

Programmering, grundkurs pgk

Föreläsningsanteckningar pgk (EDAA45)

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

- 1 Introduktion
- 2 Kodstruktur
- 3 Funktioner, Objekt
- 4 Datastrukturer

1 Introduktion

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

2 Kodstruktur

3 Funktioner, Objekt

4 Datastrukturer

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

<i>W</i>	<i>Modul</i>	<i>Övn</i>	<i>Lab</i>
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	lthopoly-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)
⇒ 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 angreppssätt => 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
- Kompatibilitet: fungerar på många plattformar
- Effektivitet: avancerad & mogen teknik ger snabba program

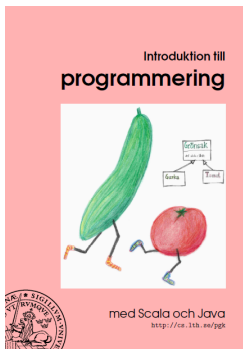
■ Java och Scala fungerar utmärkt tillsammans

- 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!**

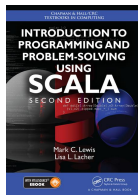
Kurslitteratur



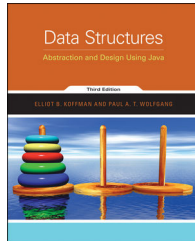
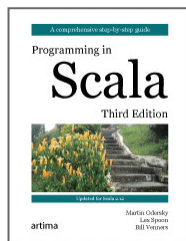
- **Kompendium** med övningar & laborationer, trycks & säljs av inst. på beställning
- Föreläsningsbilder
- Nätresurser enl. länkar

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 kursen¹²
- 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.

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

²D2: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 utvecklarteam 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.
- Gruppindelning 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 labbförberedelser.
- 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**

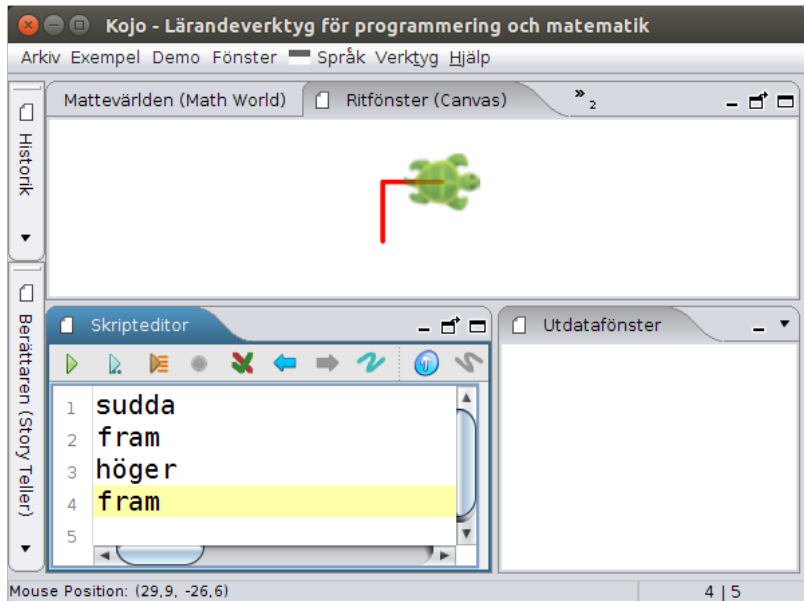
- | | | |
|---|--|--|
| <input type="checkbox"/> sekvens | <input type="checkbox"/> typ | <input type="checkbox"/> enhetsvärdet () |
| <input type="checkbox"/> alternativ | <input type="checkbox"/> tilldelning | <input type="checkbox"/> stränginterpolatorn s |
| <input type="checkbox"/> repetition |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

Om programmering

Programming unplugged: Två frivilliga?



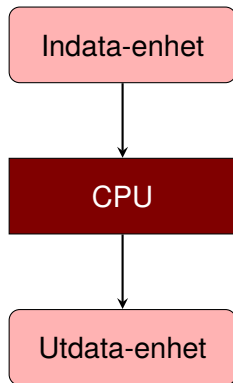
Editera och exekvera ett program



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

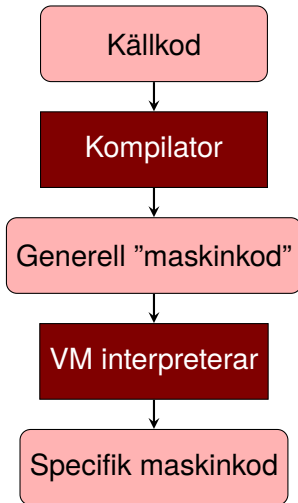


Virtuell maskin (VM) == abstrakt hårdvara

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**
- **Deklaration**: 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!
```

```
// this is Scala
```

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

```
// 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 \Rightarrow **Samarbetskultur**

De enklaste beståndsdelarna: litteraler, uttryck, variabler

Litteraler

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

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.

³Andra 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:

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

Grundtypernas implementation i JVM

Grundtyp i Scala	Antal bitar	Omfång minsta/största värde	primitiv typ i Java & JVM
Byte	8	$-2^7 \dots 2^7 - 1$	byte
Short	16	$-2^{15} \dots 2^{15} - 1$	short
Char	16	$0 \dots 2^{16} - 1$	char
Int	32	$-2^{31} \dots 2^{31} - 1$	int
Long	64	$-2^{63} \dots 2^{63} - 1$	long
Float	32	$\pm 3.4028235 \cdot 10^{38}$	float
Double	64	$\pm 1.7976931348623157 \cdot 10^{308}$	double

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

Variabler

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

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

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

Regler för identifierare

- **Enkel** identifierare: t.ex. gurka2tomat
 - Börja med bokstav
 - ...följt av bokstäver eller siffror
 - Kan även innehålla understreck
- **Operator**-identifierare, t.ex. + :
 - Börjar med ett **operatortecken**, t.ex. + - * / : ? ~ #
 - Kan följas av fler operatortecken
- En identifierare får **inte** vara ett **reserverat ord**, se snabbpreferensen 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 operatoren +
"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):

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

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

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

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

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

- **Moduloräkning** med restoperatörn %

```
1 scala> 41 % 2
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 **avrundningsfel**:

```
1 scala> 0.00000000000001
2 res6: Double = 1.0E-13
3
4 scala> 1E10 + 0.00000000000001
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 slumpstal mellan `0.0` och `0.9999999999999999`

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

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

- Studera dokumentationen här:
<http://www.scala-lang.org/api/current/#scala.math.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. boolsk algebra efter George Boole
- Enkla logiska uttryck: (finns bara två stycken)

true
false



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

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

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:

<code>! (a < b (a == 1 && b == 1))</code>	\iff
<code>! (a < b) && ! (a == 1 && b == 1)</code>	\iff
<code>! (a < b) && (! (a == 1) ! (b == 1))</code>	\iff
<code>a >= b && (a != 1 b != 1)</code>	

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 expanderar 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 deklarerera 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 slumpetal 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.

1 Introduktion

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

3 Funktioner, Objekt

4 Datastrukturer

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.

⁴Vektor 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
<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:

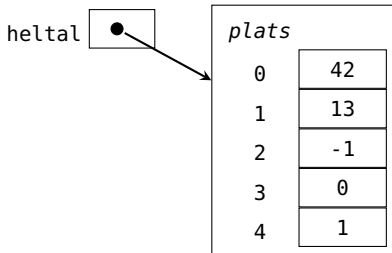
```
scala> 1 to 6
res2: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5, 6)

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:

```
1 $ scalac helloargs.scala  
2 $ scala HelloScalaArgs hej gurka tomat  
3 hej  
4 gurka  
5 tomat
```

Loopa genom argumenten i ett Java-huvudprogram

```
$ gedit HelloJavaArgs.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:

```
1 $ javac HelloJavaArgs.scala
2 $ java HelloJavaArgs hej gurka tomat
3 hej
4 gurka
5 tomat
```

Scala-skript

- Scala-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
```

⁵Du 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 öppen-källkodsprojekt som löser det: <http://www.lihaoyi.com/Ammonite/>

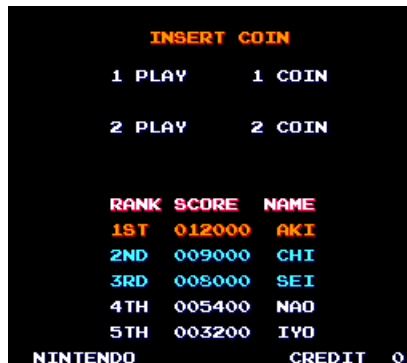
Algoritmer: stegvisa lösningar

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* \leftarrow poängen efter senaste spelet
- 2 *highscore* \leftarrow 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 många 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*

Test 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))
```

isHighscore är en **äkte funktion** som alltid ger samma svar för samma inparametrar och saknar **sideeffekter**.

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

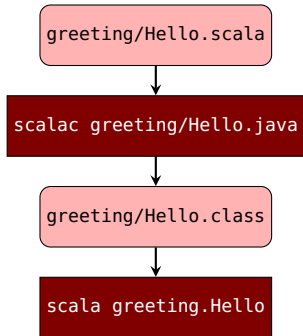
.class-fil med byte-kod

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:**

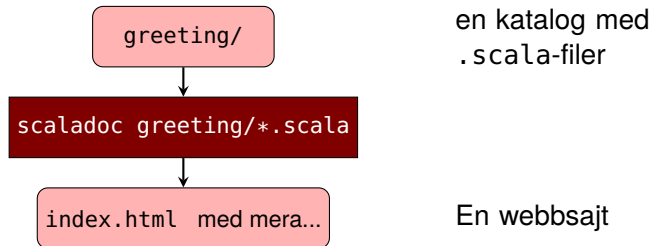
```
// Enradskommentarer börjar med dubbla snedstreck
//          men de gäller bara till radslut

/* 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.



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.

1 Introduktion

2 Kodstruktur

3 Funktioner, Objekt

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

4 Datastrukturer

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 deklareraras med tom parameterlista () kan den anropas på två sätt: med och utan tomma parenteser.

```
1 scala> def tomParameterLista() = 42
2
3 scala> tomParameterLista()
4 res2: Int = 42
5
6 scala> tomParameterLista
7 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:

```
1 scala> def ingenParameterLista = 42
2
3 scala> ingenParameterLista
4 res4: Int = 42
5
6 scala> ingenParameterLista()
7 <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

⁶en.wikipedia.org/wiki/Call_stack

⁷en.wikipedia.org/wiki/Memory_management

⁸Ej 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

⁹stackoverflow.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 42 42
```

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

```
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:

$f(\text{uttryck})$

mot krullparenteser:

$f\{\text{uttryck}\}$

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*)

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:

```
1 scala> def tvåGånger(x: Int, f: Int => Int) = f(f(x))
2
3 scala> def öka(x: Int) = x + 1
4
5 scala> def minska(x: Int) = x - 1
6
7 scala> tvåGånger(42, öka _)
8 res1: Int = 43
9
10 scala> tvåGånger(42, minska _)
11 res1: Int = 41
```

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

```
1 scala> tvåGånger(42, öka)
2 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 **funktionsliterals**.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:

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

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

```
1 scala> xs.map(x => x - 1)
2 res1: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
3
4 scala> xs map (x => x - 1)
5 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.

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

Platshållarsyntax för anonyma funktioner

- Understreck i funktionslitteraler kallas **platshållare** (eng. *placeholder*) och medger ett förkortat skrivsätt **om** den parameter som understreckt representerar används **endast en gång**.

```
_ + 1
```

Ovan expanderar av kompilatorn till följande funktionslitteral (där namnet på parametern är godtyckligt):

```
x => x + 1
```

- Det kan förekomma flera understreck; det första avser första parametern, det andra avser andra parametern etc.

```
_ + _
```

... expanderas till:

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

Exempel på platshållarsyntax med samlingsmetoden reduceLeft

Metoden `reduceLeft` applicerar 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 parameter
- 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

¹¹Även 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
```

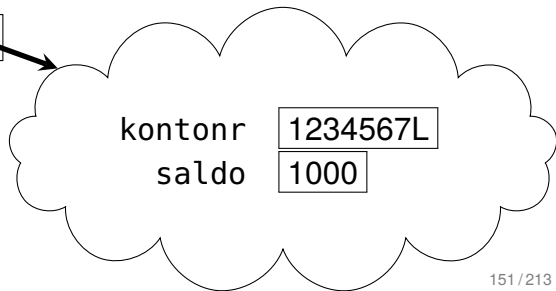
(Vi ska i nästa vecka se hur man med s.k. klasser kan skapa många upplagor av samma typ av objekt, så att vi kan ha flera olika bankkonto.)

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  
}
```

mittBankkonto



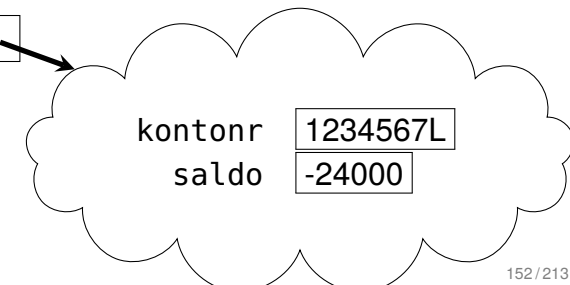
Tillståndsändring

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
```

mittBankkonto



Vad rymmer sköldpaddan i Kojo i sitt tillstånd?



position, riktning, 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 tillståndsförändringar. 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.

1 Introduktion

2 Kodstruktur

3 Funktioner, Objekt

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**.

```
1 scala> def längd(p: (Double, Double)) = math.hypot(p._1, p._2)
2
3 scala> def vinkel(p: (Double, Double)) = math.atan2(p._1, p._2)
4
5 scala> def polär(p: (Double, Double)) = (längd(p), vinkel(p))
6
7 scala> polär((3,4))
8 res2: (Double, Double) = (5.0,0.6435011087932844)
```

- Om typerna passar kan man skippa dubbla parenteser vid **ensamt tupel-argument**:

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

https://sv.wikipedia.org/wiki/Polära_koordinater

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
```

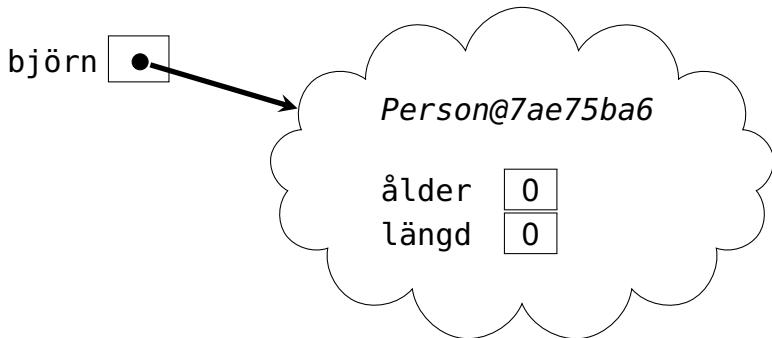
```
scala> björn.längd = 178
```

- 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 }
```

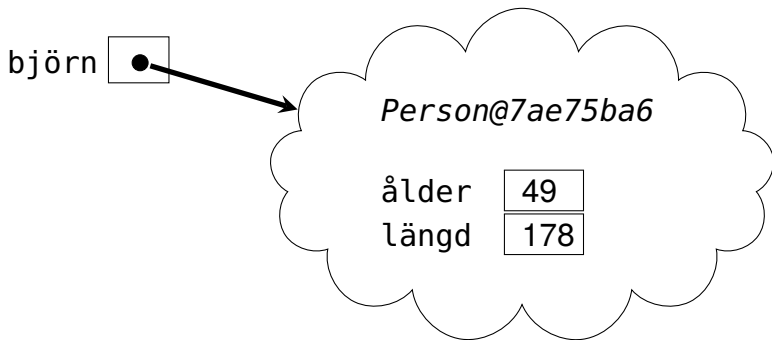
```
scala> val björn = new Person  
björn: Person = Person@7ae75ba6
```



`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 = 21  
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`

```
1 scala> class X(val i: Int)
2
3 scala> val a = new X(42)
4 a: X = X@117b2cc6
5
6 scala> a.isInstanceOf[X]
7 res0: Boolean = true
8
9 scala> val b = new X(42)
10 b: X = X@61ab6521
11
12 scala> b.isInstanceOf[X]
13 res1: Boolean = true
14
15 scala> a == b
16 res2: Boolean = false
17
18 scala> a.i == b.i
19 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 supertyp till **alla** typer och kallas för **rot-typ** i Scalas typhierarki.

```
1 scala> a.isInstanceOf[Any]
2 res4: Boolean = true
3
4 scala> b.isInstanceOf[Any]
5 res5: Boolean = true
6
7 scala> 42.isInstanceOf[Any]
8 res6: Boolean = true
```

(se quickref sid 4, mer om detta i w07)

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 nyckelordet **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 medel** 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): Broccoli = new Broccoli(vikt)
7  }
8
9  scala> val b = Broccoli(420)
10 b: Broccoli = Broccoli@6f2363
```

Kompanjonsobjekt kan accessa privata medlemmar

```
class Gurka(startVikt: Double) {  
  private var vikt = startVikt  
  def åt(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)
```

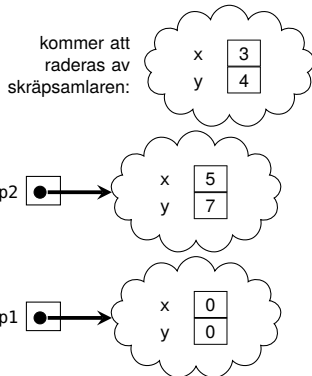
Minnessituationen efter rad 7:



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:

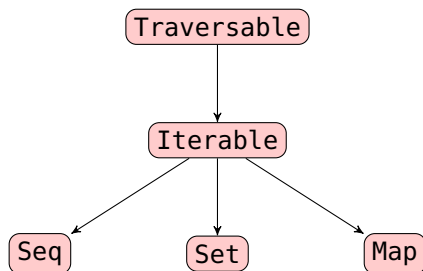
Heltalsvektor: **val** xs = Vector(-1, 2, 3, 42, 0)

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 oftast bäst att skapa en klass som *innehåller* en *färdig* samling och utgå från färdiga metoder.

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

<code>xs.size</code>	antal elementet
<code>xs.head</code>	första elementet
<code>xs.last</code>	sista elementet
<code>xs.take(n)</code>	ny samling med de första n elementet
<code>xs.drop(n)</code>	ny samling utan de första n elementet
<code>xs.foreach(f)</code>	gör f på alla element, returtyp Unit
<code>xs.map(f)</code>	gör f på alla element, ger ny samling

Iterable

<code>xs.zip(ys)</code>	ny samling med par (x, y); "zippa ihop" xs och ys
<code>xs.zipWithIndex</code>	ny samling med par (x, index för x)
<code>xs.sliding(n)</code>	ny samling av samlingar genom glidande "fönster"

Seq

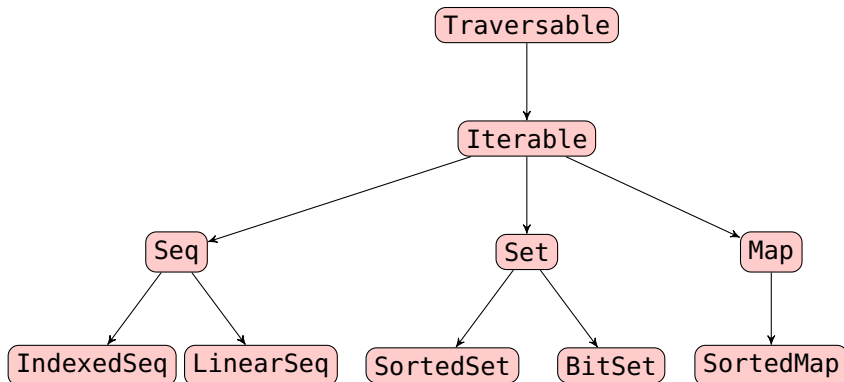
<code>xs.length</code>	samma som <code>xs.size</code>
<code>xs :+ x</code>	ny samling med x sist efter xs
<code>x ++ xs</code>	ny samling med x före xs

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

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

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

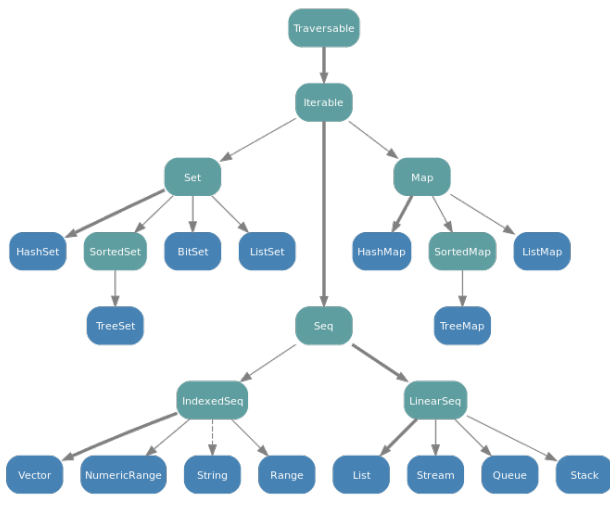
`LinearSeq.`

ListBuffer
Queue

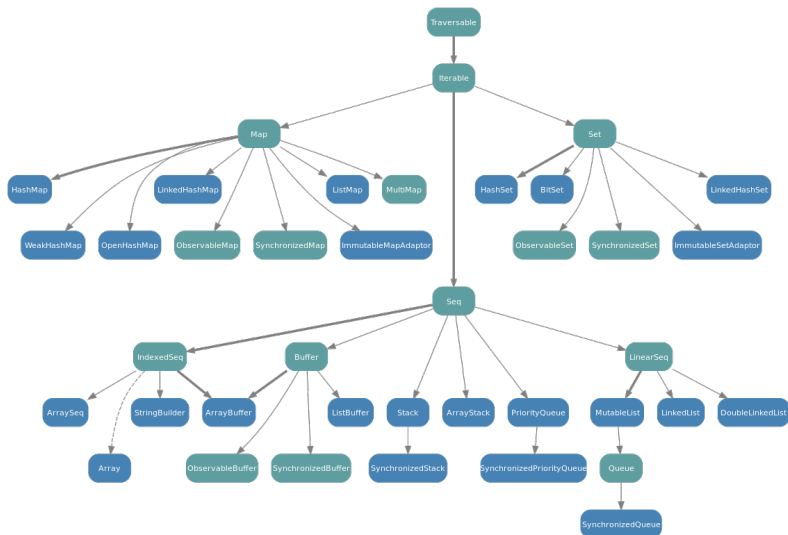
Studera samlingars egenskaper här:

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

scala.collection.immutable



scala.collection.mutable



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.

Speciella metoder på förändringsbara samlingar

Mängd: snabb innehållstest, garanterat dubblettfri

Den fantastiska nyckel-värde-tabellen Map

Integrerad utvecklingsmiljö (IDE)

Välja IDE

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 operatortnotation 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.