*) innehåller
  editor + kompilator + debugger + en massa annat
  och gör utvecklingen enklare när man lärt sig alla finesser.
- Läs om vad en IDE kan göra i appendix D
  (ingår i labbförberedelserna för lab `pirates`).
- På LTH:s datorer finns två populära IDE installerade:
  1. **Eclipse** med plugin **ScalaIDE** förinstallerad

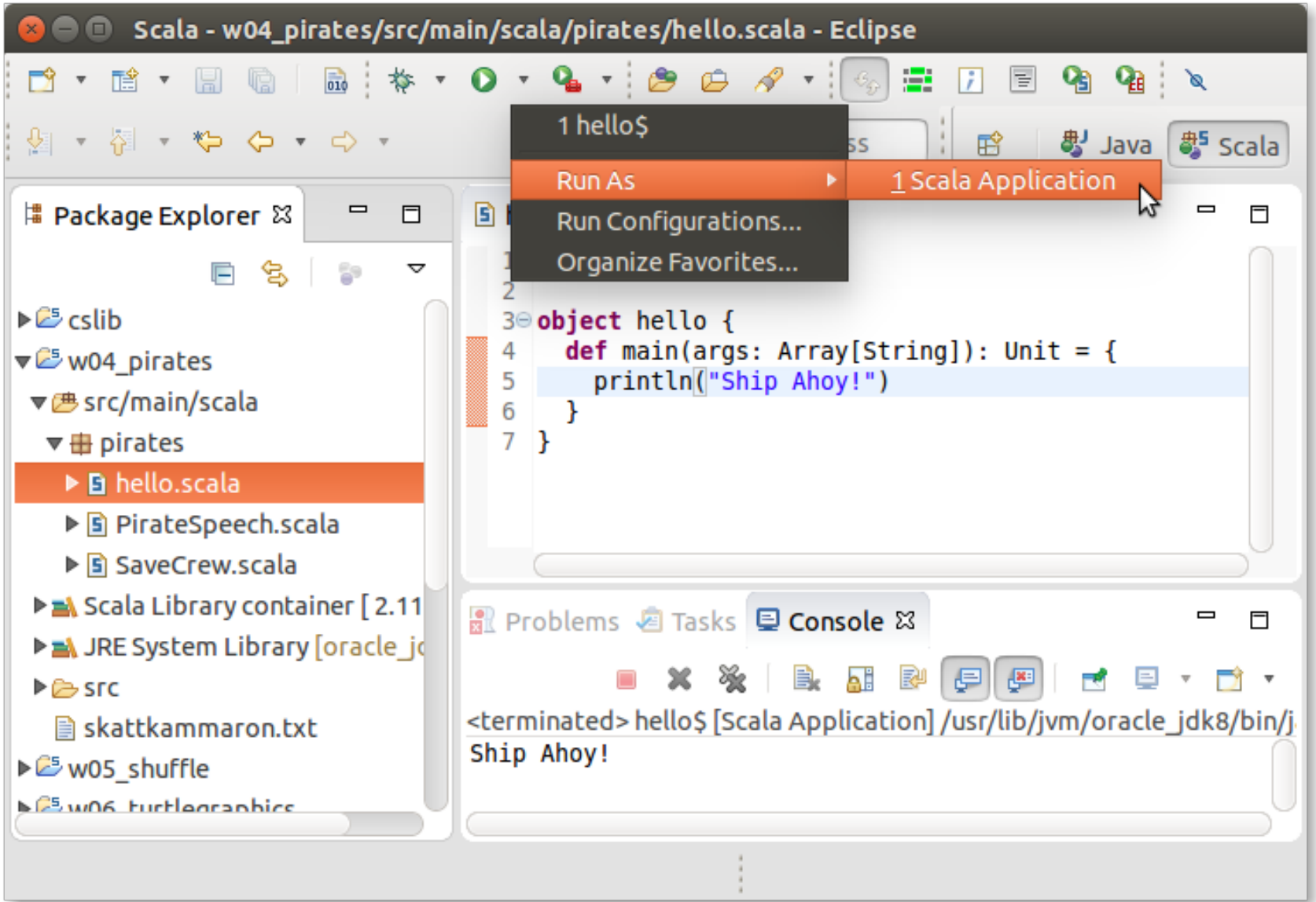
     ```
     $ scalaide
     ```

  2. **IntelliJ IDEA** (välj installera Scala-plugin när du kör första gången)

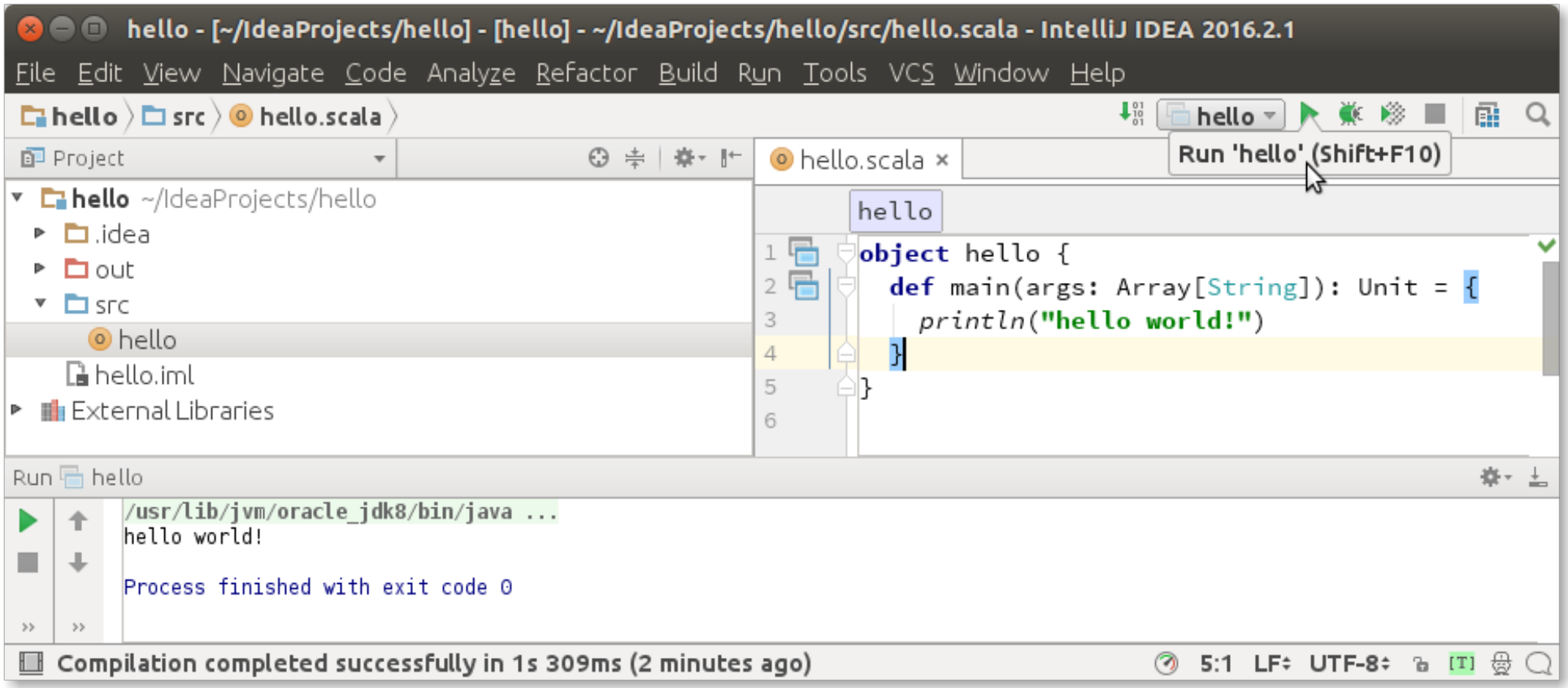
     ```
     $ idea
     ```

  Läs mer om dessa i appendix D innan du väljer vilken du vill lära dig.
  Där står även hur du installerar dem på din egen dator.
  Flest handledare har störst vana vid Eclipse.

Eclipse med ScalaIDE



IntelliJ IDEA med Scala-plugin



Denna veckas övning: `data`

- Kunna skapa och använda tupler, som variabelvärden, parametrar och returvärden.
- Förstå skillnaden mellan ett objekt och en klass och kunna förklara betydelsen av begreppet instans.
- Kunna skapa och använda attribut som medlemmar i objekt och klasser och som som klassparametrar.
- Beskriva innebörden av och syftet med att ett attribut är privat.
- Kunna byta ut implementationen av metoden `toString`.
- Kunna skapa och använda en objektfabrik med metoden `apply`.
- Kunna skapa och använda en enkel case-klass.
- Kunna använda operatornotation och förklara relationen till punktnotation.
- Förstå konsekvensen av uppdatering av föränderlig data i samband med multipla referenser.
- Känna till och kunna använda några grundläggande metoder på samlingar.
- Känna till den principiella skillnaden mellan `List` och `Vector`.
- Kunna skapa och använda en oföränderlig mängd med klassen `Set`.
- Förstå skillnaden mellan en mängd och en sekvens.
- Kunna skapa och använda en nyckel-värde-tabell, `Map`.
- Förstå likheter och skillnader mellan en `Map` och en `Vektor`.

Denna veckas laboration: `pirates`

- Kunna använda en integrerad utvecklingsmiljö (IDE).
- Kunna använda färdiga funktioner för att läsa till, och skriva från, textfil.
- Kunna använda enkla case-klasser.
- Kunna skapa och använda enkla klasser med föränderlig data.
- Kunna använda samlingstyperna `Vector` och `Map`.
- Kunna skapa en ny samling från en befintlig samling.
- Förstå skillnaden mellan kompileringsfel och exekveringsfel.
- Kunna felsöka i små program med hjälp av utskrifter.
- Kunna felsöka i små program med hjälp av en debugger i en IDE.

5 Sekvensalgoritmer

- Vad är en sekvensalgoritm?
- SEQ-COPY
- For-satser och arrayer i Java
- Exempel: PolygonWindow
- SEQ-INSERT/REMOVE-COPY
- Variabelt antal argument, ”varargs”
- SEQ-APPEND/INSERT/COPY i förändringsbar polygon
- SEQ-APPEND/INSERT/COPY med oföränderlig Polygon
- Förändringsbar eller oföränderlig? String || StringBuilder?
- Att välja sekvenssamling efter sekvensalgoritm
- Scanna filer och strängar med `java.util.Scanner`
- Återupprepningsbara pseudoslumptalssekvenser
- Registrering
- Uppgifter denna vecka

Vad är en sekvensalgoritm?

Vad är en sekvensalgoritm?

- En algoritm är en stegvis beskrivning av hur man löser ett problem.
- En sekvensalgoritm är en algoritm där dataelement i sekvens utgör en viktig del av problembeskrivningen och/eller lösningen.
- Exempel: sortera en sekvens av personer efter deras ålder.
- Två olika principer:
  - Skapa **ny sekvens** utan att förändra indatasekvensen
  - Ändra **på plats** (eng. *in place*) i den **förändringsbara** indatasekvensen

Skapa ny sekvenssamling eller ändra på plats?

- Ofta är det **lättast att skapa ny samling** och kopiera över elementen medan man loopar.
- Om man har mycket stora samlingar kan man behöva ändra på plats för att spara tid/minne.
- Det är bra att själv kunna implementera sekvensalgortimer även om många av dem finns färdiga, för att bättre förstå vad som händer ”under huven”, och för att i enstaka fall kunna optimera om det verkligen behövs.
- Vi illustrerar därför hur man kan implementera några sekvensalgoritmer med primitiva arrayer även om man sällan gör så i praktiken (i Scala).

SEQ-COPY

Algoritm: SEQ-COPY

Pseudokod för algoritmen SEQ-COPY som kopierar en sekvens, här en Array med heltal:

Indata : Heltalsarray *xs*
Resultat: 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
return *result*

Implementation av SEQ-COPY med `while`

```
1  object seqCopy {
2
3    def arrayCopy(xs: Array[Int]): Array[Int] = {
4      val result = new Array[Int](xs.length)
5      var i = 0
6      while (i < xs.length) {
7        result(i) = xs(i)
8        i += 1
9      }
10     result
11   }
12
13   def test: String = {
14     val xs = Array(1,2,3,4,42)
15     val ys = arrayCopy(xs)
16     if (xs sameElements ys) "OK!" else "ERROR!"
17   }
18
19   def main(args: Array[String]): Unit = println(test)
20 }
```

Implementation av SEQ-COPY med `for`

```
1  object seqCopyFor {
2
3    def arrayCopy(xs: Array[Int]): Array[Int] = {
4      val result = new Array[Int](xs.length)
5      for (i <- xs.indices) {
6        result(i) = xs(i)
7      }
8      result
9    }
10
11   def test: String = {
12     val xs = Array(1,2,3,4,42)
13     val ys = arrayCopy(xs)
14     if (xs sameElements ys) "OK!" else "ERROR!"
15   }
16
17   def main(args: Array[String]): Unit = println(test)
18 }
```

Implementation av SEQ-COPY med `for-yield`

```
1  object seqCopyForYield {
2
3    def arrayCopy(xs: Array[Int]): Array[Int] = {
4      val result = for (i <- xs.indices) yield xs(i)
5      result.toArray
6    }
7
8    def test: String = {
9      val xs = Array(1,2,3,4,42)
10     val ys = arrayCopy(xs)
11     if (xs sameElements ys) "OK!" else "ERROR!"
12   }
13
14   def main(args: Array[String]): Unit = println(test)
15 }
```

For-satser och arrayer i Java

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 = i + 1){ // vanligare: 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 **`while`**-sats i stället; har samma syntax i Scala & Java.

Exempel: PolygonWindow

Exempel: PolygonWindow

- En polygon kan representeras som en sekvens av punkter, där varje punkt är en 2-tupel: `Seq[(Int, Int)]`
- `PolygonWindow` nedan är ett fönster som kan rita en polygon.

```
1  class PolygonWindow(width: Int, height: Int) {
2    val w = new cslib.window.SimpleWindow(width, height, "PolyWin")
3
4    def draw(pts: Seq[(Int, Int)]): Unit = if (pts.size > 0) {
5      w.moveTo(pts(0)._1, pts(0)._2)
6      for (i <- 1 until pts.length) w.lineTo(pts(i)._1, pts(i)._2)
7      w.lineTo(pts(0)._1, pts(0)._2)
8    }
9  }
```

```
1  object polygonTest1 {
2    def main(args: Array[String]): Unit = {
3      val pw = new PolygonWindow(200,200)
4      val pts = Array((50,50), (100,100), (50,100), (30,50))
5      pw.draw(pts)
6    }
7  }
```

Typ-alias för att abstrahera typnamn

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

```
1  scala> type Pt = (Int, Int)
2
3  scala> def distToOrigo(pt: Pt): Int = math.hypot(pt._1, pt._2)
4
5  scala> type Pts = Vector[Pt]
6
7  scala> def firstPt(pts: Pts): Pt = pts.head
8
9  scala> val xs: Pts = Vector((1,1),(2,2),(3,3))
10
11 scala> firstPt(xs)
12 res0: Pt = (1,1)
```

Detta är bra om:

- man har en lång och krånglig typ och vill använda ett kortare namn,
- om man vill abstrahera en typ och öppna för möjligheten att byta implementation senare (t.ex. till en egen klass), medan man ändå kan fortsätta att använda befintligt namn.

SEQ-INSERT/REMOVE-COPY

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`

Resultat: En ny sekvens av typen `Array[Pt]` som är en kopia av *pts* men där *pt* är infogat på plats *pos*

---

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

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

Insättning/borttagning i kopia av primitiv Array

```
1  object pointSeqUtils {
2    type Pt = (Int, Int)  // a type alias to make the code more concise
3
4    def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] = {
5      val result = new Array[Pt](pts.length + 1)  // initialized with null
6      for (i <- 0 until pos) result(i) = pts(i)
7      result(pos) = pt
8      for (i <- pos + 1 to pts.length) result(i) = pts(i - 1)
9      result
10   }
11
12   def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] =
13     if (pts.length > 0) {
14       val result = new Array[Pt](pts.length - 1) // initialized with null
15       for (i <- 0 until pos) result(i) = pts(i)
16       for (i <- pos + 1 until pts.length) result(i - 1) = pts(i)
17       result
18     } else Array.empty
19
20   // above methods implemented using the powerful Scala collection method patch:
21
22   def insertCopy(pts: Array[Pt], pos: Int, pt: Pt) = pts.patch(pos, Array(pt), 0)
23
24   def removeCopy(pts: Array[Pt], pos: Int) = pts.patch(pos, Array.empty[Pt], 1)
25 }
```

Man gör **mycket lätt fel** på gränser/specialfall: +-1, to/until, tom sekvens etc.

Exempel: Test av SEQ-INSERT/REMOVE-COPY

```
1  object polygonTest2 {
2    def main(args: Array[String]): Unit = {
3      val pw = new PolygonWindow(200,200)
4      val pts = Array((50,50), (100,100), (50,100), (30,50))
5      pw.draw(pts)
6
7      val morePts = pointSeqUtils.primitiveInsertCopy(pts, 2, (90,130))
8      //val morePts = pointSeqUtils.insertCopy(pts, 2, (90,130))
9      pw.draw(morePts)
10
11     val lessPts = pointSeqUtils.primitiveRemoveCopy(morePts, morePts.length - 1)
12     //val lessPts = pointSeqUtils.removeCopy(morePts, morePts.length - 1)
13     pw.draw(lessPts)
14
15   }
16 }
```

Exempel: Göra insättning med take/drop

Om du inte vill ”uppfinna hjulet” och inte använda `patch` kan du göra så här:
Använd `take` och `drop` tillsammans med `:+` och `++`
Du kan också göra insättningen generiskt användbar för alla sekvenser:

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

scala> val ys = (xs.take(2) :+ 42) ++ xs.drop(2)
ys: scala.collection.immutable.Vector[Int] =
  Vector(1, 2, 42, 3)

scala> def insertCopy[T](xs: Seq[T], elem: T, pos: Int) =
         (xs.take(pos) :+ elem) ++ xs.drop(pos)

scala> insertCopy(xs, 42, 2)
res0: Seq[Int] = Vector(1, 2, 42, 3)
```

Övning: Implementera `insertCopy[T]` med `patch` istället.

Variabelt antal argument, ”varargs”

Parameter med variabelt antal argument, ”varargs”

Med en asterisk efter parametertypen kan antalet argument variera:

```
def sumSizes(xs: String*): Int = xs.map(_.size).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
```

Typen på `xs` blir en `Seq[String]`, egentligen en `WrappedArray[String]` som kapslar in en array så den beter sig mer som en ”vanlig” Scala-samling.

Sekvenssamling som argument till varargs-parameter

```
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 parameter som accepterar variabelt antal argument med typannoteringen

```
: _*
```

direkt **efter** sekvenssamlingen.

```
scala> sumSizes(veg: _*)
res5: Int = 10
```

SEQ-APPEND/INSERT/COPY i förändringsbar polygon

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

```
1  class PolygonArray(val maxSize: Int) {
2    type Pt = (Int, Int)
3    private val points = new Array[Pt](maxSize)  // initialized with null
4    private var n = 0
5    def size = n
6
7    def draw(w: PolygonWindow): Unit = w.draw(points.take(n))
8
9    def append(pts: Pt*): Unit = {
10     for (i <- pts.indices) points(n + i) = pts(i)
11     n += pts.length
12   }
13
14   def insert(pos: Int, pt: Pt): Unit = { // exercise: change pt to varargs pts
15     for (i <- n until pos by -1) points(i) = points(i - 1)
16     points(pos) = pt
17     n += 1
18   }
19
20   def remove(pos: Int): Unit = { // exercise: change pos to fromPos, replaced
21     for (i <- pos until n) points(i) = points(i + 1)
22     n -= 1
23   }
24
25   override def toString = points.mkString("PrimitivePolygon(",",",")")
26 }
```

Test av PolygonArray, ändring på plats

```
1  object polygonTest3 {
2    def main(args: Array[String]): Unit = {
3      val pw = new PolygonWindow(200,200)
4      val poly = new PolygonArray(100)
5
6      poly.append((50,50), (100,100), (50,100), (30,50))
7      println(poly)
8      poly.draw(pw)
9
10     poly.insert(2, (100,150))
11     println(poly)
12     poly.draw(pw)
13
14     poly.remove(0)
15     println(poly)
16     poly.draw(pw)
17   }
18 }
```

Exempel: PolygonVector, variabel referens till oföränderlig datastruktur

```
1  class PolygonVector {
2    type Pt = (Int, Int)
3    private var points = Vector.empty[Pt]  // note var declaration to allow mutation
4    def size = points.size
5
6    def draw(w: PolygonWindow): Unit = w.draw(points.take(size))
7
8    def append(pts: Pt*): Unit = {
9      points ++= pts.toVector
10   }
11
12   def insert(pos: Int, pt: Pt): Unit = { // exercise: change pt to varargs pts
13     points = points.patch(pos, Vector(pt), 0)
14   }
15
16   def remove(pos: Int): Unit = { // exercise: change pos to fromPos, replaced
17     points = points.patch(pos, Vector(), 1)
18   }
19
20   override def toString = points.mkString("PrimitivePolygon(",",",")")
21 }
```

Test av PolygonVector, variabel referens till oföränderlig datastruktur

```
1  object polygonTest4 {
2    def main(args: Array[String]): Unit = {
3      val pw = new PolygonWindow(200,200)
4      val poly = new PolygonVector
5
6      poly.append((50,50), (100,100), (50,100), (30,50))
7      println(poly)
8      poly.draw(pw)
9
10     poly.insert(2, (100,150))
11     println(poly)
12     poly.draw(pw)
13
14     poly.remove(0)
15     println(poly)
16     poly.draw(pw)
17   }
18 }
```

SEQ-APPEND/INSERT/COPY med oföränderlig Polygon

Exempel: Polygon som oföränderlig case class

```
1  object Polygon {
2    type Pt = (Int, Int)
3    type Pts = Vector[Pt]
4    def apply() = new Polygon(Vector())
5  }
6
7  import Polygon.{Pt, Pts}
8
9  case class Polygon(points: Pts) {
10   def size = points.size // for convenience but not strictly necessary (why?)
11
12   def append(pts: Pt*) = copy(points ++ pts.toVector)
13
14   def insert(pos: Int, pts: Pt*) = copy(points.patch(pos, pts, 0))
15
16   def remove(pos: Int, replaced: Int = 1) = copy(points.patch(pos, Seq(), replaced))
17 }
```

- Nu är attributet points en publik **`val`** som vi kan dela med oss av eftersom datastrukturen `Vector` är oföränderlig.
- Vi behöver inte införa ett beroende till `PolygonWindow` här då vi ger tillgång till sekvensen av punkter som kan användas vid anrop av `PolygonWindow.draw`
- Att ändra implementationen till något annat än `Vector` blir lätt om klientkoden använder typ-alias `Polygon.Pts` i stället för `Vector[(Int, Int)]`.

Test av Polygon som oföränderlig case class

```
1  object polygonTest5 {
2    def main(args: Array[String]): Unit = {
3      val pw = new PolygonWindow(200,200)
4      var poly = Polygon()
5
6      poly = poly.append((50,50), (100,100), (50,100), (30,50))
7      println(poly)
8      pw.draw(poly.points)
9
10     poly = poly.insert(2, (100,150))
11     println(poly)
12     pw.draw(poly.points)
13
14     poly = poly.remove(0)
15     println(poly)
16     pw.draw(poly.points)
17   }
18 }
```

Förändringsbar eller oföränderlig? String || StringBuilder?

Förändringsbar eller oföränderlig?

- Om den underligande **oföränderliga** datastrukturen är **smart** implementerad så att den **återanvänder redan allokerade objekt** – vilket ju är ofarligt eftersom de aldrig kommer att ändras – så är oföränderlighet **minst lika snabbt** som förändring på plats.
- Det är först när man gör **väldigt många** upprepade ändringar på, fördatastrukturen ogynsam plats, som det blir långsamt.
- Hur många är ”väldigt många”?
  $\rightarrow$ Det ska vi undersöka nu.

String eller StringBuilder?

- Strängar i JVM är **oföränderliga**.
- Implementationen av sekvensdatastrukturen `java.lang.String` är **mycket effektivt** implementerad, där **redan allokerade objekt ofta kan återanvänds**.
- **MEN** väldigt många tillägg på slutet blir långsamt. Därför finns den föränderliga `StringBuilder` med den effektivt implementerade metoden append som **ändrar på plats**.
- Undersök dokumentationen för `StringBuilder` här:
  `https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html`
- För vilka teckensekvensalgoritmer är det lönt att använda `StringBuilder`?
  $\rightarrow$ Det ska vi undersöka nu.

Timer

- `System.currentTimeMillis` ger tiden i millisekunder sedan januari 1970.
- Med `Timer.measure{ xxx }` nedan kan man mäta tiden det tar för xxx.
- Ett par `(elapsedMillis, result)` returneras som innehåller tiden det tar att köra blocket, samt resultatet av blocket.

```
1  object Timer {
2    private var startTime: Long = System.currentTimeMillis
3
4    def elapsedMillis: Long = System.currentTimeMillis - startTime
5
6    def reset: Unit = { startTime = System.currentTimeMillis }
7
8    def measure[T](block: => T): (Long, T) = {
9      reset
10     val result = block
11     (elapsedMillis, result)
12   }
13 }
```

NanananananananaNanananananananananaBatman

Prova denna kod: compendium/examples/workspace/w05-seqalg/src/NanananananananaNanananananananananaBatman.scala
medan du lyssnar till: www.youtube.com/watch?v=oDc-1zfffMw

```
1  object NanananananananaNanananananananananaBatman {
2    val na = "NanananananananaNanananananananananaBatman"
3
4    def batmanImmutable(n: Int): (Long, String) = Timer.measure {
5      var result: String = na  // Strings are immutable
6      for (i <- 2 to n) {
7        result = result + na   // Allocates a new String for each append
8      }
9      result  // return da String
10   }
11
12   def batmanMutable(n: Int): (Long, String) = Timer.measure {
13     var sb = new StringBuilder(na)  // StringBuilder is mutable
14     for (i <- 2 to n) {
15         sb.append(na)     // append ***mutates*** the instance in place
16     }
17     sb.toString           // convert to immutable String and return
18   }
19
20   def main(args: Array[String]): Unit = {
21     val warmupJVM = (batmanMutable(100), batmanImmutable(100))
```

Att välja sekvenssamling efter sekvensalgoritm

Oföränderlig eller förändringsbar?

- **Oföränderlig**: Kan ej ändra elementreferenserna, men effektiv på att skapa kopia som är (delvis) förändrad (vanliga i Scala, men inte i Java): **Vector** eller **List**
- **Förändringsbar**: kan ändra elemententreferenserna
  - Kan **ej ändra storlek** efter allokering:
    Scala+Java: **Array**: indexera och uppdatera varsomhelst
  - Kan ändra storlek efter allokering:
    Scala: **ArrayBuffer** eller **ListBuffer**
    Java: **ArrayList** eller **LinkedList**
- Ofta funkar oföränderlig sekvenssamling utmärkt, men om man efter prestandamätning upptäcker en flaskhals kan man ändra från **Vector** till t.ex. **ArrayBuffer**.

Egenskaper hos några sekvenssamlingar

- `Vector`
  - **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
  - Allsidig prestanda: **bra till det mesta**.
- `List`
  - **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
  - Snabb vid bearbetning **i början**.
  - Smidig & snabb vid **rekursiva** algoritmer.
  - Långsam vid upprepad **indexering** på godtyckliga ställen.
- `Array`
  - **Föränderlig**: **snabb indexering & uppdatering**.
  - Kan **ej ändra storlek**; storlek anges vid allokering.
  - Har särställning i JVM: ger snabbaste minnesaccessen.
- `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.

Vilken sekvenssamling ska jag välja?

- `Vector`
  - Om du vill ha oföränderlighet: `val xs = Vector[Int](1,2,3)`
  - Om du behöver ändra (men ej prestandakritiskt):
    `var xs = Vector.empty[Int]`
  - Om du ännu inte vet vilken sekvenssamling som är bäst; du kan alltid ändra efter att du mätt prestanda och kollat flaskhalsar.
- `List`
  - Om du har en rekursiv sekvensalgoritm och/eller bara lägger till i början.
- `Array`
  - Om det behövs av prestandaskäl och du **vet** storlek vid allokering:
    `val xs = Array.fill(initSize)(initValue)`
- `ArrayBuffer`
  - Om det behövs av prestandaskäl och du **inte** vet storlek vid allokering:
    `val xs = scala.collection.mutable.empty[Int]`
- `ListBuffer`
  - om det behövs av prestandaskäl och du bara behöver lägga till i början:
    `val xs = scala.collection.mutable.ListBuffer.empty[Int]`

Lämna det öppet: använd Seq[T]

```
def varannanBaklänges[T](xs: Seq[T]): Seq[T] =
  for (i <- xs.indices.reverse by -2) yield xs(i)
```

Fungerar med alla sekvenssamlingar:

```
scala> varannanBaklänges(Vector(1,2,3,4,5))
res0: Seq[Int] = Vector(5, 3, 1)

scala> varannanBaklänges(List(1,2,3,4,5))
res1: Seq[Int] = List(5, 3, 1)

scala> varannanBaklänges(collection.mutable.ListBuffer(1,2))
res2: Seq[Int] = Vector(2)
```

Scalas standardbibliotek returnerar ofta lämpligaste specifika sekvenssamlingen som är subtyp till Seq[T].

Scanna filer och strängar med `java.util.Scanner`

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
```

Återupprepningsbara pseudoslumptalssekvenser

Klassen java.util.Random

- Om man använder slumptal kan det vara svårt att leta buggar, efter som det blir **olika varje gång** man kör programmet och buggen kanske bara uppstår ibland.
- Med klassen `java.util.Random` kan man skapa **pseudo**-slumptalssekvenser.
- Om man ger ett **frö** (eng. *seed*) av typen Long som argument till konstruktorn när man skapar en instans av klassen Radnom, får man samma ”slumpmässiga” sekvens **varje gång** man kör programmet.

```
val seed = 42
val rnd = new java.util.Random(seed)  // SAMMA sekvens varje körning
val r = rnd.nextInt(6) // ger slumptal mellan 0 till och med 5
```

- Om man **inte** ger ett **frö** så sätts fröet till ”*a value very likely to be distinct from any other invocation of this constructor*”. Då vet vi inte vilket fröet blir och det blir olika varje gång man kör programmet.

```
val rnd = new java.util.Random  // OLIKA sekvens varje körning
val r = rnd.nextInt(6) // ger slumptal mellan 0 till och med 5
```

- Studera dokumentationen för klassen `java.util.Random` här: docs.oracle.com/javase/8/docs/api/java/util/Random.html

Syresättning av hjärnan vid sövande föreläsning

Prova nedan kod som finns här:

```
1  object FixSleepyBrain {
2    val seed = 42
3    val rnd = new java.util.Random(seed)
4    val names = scala.io.Source.fromFile("src/names.txt").getLines.toSet
5    def delay = Thread.sleep(3000)
6
7    def main(args: Array[String]): Unit = {
8      println("*** FIX YOUR SLEEPY BRAIN ***\n\nWHEN YOUR NAME STARTS WITH...")
9      while (true) {
10       val letter = (rnd.nextInt('Z' - 'A') + 'A').toChar
11       val theChosenOnes = names.filter(_.contains(letter))
12       val action = if (theChosenOnes.isEmpty) "EVERY BODY SIT!!!" else "STAND UP"
13       delay
14       println(s"\n$letter : $action $theChosenOnes")
15     }
16   }
17 }
```

Medan du lyssnar till: www.youtube.com/watch?v=zUwEIt9ez7M
Eller: www.youtube.com/watch?v=rvXxlXg_V-k

Registrering

Registrering

- **Registrering** innefattar algoritmer för att räkna antalet förekomster av olika saker.
- 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.

```
scala> val rnd = new java.util.Random(42L)
rnd: java.util.Random = java.util.Random@6d946eee

scala> val reg = new Array[Int](6)
reg: Array[Int] = Array(0, 0, 0, 0, 0, 0)

scala> for (i <- 1 to 1000) reg(rnd.nextInt(6)) += 1

scala> for (i <- 1 to 6) println(i +": " + reg(i - 1))
1: 178
2: 187
3: 167
4: 148
5: 155
6: 165
```

Registrering av tärningskast i Map, imperativ lösning

Vi registrerar antalet i en Map[Int, Int] där nyckeln är antalet tärningsögon och värdet är frekvensen.

```
scala> val rnd = new java.util.Random(42L)
rnd: java.util.Random = java.util.Random@6d946eee

scala> var reg = (1 to 6).map(i => i -> 0).toMap
reg: scala.collection.immutable.Map[Int,Int] =
  Map(5 -> 0, 1 -> 0, 6 -> 0, 2 -> 0, 3 -> 0, 4 -> 0)

scala> for (i <- 1 to 1000) {
         val t = rnd.nextInt(6) + 1
         reg = reg + ((t, reg(t) + 1))
       }

scala> reg
res0:scala.collection.immutable.Map[Int,Int]= Map(5 -> 155,
1 -> 178, 6 -> 165, 2 -> 187, 3 -> 167, 4 -> 148)
```

Registrering av tärningskast i `collection.mutable.Map`, imperativ lösning

Om vi är bekymrade över prestanda:

```
1  scala> val rnd = new java.util.Random(42L)
2  rnd: java.util.Random = java.util.Random@6d946eee
3
4  scala> val initPairs = (1 to 6).map(i => i -> 0)
5  initPairs: scala.collection.immutable.IndexedSeq[(Int, Int)] =
6  Vector((1,0), (2,0), (3,0), (4,0), (5,0), (6,0))
7
8  scala> var reg = scala.collection.mutable.Map(initPairs: _*)
9
10 scala> for (i <- 1 to 1000) {
11           val t = rnd.nextInt(6) + 1
12           reg(t) = reg(t) + 1
13         }
14
15 scala> reg
16 res0: scala.collection.mutable.Map[Int,Int] =
17 Map(2 -> 187, 5 -> 155, 4 -> 148, 1 -> 178, 3 -> 167, 6 -> 165)
```

Registrering av tärningskast i Map, funktionell lösning

Oföränderlighet: Skapa nya samlingar utan att ändra något.

```
scala> val rnd = new java.util.Random(42L)
rnd: java.util.Random = java.util.Random@6d946eee

scala> val dice = (1 to 1000).map(i => rnd.nextInt(6) + 1)

scala> dice.groupBy(i => i).mapValues(_.size)
res0:scala.collection.immutable.Map[Int,Int]= Map(5 -> 155,
1 -> 178, 6 -> 165, 2 -> 187, 3 -> 167, 4 -> 148)
```

Övn. för den nyfikne: mät prestanda för de olika lösningarna.

Syresättning av hjärnan med registrering av utvalda

```
1  object FixSleepyBrainRegisterChosen {
2    val seed = 42
3    val rnd = new java.util.Random(seed)
4    val names = scala.io.Source.fromFile("src/names.txt").getLines.toSet
5    val initPairs = names.map(n => n -> 0).toSeq
6    val countChosen = scala.collection.mutable.Map(initPairs: _*)
7    def delay = Thread.sleep(3000)
8
9    def main(args: Array[String]): Unit = {
10     println("*** FIX YOUR SLEEPY BRAIN ***\n\nTOGGLE WHEN YOUR NAME INCLUDES...")
11     var n = 0
12     while (countChosen.values.filter(_ == 0).size > 0) {
13       n += 1
14       val letter = (rnd.nextInt('Z' - 'A') + 'A').toChar
15       val theChosenOnes = names.filter(_.toUpperCase.contains(letter))
16       val action = if (theChosenOnes.isEmpty) "EVERY BODY SIT!!!" else "STAND or SIT"
17       delay
18       println(s"\n$n: $letter : $action $theChosenOnes")
19       for (name <- theChosenOnes) countChosen(name) += 1
20     }
21     countChosen.toSeq.sortBy(_._2).foreach(println)
22   }
23 }
```

Medan du lyssnar till: https://www.youtube.com/watch?v=ZVrgj3A0_BY

Uppgifter denna vecka

Denna veckas övning: `sequences`

- Kunna implementera funktioner som tar argumentsekvenser av godtycklig längd.
- Kunna tolka enkla sekvensalgoritmer i pseudokod och implementera dem i programkod, t.ex. tillägg i slutet, insättning, borttagning, omvändning, etc., både genom kopiering till ny sekvens och genom förändring på plats i befintlig sekvens.
- Kunna använda föränderliga och oföränderliga sekvenser.
- Förstå skillnaden mellan om sekvenser är föränderliga och om innehållet i sekvenser är föränderligt.
- Kunna välja när det är lämpligt att använda `Vector`, `Array` och `ArrayBuffer`.
- Känna till att klassen `Array` har färdiga metoder för kopiering.
- Kunna implementera algoritmer som registrerar antalet förekomster av något utfall i en sekvens som indexeras med utfallet.
- Kunna generera sekvenser av pseudoslumptal med specificerat slumptalsfrö.
- Kunna implementera sekvensalgoritmer i Java med **`for`**-sats och primitiva arrayer.
- Kunna beskriva skillnaden i syntax mellan arrayer i Scala och Java.
- Kunna använda klassen `java.util.Scanner` i Scala och Java för att läsa in heltalssekvenser från `System.in`.

Denna veckas laboration: `shuffle`

- Kunna skapa och använda sekvenssamlingar.
- Kunna använda sekvensalgoritmen SHUFFLE för blandning på plats av innehållet i en array.
- Kunna registrera antalet förekomster av olika värden i en sekvens.

6 Klasser

- Vad är en klass?
- Olika sätt att skapa instanser
- Minneshantering
- Referens saknas: `null`
- Klasser i Java
- Referensen `this`
- Getters och setters
- Likhet
- Case-klasser och likhet
- Implementation saknas: ???
- Klass-specifikationer
- Grumligt-lådan
- Veckans uppgifter

Vad är en klass?

Vad är en klass?

- En klass är en mall för att skapa objekt.
- Objekt skapas med `new Klassnamn` och kallas för **instanser** av klassen `Klassnamn`.
- En klass innehåller medlemmar (eng. *members*):
  - **attribut**, kallas även fält (eng. *field*): `val`, `lazy val`, `var`
  - **metoder**, kallas även operationer: `def`
- Varje instans har sin uppsättning värden på attributen (fälten).

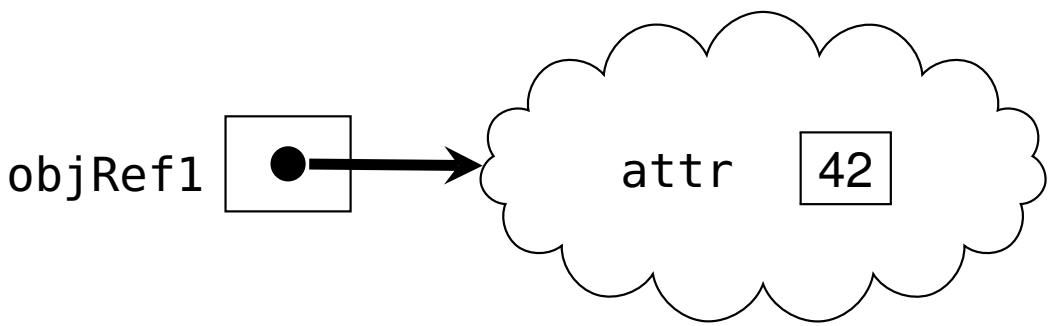
Vad är en klass?

Metafor: En klass liknar en **stämpel**

- En stämpel kan tillverkas – motsvarar deklaration av klassen.
- Det händer inget förrän man stämplar – motsvarar `new`.
- Då skapas avbildningar – motsvarar instanser av klassen.

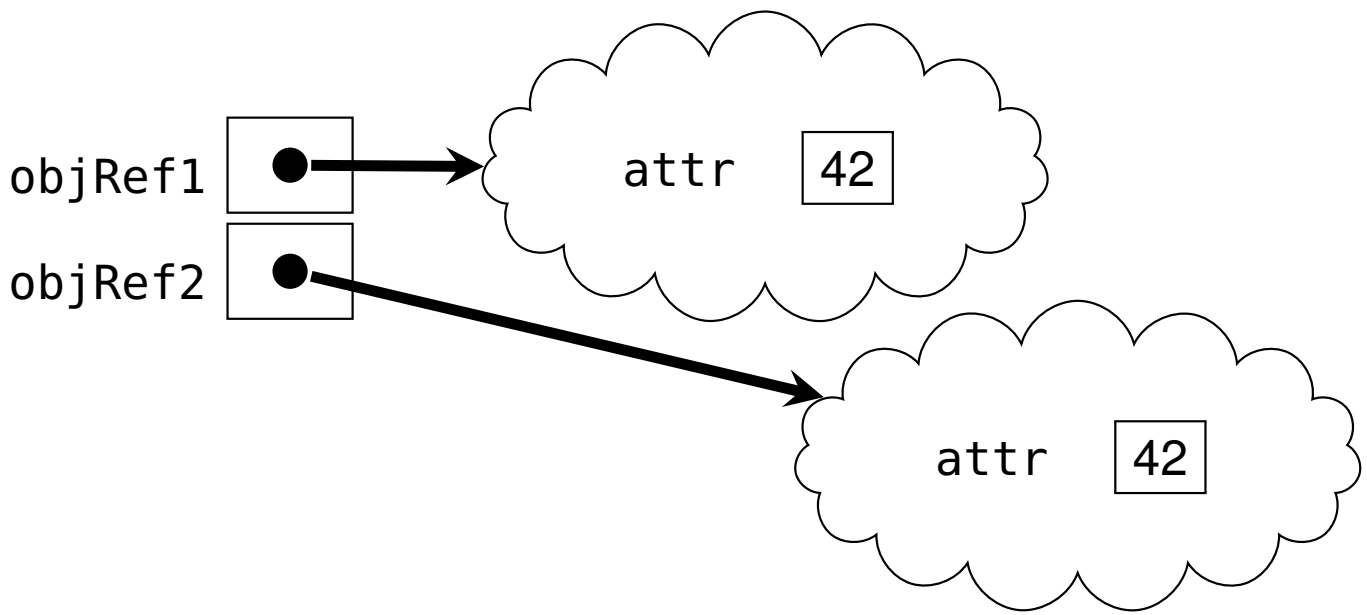
Klass och instans

```
scala> class C { var attr = 42 }

scala> val objRef1 = new C
```



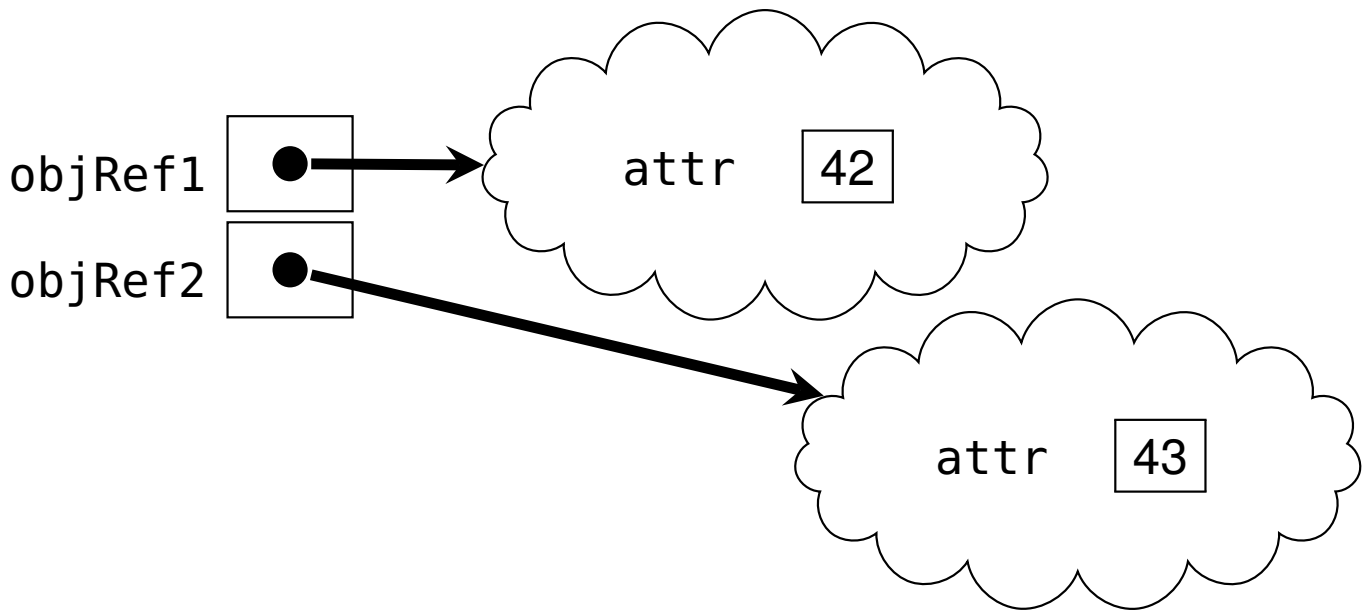
Klass och instans

```
scala> class C { var attr = 42 }

scala> val objRef1 = new C

scala> val objRef2 = new C
```



Klass och instans

```
scala> class C { var attr = 42 }

scala> val objRef1 = new C

scala> val objRef2 = new C

scala> objRef2.attr = 43
```



Klassdeklarationer och instansiering

- Syntax för deklaration av klass:

```
class Klassnamn(parametrar){ medlemmar }
```

- Exempel **deklaration**:

```
class Klassnamn(val attribut1: Int, attribut2: String){
  val attribut3: Double = 42.0              //publikt oföränderligt attribut
  private var attribut4: Boolean = false    //privat medlem syns inte utåt
  def metod(parameter: Int) = parameter + 1 //funktion i klass kallas metod
  lazy val attr4 = Vector.fill(100000)(42.0)     //fördröjd initialisering
}
```

- Parametrar initialiseras med de argument som ges vid **new**.
- Exempel **instansiering** med argument för initialisering av klassparametrar:

```
val instansReferens = new Klassnamn(42, "hej")
```

- Attribut blir **publika** (alltså synliga utåt) om inte modifieraren **private** anges.
- Parametrar som inte föregås av modifierare (t.ex. private val, val, var) blir **attribut** som är: **private[this] val** och bara synliga i **denna** instans.

Exempel: Klassen Complex i Scala

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

```
1  scala> val c1 = new Complex(3, 4)
2  c1: Complex = 3.0 + 4.0i
3
4  scala> val polarForm = (c1.r, c1.fi)
5  polarForm: (Double, Double) = (5.0,0.6435011087932844)
6
7  scala> val c2 = new Complex(1, 2)
8  c2: Complex = 1.0 + 2.0i
9
10 scala> c1 + c2
11 res0: Complex = 4.0 + 6.0i
```

Exempel: Principen om enhetlig access

```
class Complex(val re: Double, val im: Double){
  val r  = math.hypot(re, im)
  val fi = math.atan2(re, im)
  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.)

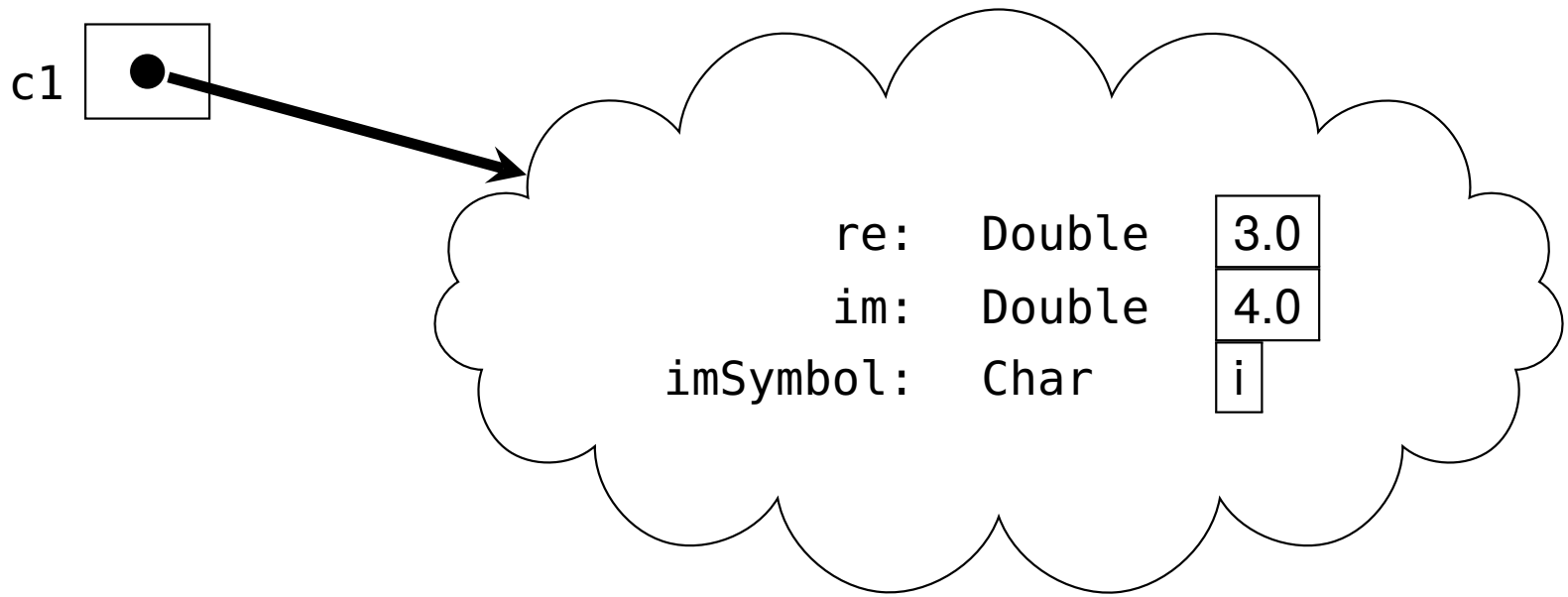
Olika sätt att skapa instanser

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)
```



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)
}
```

Instansiera **inte direkt**, utan **indirekt** genom användning av **fabriksmetoder**:

```
1  scala> import MyFactory._
2
3  scala> createComplex(3, 4)
4  res0: Complex = 3.0 + 4.0i
5
6  scala> createReal(42)
7  res1: Complex = 42.0 + 0.0i
8
9  scala> createImaginary(-1)
10 res2: Complex = 0.0 + -1.0i
```

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(re, im)
  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 kan förhindra direkt instansiering

- Ett **kompanjonsobjekt** är ett objekt som ligger i samma kodfil som en klass och 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**.

```
class Complex private (val re: Double, val im: Double){
  def r  = math.hypot(re, im)
  def fi = math.atan2(re, im)
  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)
  def real(re: Double)              = new Complex(re, 0)
  def imag(im: Double)              = new Complex(0, im)
}
```

- Fabriksmetoder i kompanjonsobjektet ovan och privat kontstruktor gör att vi **enbart** tillåter **indirekt instansiering**.

Användning av kompanjonsobjekt med fabriksmetoder för indirekt instansiering

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(re, im)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
}
```

```
1  scala> new Complex()
2  res0: Complex = 0.0 + 0.0i
3
4  scala> new Complex(re = 42)  //anrop med namngivet argument
5  res1: Complex = 42.0 + 0.0i
6
7  scala> new Complex(im = -1)
8  res2: Complex = 0.0 + -1.0i
9
10 scala> new Complex(1)
11 res3: Complex = 1.0 + 0.0i
```

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(re, im)
  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()
}
```

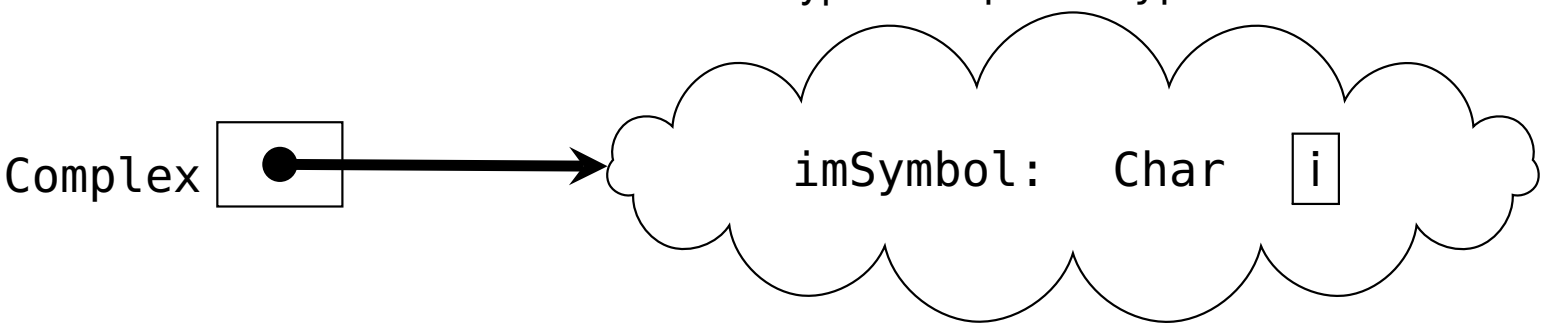
Medlemmar som bara behövs i en enda upplaga

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(re, im)
  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()
}
```

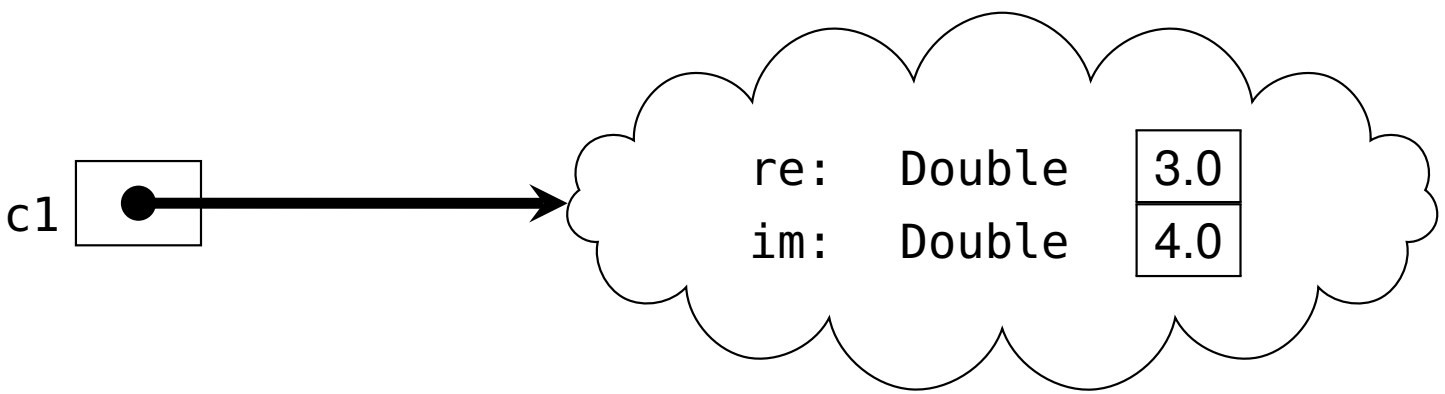
Medlemmar i singelobjekt är statiskt allokerade

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** allokerad instans:

```
scala> val c1 = Complex(3, 4)   // dynamisk allokering på heapen som växer
```



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
```

Minneshantering

Vad är en konstruktor?

- En **konstruktor** är den kod som exekveras när klasser instansieras med `new`.
- Konstruktorn skapar nya objekt i minnet under körning när den anropas.
- I Scala **genererar** kompilatorn en **primärkonstruktor** med maskinkod som initialiserar alla attribut baserat på klassparamtetrar och `val`- och `var`-deklarationer.
- I Java **måste** man **själv** skriva alla konstruktorer med speciell syntax och göra alla initialiseringar själv. Man kan ha många olika alternativa konstruktorer i Java.
- I Scala **kan** man också skriva egna **alternativa konstrukturer** med speciell syntax, men det är **ovanligt**, eftersom man har möjligheten med fabriksmetoder i kompanjonsobjekt och default-argument (saknas i Java).

Hjälpkonstruktorer i Scala

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){
  def this(x: Int, y: Int) = this(x, y, 0)   //anrop av primärkonstruktorn
  def this(x: Int) = this(x, 0)              //anrop av hjälpkonstruktor
  override def toString =
    if (z == 0) s"Point($x,$y)" else s"Point($x,$y,$z)"
}
```

Genom att känna till hur hjälpkonstruktorer fungerar i Scala, blir det lättare att begripa konstruktorer i Java.

Användning av hjälpkonstruktor

```
scala> val p1 = new Point(1)
p1: Point = Point(1,0)

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

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

Men man gör **mycket oftare** så här i Scala:

```
class Point(val x: Int, val y: Int = 0, val z: Int = 0){
  override def toString =
    if (z == 0) s"Point($x,$y)" else s"Point($x,$y,$z)"
}
```

Eller använder en fabriksmetod i kompanjonsobjekt.
Eller ännu hellre en case-klass...

Vad gör skräpsamlaren?

- Scala och Java är båda programmeringsspråk som förutsätter en körmiljö med **automatisk skräpsamling** (eng. *garbage collection*).
- **Skräpsamlaren** (eng. *the garbage colector*) är ett program som automatiskt körs i bakgrunden då och då och **städar minnet** genom att frigöra den plats som upptas av **objekt som inte längre används**.
- JVM:en bestämmer själv när skräpsamlaren ska jobba och programmeraren har ingen kontroll över detta.
- Den stora **fördelen** med automatisk skärpsamling är att man slipper bry sig om det svåra och felbenägna arbetet att **avallokera** minne.
- **Nackdelen** är att man inte kan styra exakt hur och när skräpsamlingen ska ske och man kan därmed inte bestämma när processorn ska belastas med minneshanteringen. Detta är normalt inget problem, utom i vissa tidskritiska realtidssystem med hårda minnesbegränsningar och svarstidskrav.
- I språk utan automatisk skräpsamling, t.ex. C++, måste man ta hand om destruktion av objekt och skriva egna s.k. **destruktorer**.

Referens saknas: `null`

Referens saknas: `null`

- I Java och många andra språk använder man ofta literalen **`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

```
1  scala> class Gurka(val vikt: Int)
2  defined class Gurka
3
4  scala> var g: Gurka = null        // ingen instans allokerad än
5  g: Gurka = null
6
7  scala> g.vikt
8  java.lang.NullPointerException
9
10 scala> g = new Gurka(42)          // instansen allokeras
11 g: Gurka = Gurka@1ec7d8b3
12
13 scala> g.vikt
14 res0: Int = 42
15
16 scala> g = null         // instansen kommer att destrueras av skräpsamlaren
```

- Scala har **`null`** av kompabilitetsskäl, men det är brukligt att **endast** använda **`null`** om man anropar Java-kod.
- Scala erbjuder smidiga `Option`, `Some` och `None` för säker hantering av saknade värden; mer om detta i vecka 8.

Klasser i Java

Typisk utformning av Java-klass

Typisk ”anatomi” av en Java-klass:

```
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
}
```

www.oracle.com/technetwork/java/codeconventions-141855.html#1852

Java-exempel: Klassen JComplex

```
public class JComplex {            // man kan ej deklarera klassparametrar i Java
    private double re;             // initialiseras i konstruktorn nedan
    private double im;             // initialiseras i konstruktorn nedan
    public  char   imSymbol = 'i'; // publikt attribut (inte vanligt i Java)

    public JComplex(double real, double imag){  // konstruktor, anropas vid new
        re = real;
        im = imag;
    }

    public double getRe(){  // en så kallad "getter" som ger attributvärdet, förhindra förändring av re
        return re;
    }

    public double getIm(){ return im; }  // ej bruklig formattering i Java, så metoder blir minst 3 rader

    public double getR(){
        return Math.hypot(re, im);       // Math med stort M i Java
    }

    public double getFi(){
        return Math.atan2(re, im);
    }

    public JComplex add(JComplex other){ // Javametodnamn får ej ha operatortecken t.ex. +, därav namnet add
        return new JComplex(re + other.getRe(), im + other.getIm());
    }

    @Override public String toString(){
        return re + " + " + im + imSymbol;
    }
}
```

Exempel: Använda JComplex i Scala-kod

```
$ javac JComplex.java
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
Type in expressions for evaluation. Or try :help.

scala> val jc1 = new 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 = new 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 **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)`

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 kan göra detta inuti klassen `JComplex`:

```
public static char imSymbol = 'i';
```

- Effekten blir den samma som ett singelobjekt i Scala:
  - Alla statiska medlemmar i en Java-klass allokeras automatisk och hamnar i en egen singulär ”klassinstans” som existerar oberoende av de dynamiska instanserna.
  - De statiska medlemmarna accessas med punktnotation genom klassnamnet:

```
JComplex.imSymbol = 'j';
```

Referensen `this`

Referensen this

- Nyckelordet **this** ger en referens till den aktuella instansen.

```
scala> class Gurka(var vikt: Int){def jagSjälv = this}
defined class Gurka

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

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

scala> g.jagSjälv.vikt
res1: Int = 42

scala> g.jagSjälv.jagSjälv.vikt
res2: Int = 42
```

- Referensen **this** används ofta för att komma runt ”namnkrockar” där en variabler med samma namn gör så att den ena variabeln inte syns.

Getters och setters

Getters och setters i Java

- I Java finns inget motsvarande nyckelord **`val`** som garanterar oföränderliga attributreferenser.[12]
- Därför gör man i Java nästan alltid attribut **privata** för att förhindra att de ändras på ett okontrollerat sätt.
- Java följer inte principen om enhetlig access: åtkomst av metoder och variabler sker med olika syntax.
- Därför är det normala i Java att införa metoder som kallas **getters** och **setters**, som används för att **indirekt** läsa och uppdatera **attribut**.
- Dessa metoder känns igen genom Java-konventionen att de heter något som börjar med **get** respektive **set**.
- Med indirekt access av attribut kan man i Java åstadkomma **flexibilitet**, så att klassimplementationen 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.
- Om klassen **inte** erbjuder en **setter** för privata attribut kan man åstadkomma **oföränderliga** datastrukturer där attributreferenserna inte förändras efter allokering.

[12] Det finns visserligen **`final`** men det är annorlunda som vi ska se senare.

Java-exempel: Klassen JPerson

Indirekt access av **privata** attribut:

```
public class JPerson {
    private String name;
    private int age;

    public JPerson(String name){
        //namnkrock fixas med this
        this.name = name;
        age = 0;
    }

    public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }

    public void setAge(int age){
        this.age = age;
    }
}
```

```
$ javac JPerson.java
$ scala
Welcome to Scala 2.11.8 (Java HotSpot
Type in expressions for evaluation. O

scala> val p = new JPerson("Björn")
p: JPerson = JPerson@7e774085

scala> p.getAge
res0: Int = 0

scala> p.setAge(42)

scala> p.getAge
res1: Int = 42

scala> p.age
error:
value age is not a member of JPerson
```

Motsvarande JPerson men 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**?
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** i Scala är en **procedur som har ett namn som 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) myPrivateAge = a else myPrivateAge = 0
}
```

```
1  scala> val p = new Person("Björn")
2  p: Person = Person@28ac3dc3
3
4  scala> p.age = 42      // najs syntax om getter parad med setter enl ovan
5  p.age: Int = 42
6
7  scala> p.age = -1      // nu förhindras negativ ålder
8  p.age: Int = 0
```

Likhet

Referenslikhet eller strukturlikhet?

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*) där två referenser anses lika om de refererar till **samma instans** i minnet.
- **Strukturlikhet** (eng. *structural equality*) där två referenser anses lika om de refererar till instanser 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 **strukturlikhet** genom metoden ==
- I Java är det annorlunda: symbolen == är ingen metod i Java utan specialsyntax som testar referenslikhet mellan instanser, medan metoden `equals` kan överskuggas med valfri likhetstest.

Exempel: referenslikhet och strukturlikhet

I Scalas standardbibliotek har man överskuggat `equals` så att metoden == ger test av **strukturlikhet** mellan instanser:

```
1  scala> val v1 = Vector(1,2,3)
2  v1: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
3
4  scala> val v2 = Vector(1,2,3)
5  v2: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
6
7  scala> v1 eq v2                 //referenslikhetstest: olika instanser
8  res0: Boolean = false
9
10 scala> v1 ne v2
11 res1: Boolean = true
12
13 scala> v1 == v2                 //strukturlikhetstest: samma innehåll
14 res2: Boolean = true
15
16 scala> v1 != v2
17 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 och likhet

Varför case-klass?

Med case-klasser får du mycket ”godis på köpet”:

- Skapa **oföränderlig datastruktur** med få kodrader.
- Klassparametrar blir automatiskt publika **`val`**-attribut (inte `private[this]` som i vanliga klasser).
- Du får en automatisk **toString** som ger klassens namn och värdet av alla **`val`**-attribut som ges av klassparametrarna.
- Du slipper skriva **`new`** eftersom du får ett automatiskt kompanjonsobjekt med en fabriksmetod `apply` för indirekt instansiering där alla klassparametrarnas **`val`**-attribut initialiseras.
- Metoden == ger **strukturlikhet** (och inte referenslikhet).

Likhet och case-klasser

Metoden `equals` är i case-klasser automatiskt överskuggad så att metoden == ger test av strukturlikhet.

```
1  scala> case class Gurka(vikt: Int)
2
3  scala> val g1 = Gurka(42)
4  g1: Gurka = Gurka(42)
5
6  scala> val g2 = Gurka(42)
7  g2: Gurka = Gurka(42)
8
9  scala> g1 eq g2          // olika instanser
10 res0: Boolean = false
11
12 scala> g1 == g2          // samma innehåll!
13 res1: Boolean = true
```

Sammanfattning case-klass-godis

Minnesschecklista med ”godis” i **`case`**-klasser så här långt:

1. klassparametrar blir **`val`**-attribut
2. najs toString
3. slipper skriva **`new`**
4. == ger strukturlikhet

   ...

Men vi har inte sett allt godis än...
Vecka 8: Mönstermatchning.

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**.
- 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
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  at .sprängsSnart$lzycompute(<console>:11)
  at .sprängsSnart(<console>:11)
```

Exempel: ofärdig kod

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

```
scala> Person("Björn", 49).ärTonåring
res23: Boolean = false

scala> Person("Sandra", 35).ärUng
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  at Person.ärGammal(<console>:12)
  at Person.ärUng(<console>:13)
```

Klass-specifikationer

Specifikationer av klasser i Scala

- Specifikationer av klasser innehåller information som *den som ska implementera* klassen behöver veta.
- Specifikationer innehåller liknande information som dokumentationen av klassen (scaladoc), som beskriver vad *användaren* av klassen behöver veta.

Specification **Person**

```
/** Encapsulate immutable data about a Person: name and age. */
case class Person(name: String, age: Int = 0){
  /** Tests whether this Person is more than 17 years old. */
  def isAdult: Boolean = ???
}
```

- Specifikationer av Scala-klasser utgör i denna kurs ofullständig kod som kan kompileras utan fel.
- Saknade implementationer markeras med ???
- **Dokumentationskommentarer** utgör **krav** på implementationen.

Specifikationer av klasser och objekt

Specification **MutablePerson**

```
/** Encapsulates mutable data about a person. */
class MutablePerson(initName: String, initAge: Int){
  /** The name of the person. */
  def getName: String = ???

  /** Update the name of the Person */
  def setName(name: String): Unit = ???

  /** The age of this person. */
  def getAge: Int = ???

  /** Update the age of this Person */
  def setAge(age: Int): Unit = ???

  /** Tests whether this Person is more than 17 years old. */
  def isAdult: Boolean = ???

  /** A string representation of this Person, e.g.: Person(Robin, 25) */
  override def toString: String = ???
}
object MutablePerson {
  /** Creates a new MutablePerson with default age. */
  def apply(name: String): MutablePerson = ???
}
```

Specifikationer av Java-klasser

- Specificerar signaturer för konstruktorer och metoder.
- Kommentarerna utgör krav på implementationen.
- Används flitigt på extentor i EDA016, EDA011, EDA017...
- Javaklass-specifikationerna **saknar implementationer** och behöver kompletteras med metodkroppar och klassrubriker innan de kan kompileras.

`class Person`

```
/** Skapar en person med namnet name och åldern age. */
Person(String name, int age);

/** Ger en sträng med denna persons namn. */
String getName();

/** Ändrar denna persons ålder. */
void setAge(int age);

/** Anger åldersgränsen för när man blir myndig. */
static int adultLimit = 18;
```

Grumligt-lådan

Grumligt-lådan

Veckans skörd av lappar i ”grumligtlådan”:

12 case class
8 Map och map
8 private, public
5 override
3 toString
3 kompanjonsobjekt
2 typparametrar [Int]
2 Specialfall, sekvensalgoritmer

1 lab pirates
1 Hur ska jag träna datastrukturer?
1 underscore i olika sammanhang
1 Stränginterpolator s"\$x"
1 heap
1 Assume
1 Mutable / immutable
1 Vad är en typ och hur kan klass bli en typ?
1 tomma parenteser ()
1 skillnad mellan argument och parameter
1 pseudokod
1 Hur hitta buggar?
1 w04 datastrukturer
1 skillnad på olika parenteser \{[(
1 terminologi allmänt
1 uppdatering av variabler som refererar till varandra
1 val, lazy val, var
1 när använda terminal, editor, IDE?
1 Formattering/upplägg av kod, indrag, var ska objekt vara?
— Bättre instruktioner på labbarna
— bra med sammanfattning på slutet av föreläsningarna

Veckans uppgifter

Övning: `classes`

- Kunna deklarera klasser med klassparametrar.
- Kunna skapa objekt med **`new`** och konstruktorargument.
- Förstå innebörden av referensvariabler och värdet **`null`**.
- Förstå innebörden av begreppen instans och referenslikhet.
- Kunna använda nyckelordet **`private`** för att styra synlighet i primärkonstruktor.
- Förstå i vilka sammanhang man kan ha nytta av en privat konstruktor.
- Kunna implementera en klass utifrån en specikation.
- Förstå skillnaden mellan referenslikhet och strukturlikhet.
- Känna till hur case-klasser hanterar likhet.
- Förstå nyttan med att möjliggöra framtida förändring av attributrepresentation.
- Känna till begreppen getters och setters.
- Känna till accessregler för kompanjonsobjekt.
- Känna till skillnaden mellan == och eq, samt != versus ne.

Laboration: `turtlegraphics`

- Kunna skapa egna klasser.
- Förstå skillnaden mellan klasser och objekt.
- Förstå skillnaden mellan muterbara och omuterbara objekt.
- Förstå hur ett objekt kan innehålla referenser till objekt av andra klasser, och varför detta kan vara användbart.
- Träna på att fatta beslut om vilka datatyper som bäst passar en viss tillämpning.

Grupplaboration w07

- Läs kapitel 0.1.2 på sidan 13
- Skumläs laboration `turtlerace-team`
- Träffas i din samarbetsgrupp på en fika och börja planera arbetet.

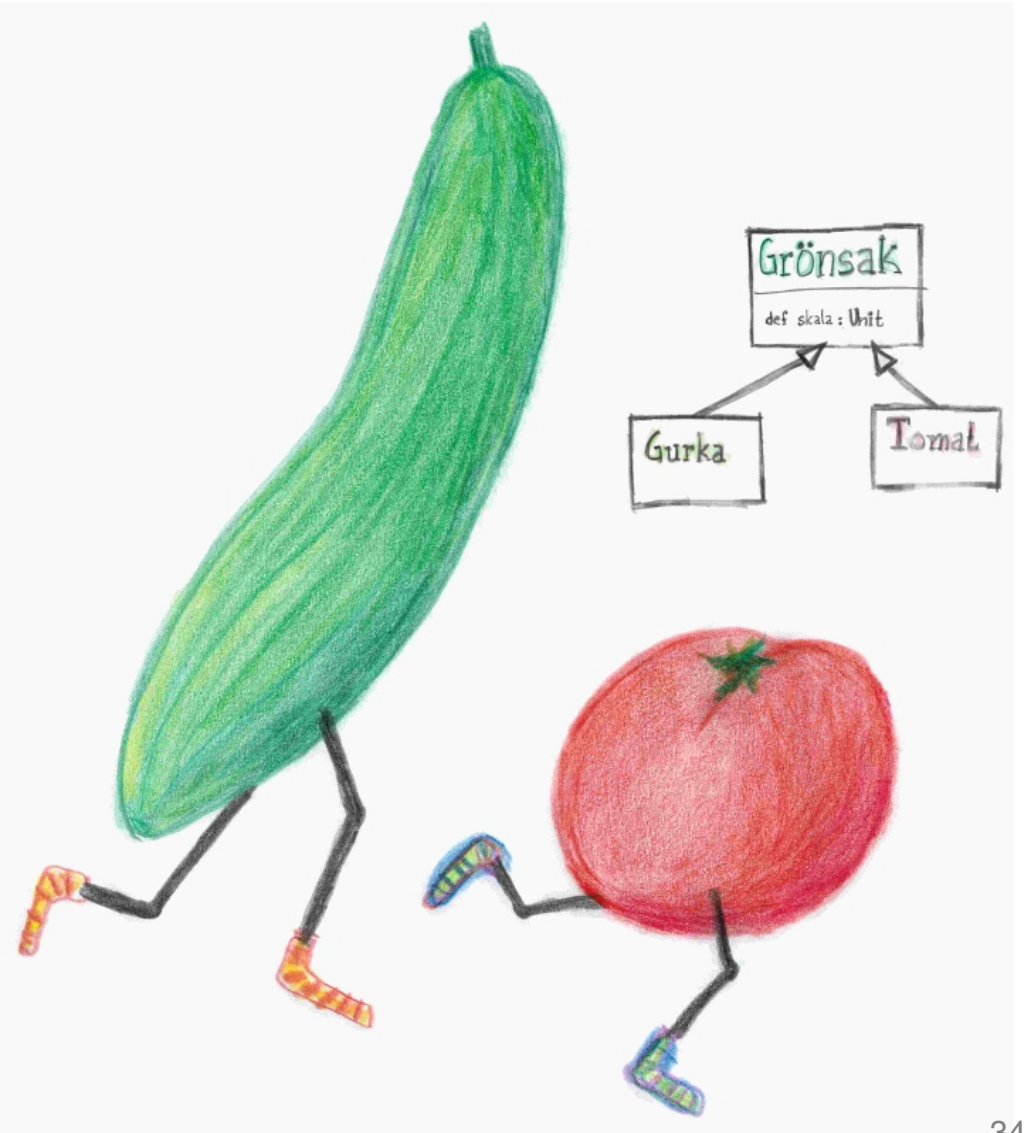
7 Arv

- Vad är arv?
- Överskuggingsregler
- `super`
- Trait eller abstrakt klass?

Vad är arv?

Vad är arv?

Med arv kan man beskriva 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 generiska samlingar kan ges en mer specifik elementtyp.
  - Det räcker att man vet bastypen för att kunna anropa gemensamma metoder på alla element i samlingen.

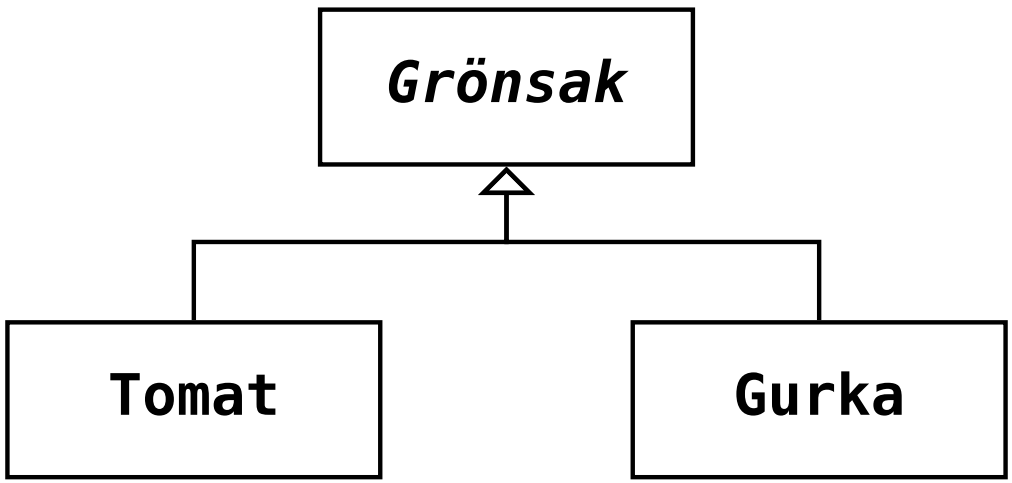
Behovet av gemensam bastyp

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

scala> class Tomat(val vikt: Int)

scala> val gurkor = Vector(new Gurka(200), new Gurka(300))
gurkor: scala.collection.immutable.Vector[Gurka] =
  Vector(Gurka@60856961, Gurka@2fd953a6)

scala> gurkor.map(_.vikt)
res0: scala.collection.immutable.Vector[Int] = Vector(200, 300)

scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
grönsaker: scala.collection.immutable.Vector[Object] =
  Vector(Gurka@669253b7, Tomat@5305c37d)

scala> grönsaker.map(_.vikt)
<console>:15: error: value vikt is not a member of Object
       grönsaker.map(_.vikt)
```

Hur ordna en mer specifik typ än `Vector[Object]`? → Skapa en **bastyp**!

Skapa en gemensam bastyp

Typen ***Grönsak*** är en **bastyp** i nedan arvshierarki:



Pilen ⇑ betecknar **arv** och utläses ”**är en**”

Typerna `Tomat` och `Gurka` är **subtyper** till den **abstrakta** typen `Grönsak`.

Skapa en gemensam bastyp med `trait` och `extends`

Med **`trait`** `Grönsak` kan klasserna `Gurka` och `Tomat` få en gemensam **bastyp** genom att båda **subtyperna** gör **`extends`** `Grönsak`:

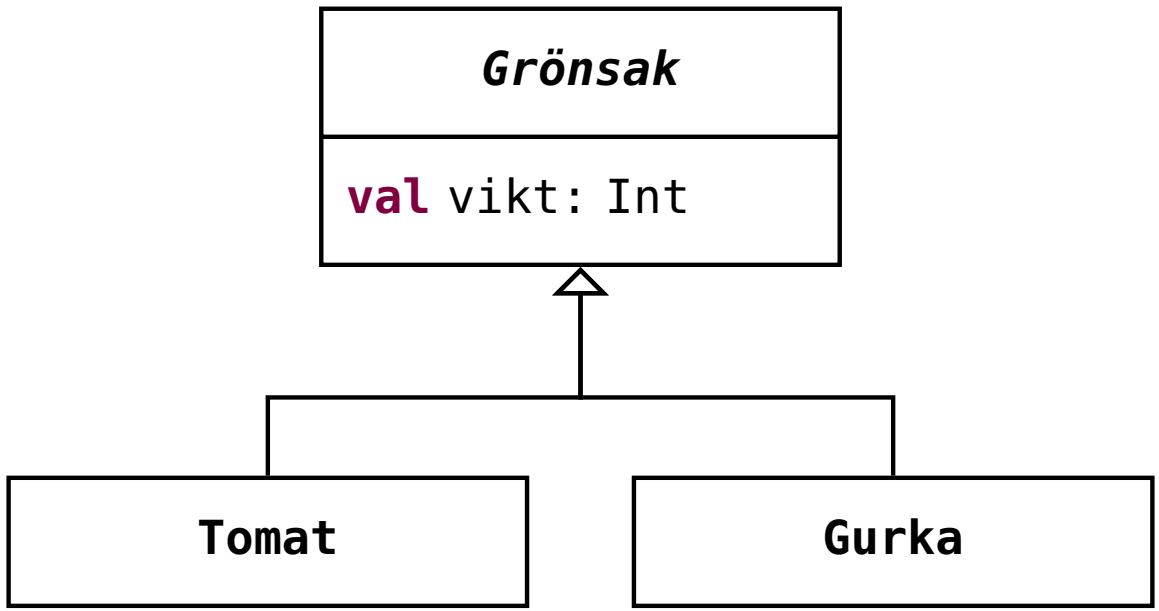
```
1  scala> trait Grönsak
2
3  scala> class Gurka(val vikt: Int) extends Grönsak
4
5  scala> class Tomat(val vikt: Int) extends Grönsak
6
7  scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
8  grönsaker: scala.collection.immutable.Vector[Grönsak] =
9    Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)
```

Men det är fortfarande inte som vi vill ha det:

```
scala> grönsaker.map(_.vikt)
<console>:15: error: value vikt is not a member of Grönsak
       grönsaker.map(_.vikt)
```

En gemensam bastyp med gemensamma delar

Placera gemensamma medlemmar i bastypen:



- Alla grönsaker har attributet `val vikt`.
- Det specifika värdet på vikten definieras **inte** i bastypen.
- Medlemen `vikt` kallas **abstrakt** eftersom den **saknar implementation**.

Placera gemensamma delar i bastypen

Vi inkluderar det gemensamma attributet `val vikt` som en **abstrakt medlem** i bastypen:

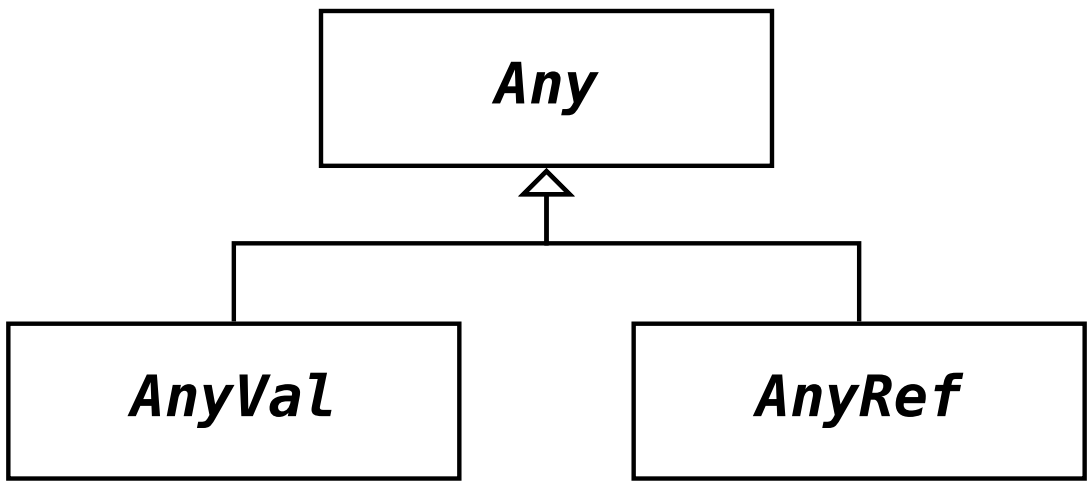
```
trait Grönsak { val vikt: Int }

class Gurka(val vikt: Int) extends Grönsak

class Tomat(val vikt: Int) extends Grönsak
```

Nu vet kompilatorn att alla grönsaker har en vikt:

```
1  scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
2  grönsaker: scala.collection.immutable.Vector[Grönsak] =
3    Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)
4
5  scala> grönsaker.map(_.vikt)
6  res0: scala.collection.immutable.Vector[Int] = Vector(200, 42)
```

Scalas typhierarki och typen `Object`

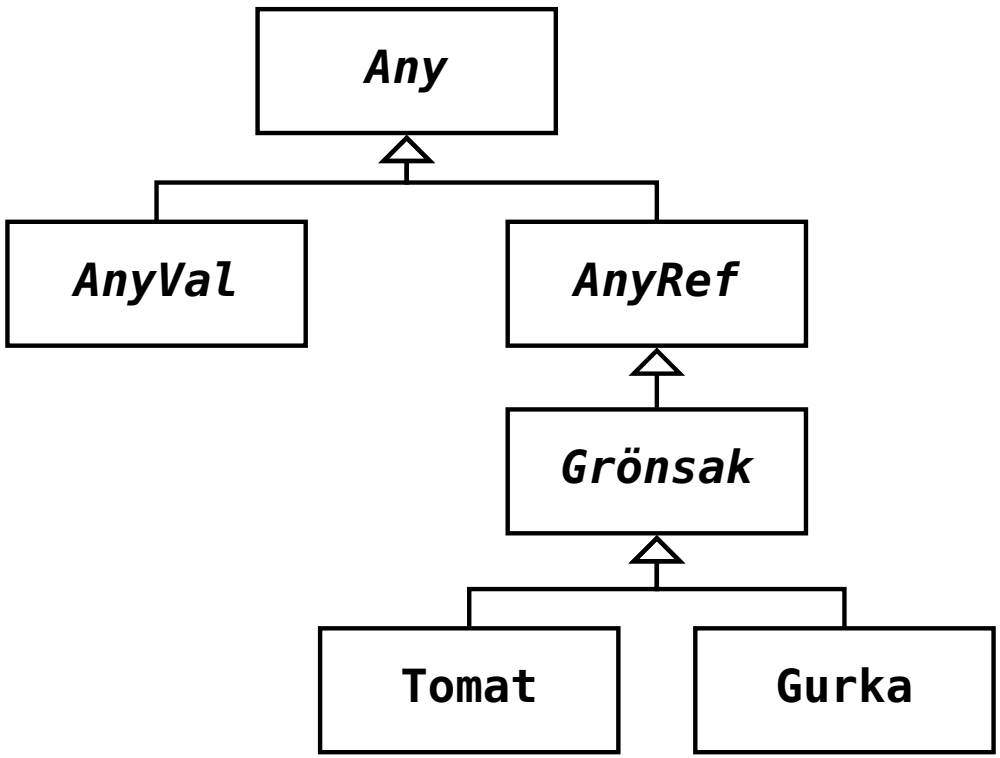
Den översta delen av typhierarkin i Scala:



- 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 kräver (direkt eller indirekt) konstruktion med **`new`**.
- `AnyRef` motsvaras av **`java.lang.Object`** i JVM.

Implicita supertyper till dina egna klasser

Alla dina egna typer ingår underförstått i Scalas typhierarki:



Vad är en trait?

- **Trait** betyder **egenskap** på engelska.
- En trait liknar en klass, **men** speciella regler gäller:
  - den **kan** innehålla delar som **saknar implementation**
  - den **kan mixas** med flera andra traits så att olika koddelar kan återanvändas på flexibla sätt.
  - den **kan inte** instansieras direkt som den är; den måste återanvändas genom arv.
  - den **kan inte** ha klassparametrar eller konstruktorer
- Jämförelse med Java:
  - En Scala-trait liknar det som i Java kallas `interface`, men man kan göra mer med Scala-traits: färre begränsningar, fler abstraktionsmöjligheter.
  - En Scala-trait med enbart abstrakta medlemmar kompileras till bytekod i JVM:en som kan användas från Java-kod precis som ett Java-interface.

Vad används en trait till?

En **`trait`** används 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
```

```
1  scala> val a = new Subtyp1
2  a: Subtyp1 = Subtyp1@51016012
3
4  scala> a.x
5  res0: Int = 42
6
7  scala> a.y
8  res1: Int = 43
9
10 scala> a.z
11 <console>:15: error: value z is not a member of Subtyp1
12
13 scala> new Bastyp
14 <console>:13: error: trait Bastyp is abstract; 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
```

```
scala> val a = new A
a: A = A@5faeada1

scala> val b = new B
b: B = B@cb51256

scala> val xs = Vector(a,b)
xs: scala.collection.immutable.Vector[X] = Vector(A@5faeada1, B@cb51256)

scala> xs.map(_.x)
res0: scala.collection.immutable.Vector[Int] = Vector(42, 43)

scala> class Y { val y: Int }
  error: class Y needs to be abstract, since value y is not defined

scala> trait Z(x: Int)
  error: traits or objects may not have parameters
```

Terminologi och nyckelord

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 kan ha abstrakta medlemmar; kan ej instansieras
konkret typ	en typ som ej har abstrakta medlemmar; kan instansieras
`class`	en klass är en konkret typ som **ej kan ha abstrakta medlemmar**
`abstract class`	en klass är en abstrakt typ som **kan ha parametrar**
`trait`	är en abstrakt typ som **ej kan ha parametrar** men **kan mixas in**
`extends`	står före en supertyp, indikerar arv
`override`	en medlem överskuggar (byter ut) en medlem i en superttyp
`protected`	gör en medlem synlig i subtyper till denna typ (jmf `private`)
`final gurka`	gör medlemen gurka final: förhindrar överskuggning
`final class`	gör klassen final: förhindrar vidare subtypning
`sealed trait`	förseglad trait: bara de direkta subtyperna i denna kodfil
`super.gurka`	refererar till supertypens medlem `gurka` (jmf `this`)

Abstrakta och konkreta medlemmar

```
1  object exempelVego1 {
2
3    trait Grönsak {
4      def skala(): Unit
5      var vikt: Double
6      val namn: String
7      var ärSkalad: Boolean = false
8      override def toString = s"$namn ${if (!ärSkalad) "o" else ""}skalad $vikt g"
9    }
10
11   class Gurka(var vikt: Double) extends Grönsak {
12     val namn = "gurka"
13     def skala(): Unit = if (!ärSkalad) {
14       println("Gurkan skalas med skalare.")
15       vikt = 0.99 * vikt
16       ärSkalad = true
17     }
18   }
19
20   class Tomat(var vikt: Double) extends Grönsak {
21     val namn = "tomat"
22     def skala(): Unit = if (!ärSkalad) {
23       println("Tomaten skalas genom skållning.")
24       vikt = 0.99 * vikt
25       ärSkalad = true
26     }
27   }
28 }
```

Undvika kodduplicering med hjälp av arv

```
1  object exempelVego2 {
2
3    trait Grönsak {  // innehåller alla gemensamma delar; hjälper oss undvika upprepning
4      val skalningsmetod: String
5      val skalfaktor = 0.99
6      def skala(): Unit = if (!ärSkalad) {
7        println(skalningsmetod)
8        vikt = skalfaktor * vikt
9        ärSkalad = true
10     }
11     var vikt: Double
12     val namn: String
13     var ärSkalad: Boolean = false
14     override def toString = s"$namn ${if (!ärSkalad) "o" else ""}skalad $vikt g"
15   }
16
17   class Gurka(var vikt: Double) extends Grönsak { // bara det som är speciellt för gurkor
18     val namn = "gurka"
19     val skalningsmetod = "Skalas med skalare."
20   }
21
22   class Tomat(var vikt: Double) extends Grönsak { // bara det som är speciellt för tomater
23     val namn = "tomat"
24     val skalningsmetod = "Skållas."
25   }
26 }
```

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:
  `DRY == "Don't Repeat Yourself!"`
- Men det kan finnas tillfällen när kodduplicering faktiskt är att föredra: t.ex. om man vill att olika delar av koden ska vara helt oberoende av varandra.

Överskuggning

```
1  object exempelVego3 {
2
3    trait Grönsak {
4      val skalningsmetod: String
5      val skalfaktor = 0.99
6      def skala(): Unit = if (!ärSkalad) {
7        println(skalningsmetod)
8        vikt = skalfaktor * vikt
9        ärSkalad = true
10     }
11     var vikt: Double
12     val namn: String
13     var ärSkalad: Boolean = false
14     override def toString = s"$namn ${if (!ärSkalad) "o" else ""}skalad $vikt g"
15   }
16
17   class Gurka(var vikt: Double) extends Grönsak {
18     val namn = "gurka"
19     val skalningsmetod = "Skalas med skalare."
20   }
21
22   class Tomat(var vikt: Double) extends Grönsak {
23     val namn = "tomat" //nyckelordet override behövs ej vid abstrakt medlem, men tillåtet:
24     override val skalningsmetod = "Skållas."
25 //  override val skalningmetod = "Skållas." //kompilatorn hittar felet (stavfel, s saknas)
26     override val skalfaktor = 0.95  //överskuggning: override måste anges vid ny impl.
27   }
28 }
```

En final medlem kan ej överskuggas

```
1  object exempelVego4 {
2
3    trait Grönsak {
4      val skalningsmetod: String
5      final val skalfaktor = 0.99              // en final medlem kan ej överskuggas
6      def skala(): Unit = if (!ärSkalad) {
7        println(skalningsmetod)
8        vikt = skalfaktor * vikt
9        ärSkalad = true
10     }
11     var vikt: Double
12     val namn: String
13     var ärSkalad: Boolean = false
14     override def toString = s"$namn ${if (!ärSkalad) "o" else ""}skalad $vikt g"
15   }
16
17   class Gurka(var vikt: Double) extends Grönsak {
18     val namn = "gurka"
19     val skalningsmetod = "Skalas med skalare."
20   }
21
22   class Tomat(var vikt: Double) extends Grönsak {
23     val namn = "tomat"
24     val skalningsmetod = "Skållas."
25 //  override val skalfaktor = 0.95 // KOMPILERINGSFEL: "cannot override final member"
26   }
27 }
```

Protected ger synlighet begränsad till subtyper

Filnamnsregler och -konventioner

- Java
  - I Java får man bara ha **en enda** publik klass per kodfil.
  - I Java måste kodfilen ha **samma namn** som den publika klassen, t.ex. `KlassensNamn.java`
- Scala
  - I Scala får man ha **många** klasser/traits/singelobjekt i samma kodfil.
  - I Scala får man döpa kodfilerna **oberoende** av deras innehåll. Dessa **konventioner** används:
    - 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. `bastypensNamn.scala`

Exempel: shapes1.scala

Typisk scala-kod: En trait som bastyp åt flera case-klasser.

```
1  object shapes1 {
2    type Pt = (Double, Double)
3
4    trait Shape {
5      def pos: Pt
6      def move(dx: Double, dy: Double): Shape
7    }
8
9    case class Rectangle(pos: Pt, dxy: Pt) extends Shape {
10     def move(dx: Double, dy: Double): Shape =
11       Rectangle((pos._1 + dx, pos._2 + dy), dxy)
12   }
13
14   case class Circle(pos: Pt, radius: Double) extends Shape {
15     def move(dx: Double, dy: Double): Shape =
16       Circle((pos._1 + dx, pos._2 + dy), radius)
17   }
18
19   case class Triangle(pos: Pt, dxy1: Pt, dxy2: Pt) extends Shape {
20     override def move(dx: Double, dy: Double): Shape =
21       Triangle((pos._1 + dx, pos._2 + dy), dxy1, dxy2)
22   }
23 }
```

Exempel: draw.scala

Två traits som kan användas för att ”koppla ihop” kod och minimera ändringar av befintlig kod:

```
1  trait DrawingWindow {
2    def penTo(pt: (Double, Double)): Unit
3    def drawTo(pt: (Double, Double)): Unit
4  }
5
6  trait CanDraw {
7    def draw(dw: DrawingWindow): Unit
8  }
```

- Traits som använda för att abstrahera implementation och möjliggöra uppfyllandet av ett slags ”kontrakt” om vad som ska finnas kallas **gränssnitt** (eng. *interface*) och är grunden för skapandet av ett flexibelt api.
- Implementationen av de delar vi vill kunna ändra senare placeras i subtyper som inte används direkt av klientkoden.
- Vi visar bara information om vad som erbjuds men inte hur det ser ut ”inuti”.

Exempel: shapes2.scala

Genom att **mixa in** vår **`trait`** `CanDraw` kan en rektangel nu även ritas ut:

```
1  object shapes2 {
2    type Pt = (Double, Double)
3
4    trait Shape {
5      def pos: Pt
6      def move(dx: Double, dy: Double): Shape
7    }
8
9    case class Rectangle(pos: Pt, dxy: Pt) extends Shape with CanDraw { // inmixning
10     override def move(dx: Double, dy: Double): Rectangle =
11       Rectangle((pos._1 + dx, pos._2 + dy), dxy)
12
13     override def draw(dw: DrawingWindow): Unit = { // implementation av draw
14       dw.penTo(pos)
15       dw.drawTo((pos._1 + dxy._1, pos._2))
16       dw.drawTo((pos._1 + dxy._1, pos._2 + dxy._2))
17       dw.drawTo((pos._1, pos._2 + dxy._2))
18       dw.drawTo(pos)
19     }
20   }
21 }
```

Notera: ingen ändring i `Shape`! Vi behöver nu bara ett **`DrawingWindow`**...

Exempel: SimpleDrawingWindow.scala

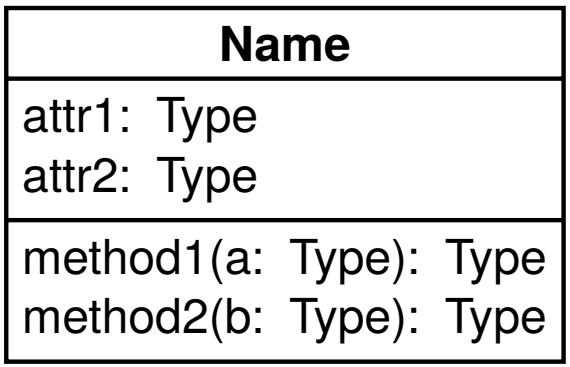
Vi skapar en ny klass som ärver `SimpleWindow`, som **dessutom** även **är ett** `DrawingWindow`, tack vare **inmixning** med nyckelordet **`with`**.
Observera att vi måste skicka vidare klassparametrarna till superklassens konstruktor.

```
1  class SimpleDrawingWindow(title: String = "Untitled", size: (Int, Int) = (640, 400))
2    extends cslib.window.SimpleWindow(size._1, size._2, title)
3      with DrawingWindow {
4
5    def xPos(pt: (Double, Double)): Int = pt._1.round.toInt
6    def yPos(pt: (Double, Double)): Int = pt._2.round.toInt
7
8    override def penTo(pt: (Double, Double)): Unit  = moveTo(xPos(pt), yPos(pt))
9    override def drawTo(pt: (Double, Double)): Unit = lineTo(xPos(pt), yPos(pt))
10 }
```

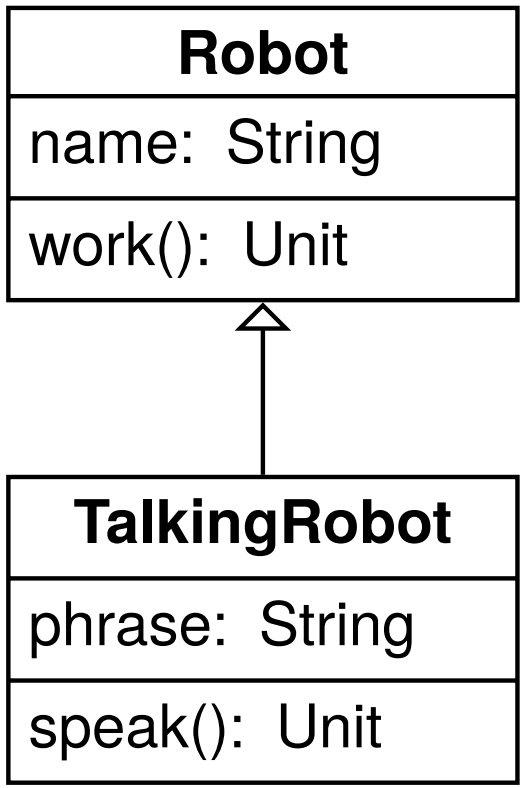
```
1  import shapes2._
2
3  object shapesTest2 {
4    def main(args: Array[String]): Unit = {
5      val sdw = new SimpleDrawingWindow(title="Shapes")
6      val r = Rectangle(pos=(100, 100), dxy=(75, 120))
7      r.draw(sdw)
8      r.move(dx=42, dy=84).draw(sdw)
9    }
10 }
```

Attribut och metoder i UML-diagram

En klass i ett **UML**-diagram kan ha 3 delar:



Ibland utelämnar man typerna.



en.wikipedia.org/wiki/Class_diagram

Överskuggingsregler

Medlemmar, arv och överskuggning

Olika sorters överskuggningsbara medlemmar i klasser och traits 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.
- När man konstruerar ett objektorienterat språk gäller det att man definierar sunda överskuggningsregler vid arv. Detta är förvånansvärt knepigt.
- Singelobjekt kan ej ärvas (och medlemmar i singelobjekt kan därmed ej överskuggas).

Fördjupning: Regler för överskuggning i Scala

En medlem M1 i en supertyp får överskuggas av en medlem M2 i en subtyp, enligt dessa regler:

1. M1 och M2 ska ha samma namn och typerna ska matcha.
2. `def` får bytas ut mot: `def`, `val`, `var`, `lazy val`
3. `val` får bytas ut mot: `val`, och om M1 är abstrakt mot en `lazy val`.
4. `var` får bara bytas ut mot en `var`.
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` eller `private[this]`, men kan vara `private[X]` om M2 också är `private[X]`, eller `private[Y]` om X innehåller Y.
10. Om M1 är `protected` måste även M2 vara det.

Fördjupning: Regler för överskuggning i Java

```
http://docs.oracle.com/javase/tutorial/java/
IandI/override.html
```

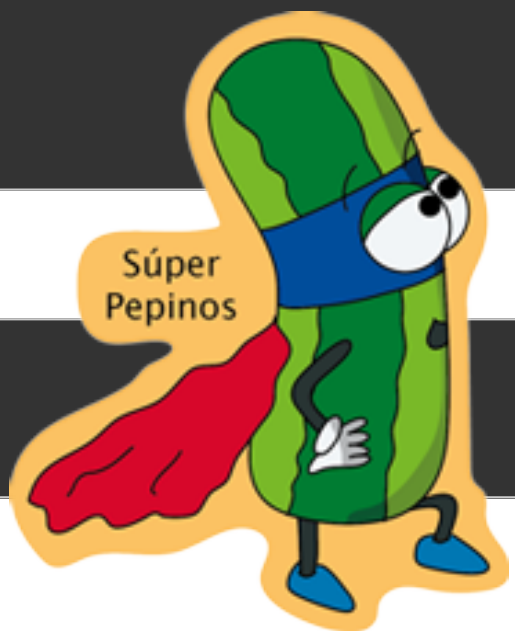
super

Att skilja på mitt och ditt med `super`

```
1  scala> class X { val gurka = "super pepinos" }
2
3  scala> class Y extends X {
4           override val gurka = ":("
5           val sg = super.gurka
6         }
7
8  scala> val y = new Y
9  y: Y = Y@26ba2a48
10
11 scala> y.gurka
12 res0: String = :(
```

Super Pepinos to the rescue:

```
scala> y.sg
res1: String = super pepinos
```



Trait eller abstrakt klass?

Trait eller abstrakt klass?

Använd en **trait** som supertyp 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 bara har abstrakta medlemmar.

Använd en **abstract class** som supertyp om...

- ...du vill ge supertypen en parameter vid konstruktion.
- ...du vill ärva supertypen från klasser skrivna i Java.
- ...du vill minimera vad som behöver omkompileras vid ändringar.