Programmering, grundkurs pgk

Föreläsningsanteckningar pgk (EDAA45)

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

Om kursen

Nytt för i år 2016

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

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

Veckoöversikt

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

Vad lär du dig?

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

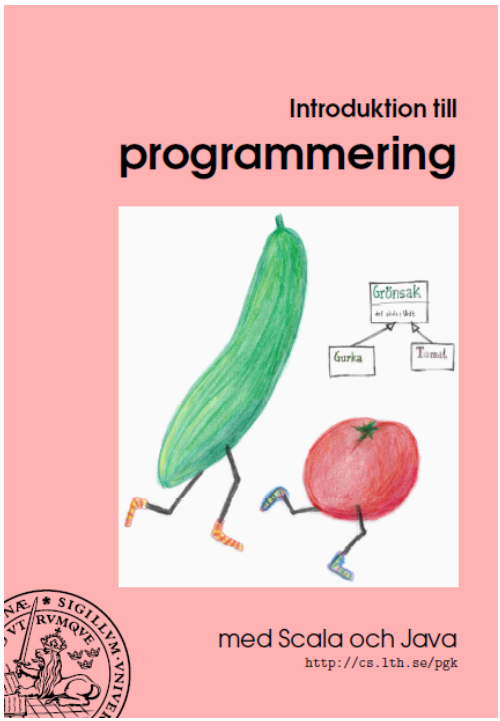
Varför Scala + Java som förstaspråk?

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

Hur lär du dig?

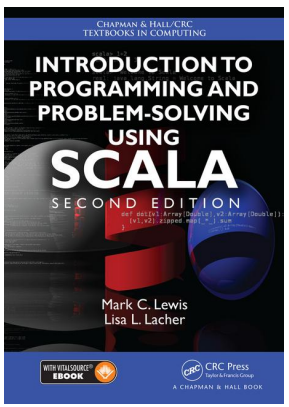
- Genom praktiskt **eget arbete**: **Lära genom att göra!**
  - Övningar: applicera koncept på olika sätt
  - Laborationer: kombinera flera koncept till en helhet
- Genom studier av kursens teori: **Skapa förståelse!**
- Genom samarbete med dina kurskamrater: **Gå djupare!**

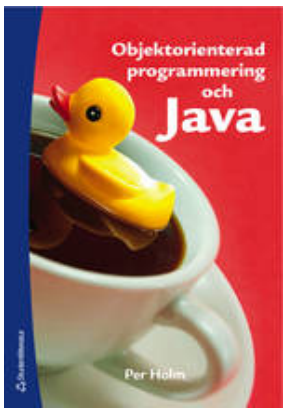
Kurslitteratur



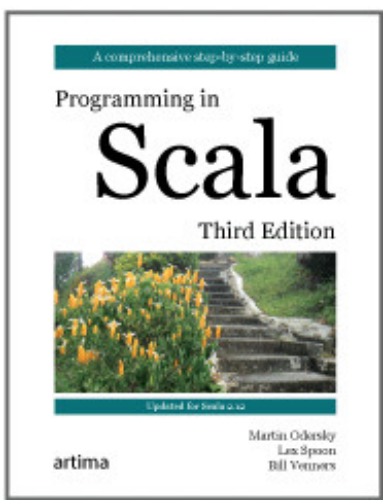
- **Kompendium** med övningar & laborationer, trycks & säljs av inst. på beställning
- Föreläsningsbilder
- Nätresurser enl. länkar

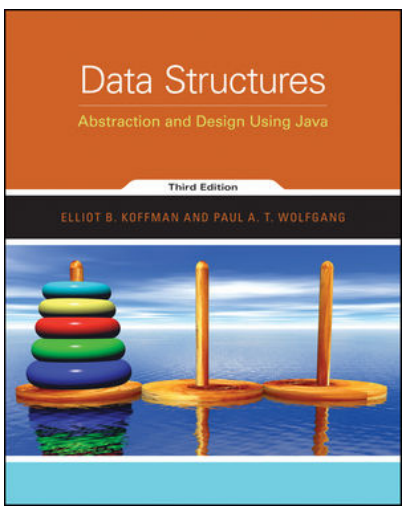
Bra, men ej nödvändig, **bredvidläsning**:
– för **nybörjare**:





– för de som **redan kodat** en del:





Beställning av kompendium och snabbreferens

- **Kompendiet** finns i pdf för fri nedladdning enl. CC-BY-SA, men det **rekommenderas starkt** att du köper den tryckta bokversionen.
- Det är mycket lättare att ha övningar och labbar **på papper bredvid skärmen**, när du ska tänka, koda och plugga!
- **Snabbreferensen** finns också i pdf men du behöver ha en tryckt version eftersom det är **enda tillåtna hjälpmedlet** på skriftliga kontrollskrivningen och tentamen.
- Kompendiet och snabbreferens trycks här i E-huset och säljs av institutionen till **självkostnadspris**.
- Pris för kompendium **beror på hur många som beställer**.
- Snabbreferens kostar 10 kr.
- Kryssa i **BOK** på listan som snart skickas runt – tryckning enligt denna beställning.
- Du betalar **kontant** med **jämna pengar** på cs expedition, våning 2.

Föreläsningsanteckningar

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

Personal

Kursansvarig:

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

Kurssekreterare:

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

Handledare:

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

Kursmoment — varför?

- **Föreläsningar**: skapa översikt, ge struktur, förklara teori, svara på frågor, motivera varför.
- **Övningar**: bearbeta teorins steg för steg, **grundövningar** för alla, **extraövningar** om du vill/behöver öva mer, **fördjupningsövningar** om du vill gå djupare; **förberedelse inför laborationerna**.
- **Laborationer**: **obligatoriska**, sätta samman teorins delar i ett större program; lösningar redovisas för handledare; gk på alla för att få tenta.
- **Resurstider**: få hjälp med övningar och laborationsförberedelser av handledare, fråga vad du vill.
- **Samarbetsgrupper**: grupplärande genom samarbete, hjälpa varandra.
- **Kontrollskrivning**: **obligatorisk**, diagnostisk, kamraträttad; kan ge samarbetsbonuspoäng till tentan.
- **Individuell projektuppgift**: **obligatorisk**, du visar att du kan skapa ett större program självständigt; redovisas för handledare.
- **Tentamen**: **obligatorisk**, skriftlig, enda hjälpmedel: snabbreferensen. `http://cs.lth.se/pgk/quickref`

Detta är bara början...

Exempel på efterföljande kurser som bygger vidare på denna:

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

Registrering

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

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

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

Förkunskaper

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

Förkunskapsenkät

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

Samarbetgrupper

- Ni delas in i **samarbetsgrupper** om ca 5 personer baserat på förkunskapsenkäten, så att olika förkunskapsnivåer sammanförs
- Några av laborationerna är mer omfattande **grupplabbar** och kommer att göras i samarbetsgrupperna
- Kontrollskrivningen i halvtid kan ge **samarbetsbonus** (max 5p) som adderas till ordinarie tentans poäng (max 100p) med medelvärdet av gruppmedlemmarnas individuella kontrollskrivningspoäng

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

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

Varför studera i samarbetsgrupper?

Huvudsyfte: **Bra lärande!**

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

Samarbetskontrakt

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

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

Diskutera hur ni ska uppfylla dessa innan alla skriver på.
Ta med samarbetskontraktet och visa för handledare på labb 1.

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

Bestraffa inte frågor!

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

Plagiatregler

Läs dessa regler noga och diskutera i samarbetsgrupperna:

- `http://cs.lth.se/utbildning/samarbete-eller-fusk/`
- Föreskrifter angående obligatoriska moment

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

En typisk kursvecka

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

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

Laborationer

- **Programmering lär man sig bäst genom att programmera...**
- Labbarna är **individuella** (utom 3) och **obligatoriska**
- Gör **övningarna** och **labbförberedelserna** noga **innan** själva labben – detta är ofta helt nödvändigt för att du ska hinna klart. Dina labbförberedelserna kontrolleras av handledare under labben.
- Är du **sjuk?** Anmäl det **före** labben till `bjorn.regnell@cs.lth.se`, få hjälp på resurstid och redovisa på resurstid (eller labbtid, när handledaren har tid över)
- Hinner du inte med hela labben? Se till att handledaren **noterar din närvaro**, och fortsätt på resurstid och ev. uppsamlingstider.
- Läs noga kapitel noll ”**Anvisningar**” i kompendiet!
- Laborationstiderna är gruppindelade enligt schemat. Du ska gå till den tid och den sal som motsvarar din grupp som visas i TimeEdit. **Gruppindelning** meddelas på hemsidan senast onsdag morgon.

Resurstider

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

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

Att lära denna läsvecka w01

Att lära denna läsvecka w01

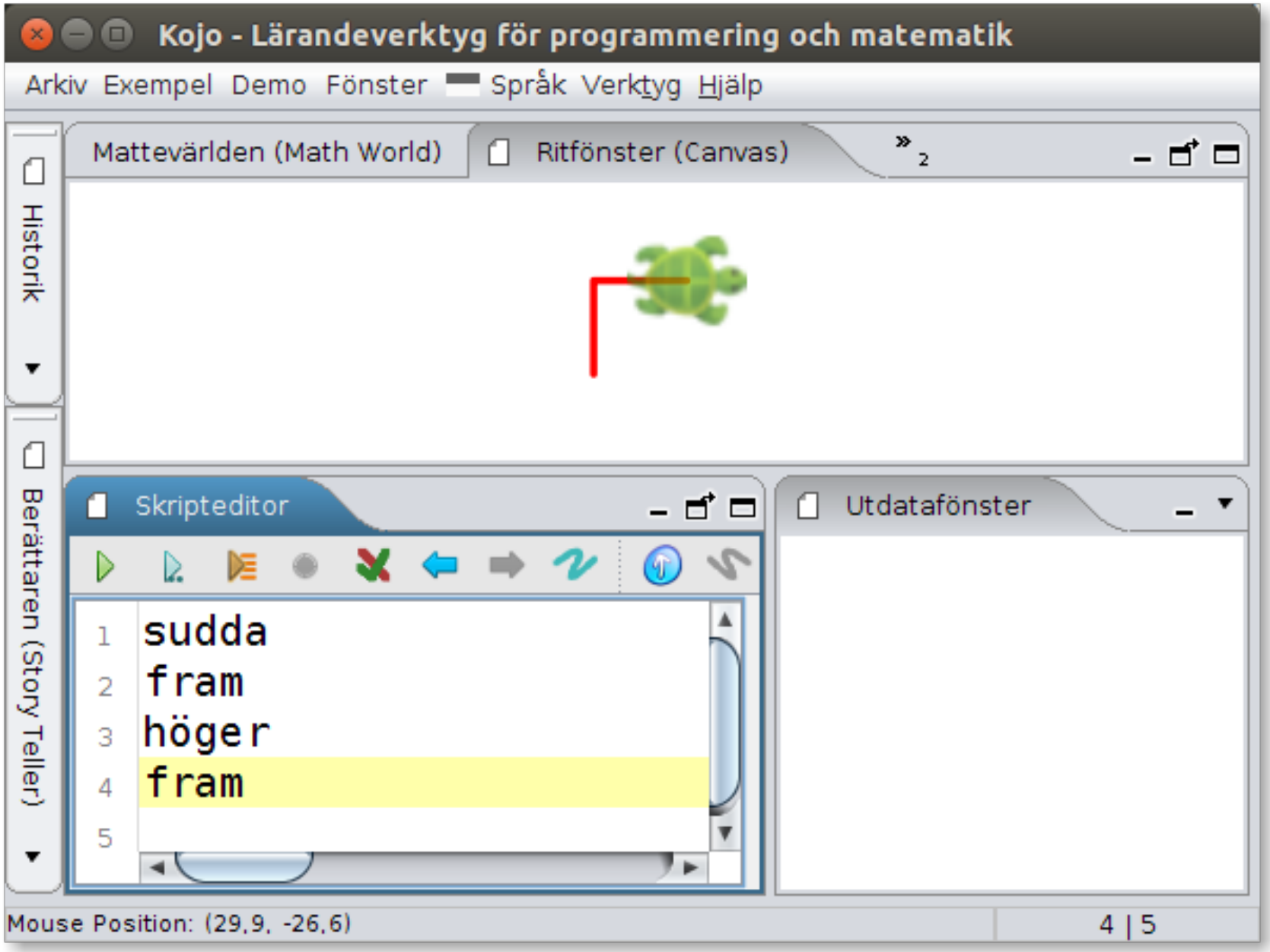
Modul **Introduktion**: Övn **expressions** → Labb **kojo**

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

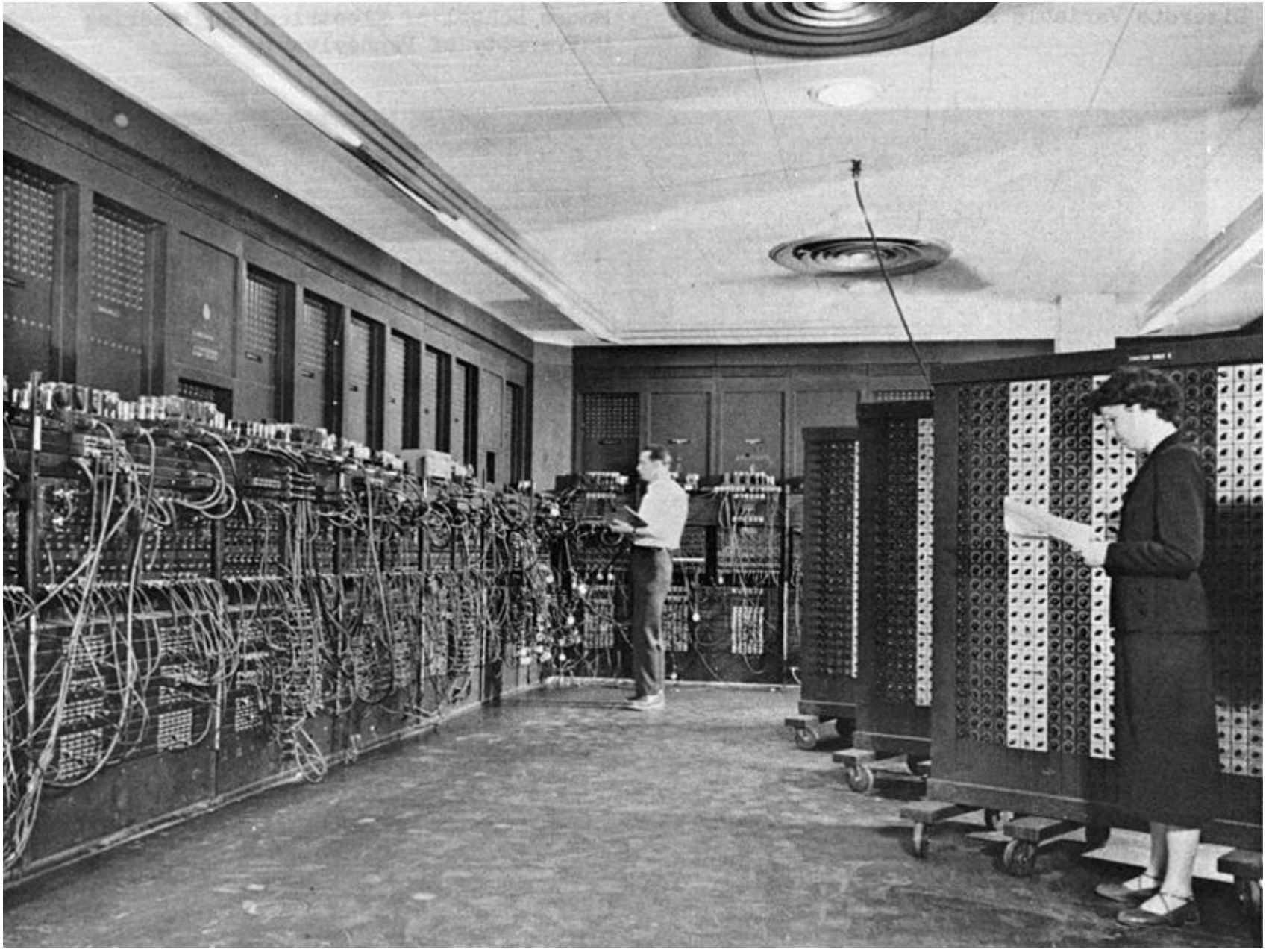
Om programmering

Programming unplugged: Två frivilliga?

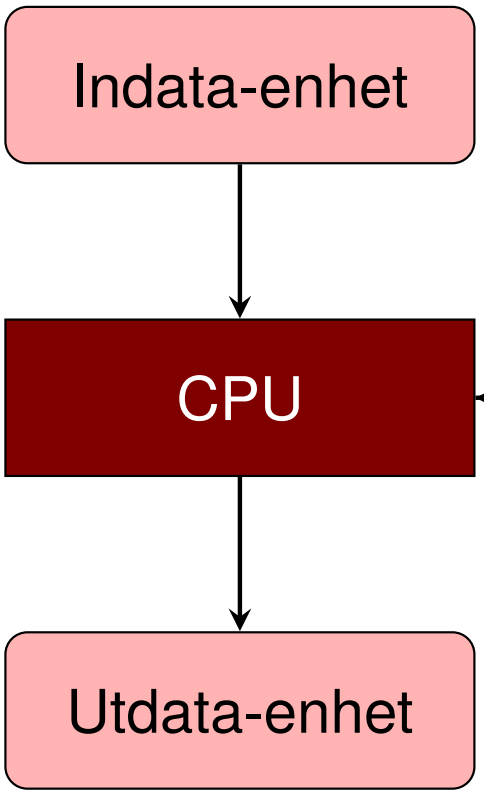
Editera och exekvera ett program



Vad är en dator?



Hur fungerar en dator?



Minne med minnesceller

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

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

Vad är programmering?

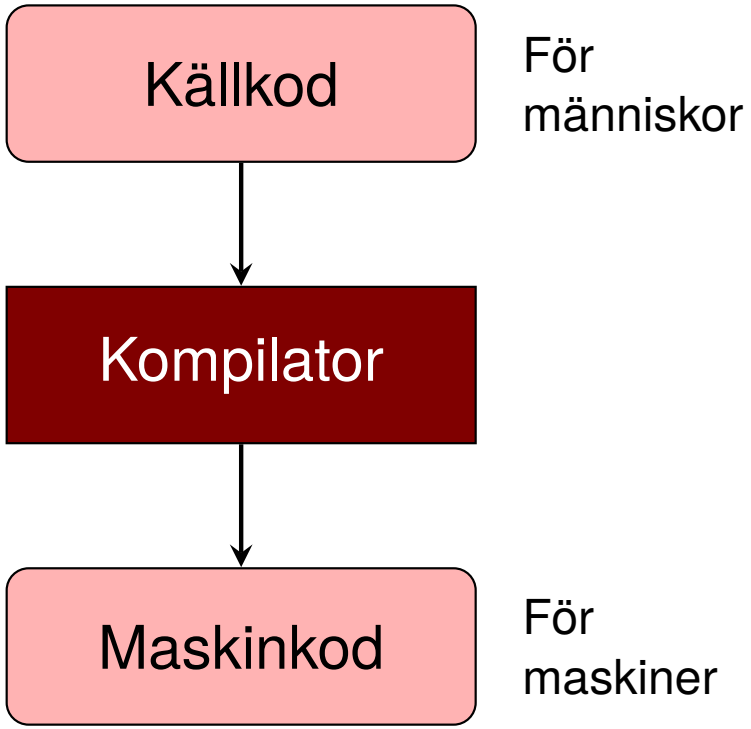
- Programmering innebär att ge instruktioner till en maskin.
- Ett **programmeringsspråk** används av människor för att skriva **källkod** som kan översättas av en **kompilator** till **maskinspråk** som i sin tur **exekveras** av en dator.
- Ada Lovelace skrev det första programmet redan på 1800-talet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

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

Vad är en kompilator?

Grace Hopper uppfann första kompilatorn 1952.

en.wikipedia.org/wiki/Grace_Hopper

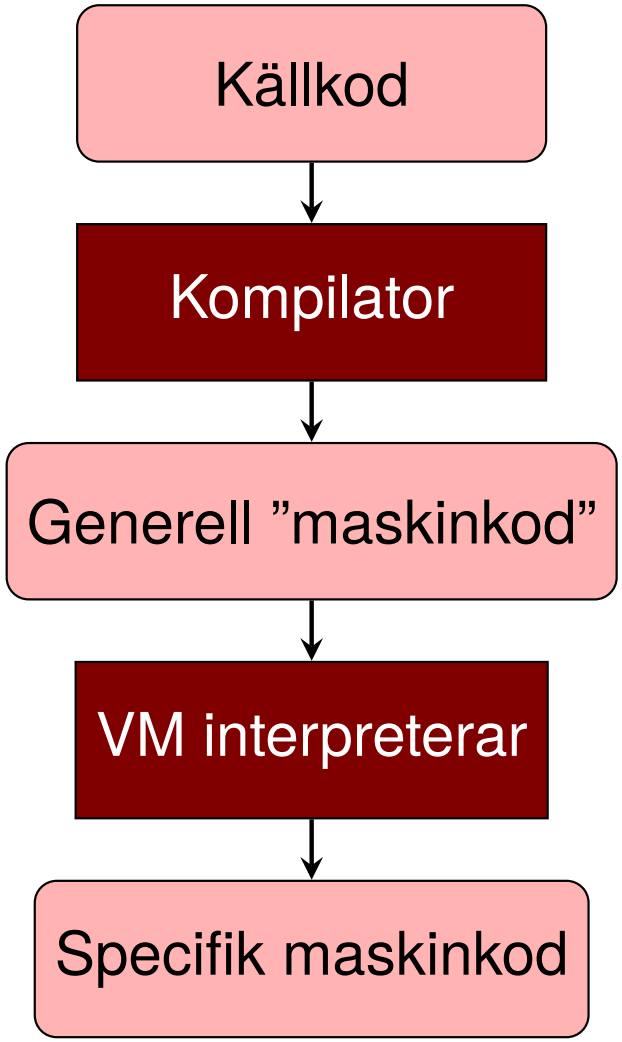


Virtuell maskin (VM) == abstrakt hårdvara

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

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

Exempel:
Java Virtual Machine



Vad består ett program av?

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

Exempel på programmeringsspråk

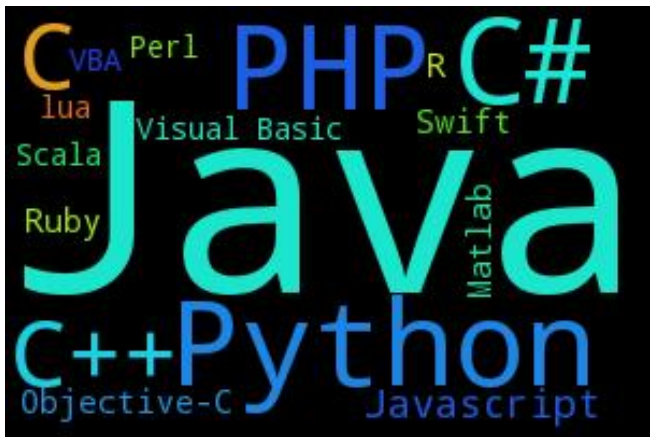
Det finns massor med olika språk och det kommer ständigt nya.

Exempel:

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

Topplistor:

- TIOBE Index
- PYPL Index



Olika programmeringsparadigm

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

Hello world

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

```scala
// this is Scala

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

```java
// this is Java

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

Utvecklingscykeln

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

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

Utvecklingsverktyg

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

Att skapa koden som styr världen

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

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

Hur blir ditt framtida yrkesliv som systemutvecklare?

- Det är sedan lång tid en **skriande brist** på utvecklare och bristen blir bara värre och värre...
  CS 2016-08-23
- Störst brist är det på **kvinnliga** utvecklare:
  DN 2015-04-02
- Global kompetensmarknad
  CS 2015-06-14
  CS 2016-07-14

Utveckling av mjukvara i praktiken

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

De enklaste beståndsdelarna: litteraler, uttryck, variabler

Literaler

- Literaler representerar ett fixt **värde** i koden och används för att skapa **data** som programmet ska bearbeta.
- Exempel:

`42`	heltalslitteral
`42.0`	decimaltalslitteral
`'!'`	teckenlitteral, omgärdas med 'enkelfnuttar'
`"hej"`	stränglitteral, omgärdas med "dubbelfnuttar"
`true`	litteral för sanningsvärdet "sant"

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

Exempel på inbyggda datatyper i Scala

- Alla värden, uttryck och variabler har en **datatyp**, t.ex.:
  - `Int` för heltal
  - `Long` för *extra* stora heltal (tar mer minne)
  - `Double` för decimaltal, så kallade flyttal med flytande decimalpunkt
  - `String` för strängar
- Kompilatorn håller reda på att uttryck kombineras på ett **typsäkert** sätt. Annars blir det **kompileringsfel**.
- Scala och Java är s.k. **statiskt typade** språk, vilket innebär att **all** typinformation måste finnas redan vid kompilering (eng. *compile time*)[3].
- Scala-kompilatorn gör **typhärledning**: man **slipper skriva typerna** om kompilatorn kan lista ut dem med hjälp av typerna hos deluttrycken.

[3] Andra språk, t.ex. Python och Javascript är **dynamiskt typade** och där skjuts typkontrollen upp till körningsdags (eng. *run time*)
Vilka är för- och nackdelarna med statisk vs. dynamisk typning?

Grundtyper i Scala

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

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

Grundtypernas implementation i JVM

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

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

Uttryck

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

Variabler

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

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

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

Regler för identifierare

- **Enkel** identifierare: t.ex. `gurka2tomat`
  - Börja med bokstav
  - ...följt av bokstäver eller siffror
  - Kan även innehålla understreck
- **Operator**-identifierare, t.ex. `+:`
  - Börjar med ett **operatortecken**, t.ex. `+ - * / : ? ~ #`
  - Kan följas av fler operatortecken
- En identifierare får **inte** vara ett **reserverat ord**, se snabbreferensen för alla reserverade ord i Scala & Java.
- **Bokstavlig** identifierare: `` `kan innehålla allt` ``
  - Börjar och slutar med **backticks** `` ` ``
  - Kan innehålla vad som helst (utom backticks)
  - Kan användas för att undvika krockar med reserverade ord: `` `val` ``

Att bygga strängar: konkatenering och interpolering

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

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

- Man kan i Scala (men inte Java) få hjälp av kompilatorn att övervaka bygget av strängar med **stränginterpolatorn s**:

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

Heltalsaritmetik

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

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

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

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

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

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

- **Moduloräkning** med restoperatorn %

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

Flyttalsaritmetik

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

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

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

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

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

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

Funktioner

Definiera namn på uttryck

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

```
def gurklängd = 42 + x
```

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

```
gurklängd
```

Funktioner kan ha parametrar

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

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

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

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

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

Färdiga matte-funktioner i paketet `scala.math`

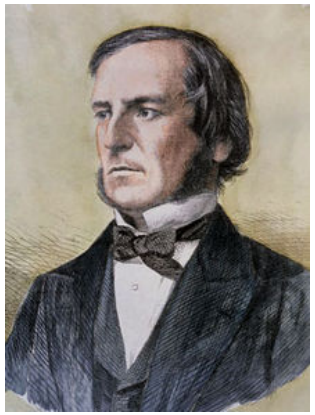
- I paketet **`scala.math`** finns många användbara funktioner: t.ex. `math.random` ger slumptal mellan `0.0` och `0.9999999999999999`

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

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

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

Logik

Logiska uttryck



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

```
true
false
```

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

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

De Morgans lagar

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

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

Exempel på förenkling där de Morgans lagar används upprepat:

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

Alternativ med if-uttryck

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

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

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

```
scala> slumpgrönsak
res13: String = gurka

scala> slumpgrönsak
res14: String = gurka

scala> slumpgrönsak
res15: String = tomat
```

Satser

Tilldelningssatser

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

Dessa deklarationer...

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

... ger detta innehåll någonstans i minnet:

x	42
y	43

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

```
x = 13
```

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

x	13
y	43

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

Tilldelningssatser är *inte* matematisk likhet

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

```
x = x + 1
```

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

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

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

Förkortade tilldelningssatser

- Det är vanligt att man vill applicera en **tilldelningsoperator** på variabeln själv, så som i `x = x + 1`
- Därför finns **förkortade tilldelningssatser** som gör så att man sparar några tecken och det blir tydligare (?) vad som sker (när man vant sig vid detta skrivsätt):

  ```
  x += 1
  ```

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

Exempel på förkortade tilldelningssatser

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

scala> x *= 2

scala> x
res0: Int = 84


scala> x /= 3

scala> x
res2: Int = 28
```

Övning: Tilldelningar i sekvens

Rita hur minnet ser ut efter varje rad nedan:

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

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

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

Variabler som ändrar värden kan vara knepiga

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

Kontrollstrukturer: alternativ och repetition

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

- if-sats:

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

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

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

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

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

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

Procedurer

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

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

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

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

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

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

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

Om veckans övning: `expressions`

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

Om veckans labb: `kojo`

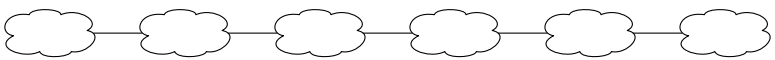
- Kunna kombinera principerna sekvens, alternativ, repetition, och abstraktion i skapandet av egna program om minst 20 rader kod.
- Kunna förklara vad ett program gör i termer av sekvens, alternativ, repetition, och abstraktion.
- Kunna tillämpa principerna sekvens, alternativ, repetition, och abstraktion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

Datastrukturer och kontrollstrukturer

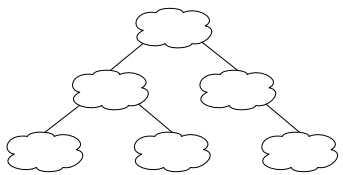
Vad är en datastruktur?

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

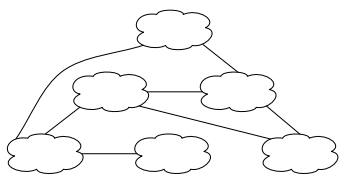
Lista



Träd



Graf



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

Vad är en vektor?

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

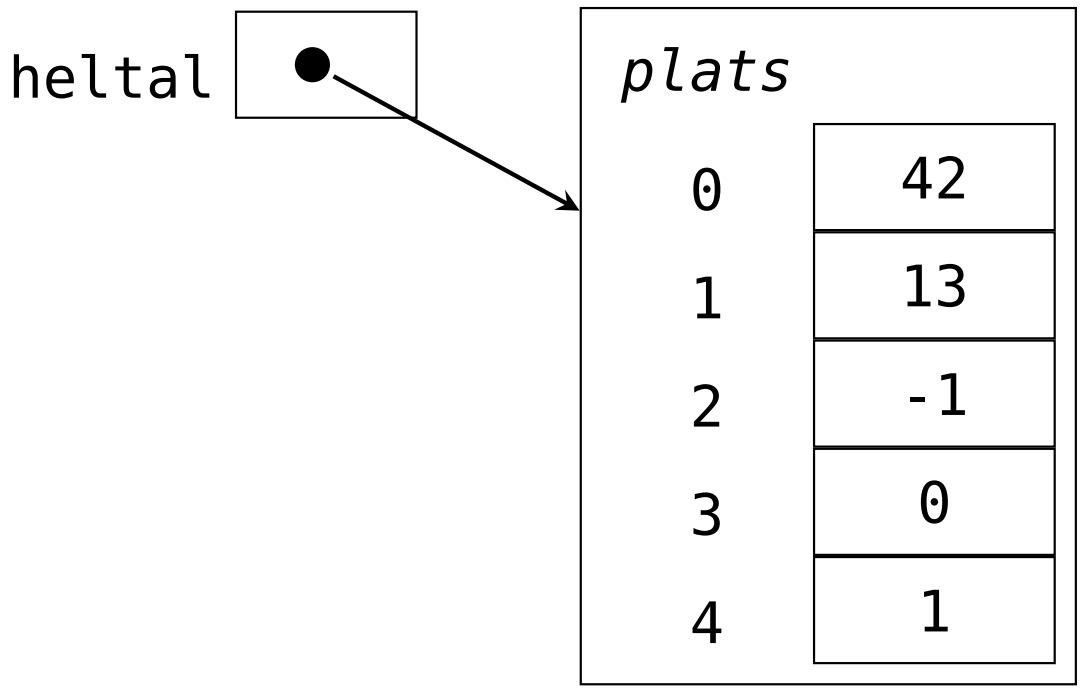
```
1  scala> val heltal = Vector(42, 13, -1, 0 , 1)
2  heltal: scala.collection.immutable.Vector[Int] = Vector(42, 13, -1, 0, 1)
3
4  scala> heltal.apply(0)
5  res0: Int = 42
6
7  scala> heltal(1)    // man kan skippa .apply
8  res1: Int = 13
9
10 scala> heltal(5)
11 java.lang.IndexOutOfBoundsException: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

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

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

En konceptuell bild av en vektor

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

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



En samling strängar

- En vektor kan lagra många värden av samma typ.
- Elementen kan vara till exempel heltal eller strängar.
- Eller faktiskt vad som helst.

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

Vad är en kontrollstruktur?

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

  Exempel på inbyggda kontrollstrukturer:

  `for`-sats, `while`-sats

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

  Exempel: `upprepa` som du använt i Kojo

```
upprepa(4){fram; höger}
```

Mitt första program: en oändlig loop på ABC80

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

Mitt första program: en oändlig loop på ABC80

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


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

Loopa genom elementen i en vektor

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

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

Bygga en ny samling från en befintlig med for-uttryck

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

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

```
1  scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")
2
3  scala> for (g <- grönsaker) yield "god " + g
4  res0: scala.collection.immutable.Vector[String] =
5    Vector(god gurka, god tomat, god paprika, god selleri)
6
7  scala> val åsikter = for (g <- grönsaker) yield s"god $g"
8  åsikter: scala.collection.immutable.Vector[String] =
9    Vector(god gurka, god tomat, god paprika, god selleri)
```

Samlingen Range håller reda på intervall

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

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

- Men alla värden däremellan skapas inte förrän de behövs:

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

Loopa med Range

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

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

Du kan skapa en Range med `until` efter ett heltal:

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

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

Loopa med Range skapad med `to`

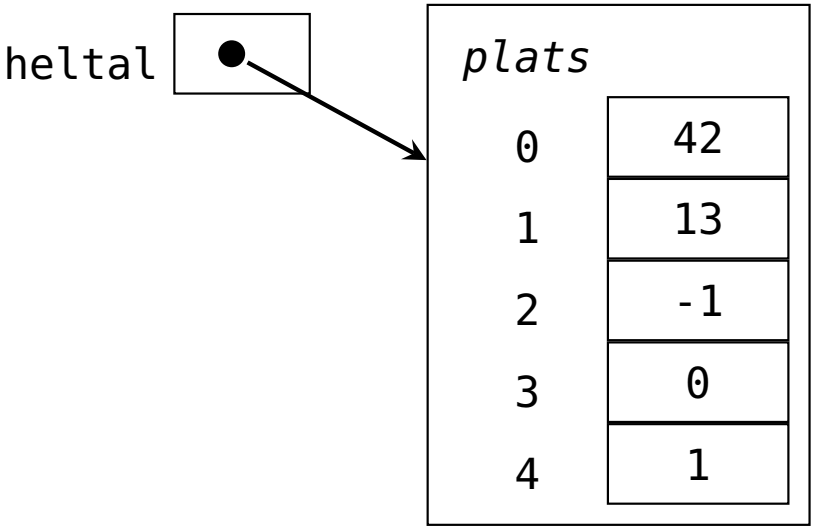
Med `to` efter ett heltal får du en Range till och **med** sista:

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

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

Vad är en `Array` i JVM?

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

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



Några likheter & skillnader mellan `Vector` och `Array`

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

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

Några likheter mellan `Vector` och `Array`

- Båda är samlingar som kan innehålla många element.
- Med båda kan man snabbt accessa vilket element som helst: `xs(2)`
- Båda har en fix storlek efter allokering.

Några viktiga skillnader:

Vector

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

Array

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

Huvudprogram med `main` i Scala och Java

Ett minimalt fristående program i Scala och Java

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

```
// this is Scala

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

Ett minimalt fristående program i Scala och Java

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

```
// this is Scala

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

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

```
// this is Java

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

Loopa genom en samling med en `while`-sats

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

scala> xs.size
res0: Int = 4

scala> var i = 0
i: Int = 0

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

Loopa genom argumenten i ett Scala-huvudprogram

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

```
$ gedit helloargs.scala
```

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

Kompilera och kör:

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

Loopa genom argumenten i ett Java-huvudprogram

```
$ gedit HelloJavaArgs.java
```

```java
// this is Java

public class HelloJavaArgs {
    public static void main(String[] args) {
    int i = 0;
    while (i < args.length) {
      System.out.println(args[i]);
      i = i + 1;
    }
  }
}
```

Kompilera och kör:

```
$ javac HelloJavaArgs.scala
$ java HelloJavaArgs hej gurka tomat
hej
gurka
tomat
```

Scala-skript

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

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

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

```
$ scala myscript.scala
```

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

Algoritmer: stegvisa lösningar

Vad är en algoritm?

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

Exempel:

- baka en kaka

Vad är en algoritm?

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

Exempel:

- baka en kaka
- räkna ut din pensionsprognos

Vad är en algoritm?

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

Exempel:

- baka en kaka
- räkna ut din pensionsprognos
- köra bil

Vad är en algoritm?

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

Exempel:

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

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

Varför?

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

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

Algoritm:

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

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

Algoritm:

1. *points* $\leftarrow$ 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!”

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

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

Algoritm:

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

Hittar du buggen?

HIGHSCORE implementerad i Scala

```
import scala.io.StdIn.readLine

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

HIGHSCORE implementerad i Scala

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

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

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

HIGHSCORE implementerad i Java

```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 $prod$

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*

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
```

Mall för funktionsdefinitioner

def funktionsnamn(parameterdeklarationer): returtyp = block

Exempel:

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

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
```

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
}
```

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(":(")
```

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))
```

Bättre många små abstraktioner som gör en sak var

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

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

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

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

Vad är ett block?

- Ett block **kapslar in** flera satser/uttryck och ser ”utifrån” ut som en enda sats/uttryck.
- Ett block skapas med hjälp av klammerparenteser (”krullparenteser”)

```
{ uttryck1; uttryck2; ... uttryckN }
```

Vad är ett block?

- Ett block **kapslar in** flera satser/uttryck och ser ”utifrån” ut som en enda sats/uttryck.
- Ett block skapas med hjälp av klammerparenteser (”krullparenteser”)

```
{ uttryck1; uttryck2; ... uttryckN }
```

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

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

Namn i block blir **lokala**

Synlighetsregler:

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

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

Parameter och argument

Skilj på parameter och argument!

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

```
scala> val ettArgument = 42

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

scala> öka(ettArgument)
```

Speciell syntax: anrop med s.k. **namngiven parameter**

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

Procedurer

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

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

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

”Ingenting” *är* faktiskt någonting i Scala

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

  `https://en.wikipedia.org/wiki/Void_type`

  `https://en.wikipedia.org/wiki/Unit_type`

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

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

Exempel på **funktionell nedbrytning**

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

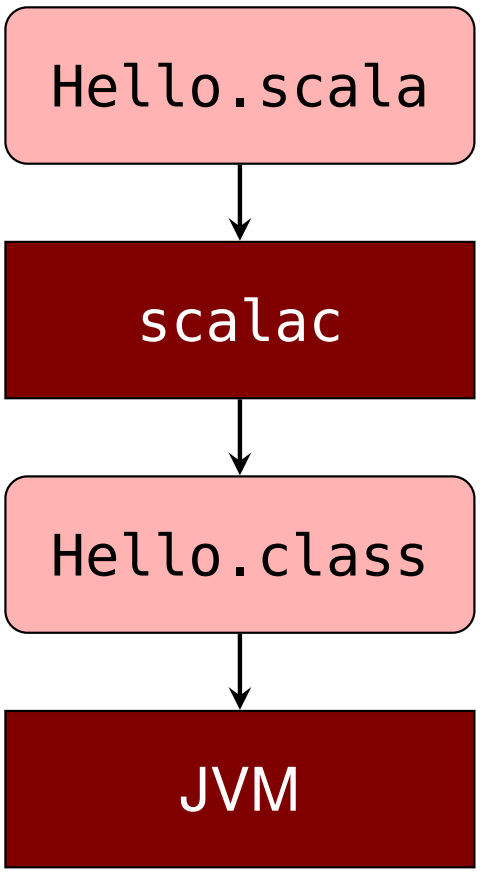
```
// skapa abstraktioner som bygger på varandra

def kvadrat = upprepa(4){fram; höger}

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

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

// huvudprogram

sudda; sakta(200)
rutnät
```

Varför abstraktion?

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

Katalogstruktur för kodfiler med paket

Källkodsfiler och klassfiler



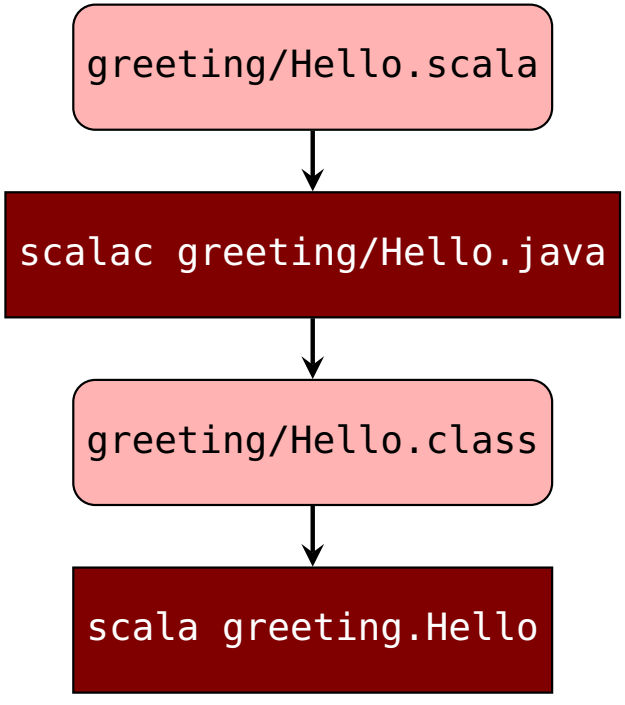
Källkodsfil

`.class`-fil med byte-kod

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

Paket

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




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

Paketens bytekod hamnar i katalog med samma namn som paketnamnet

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

Import

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

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

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

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

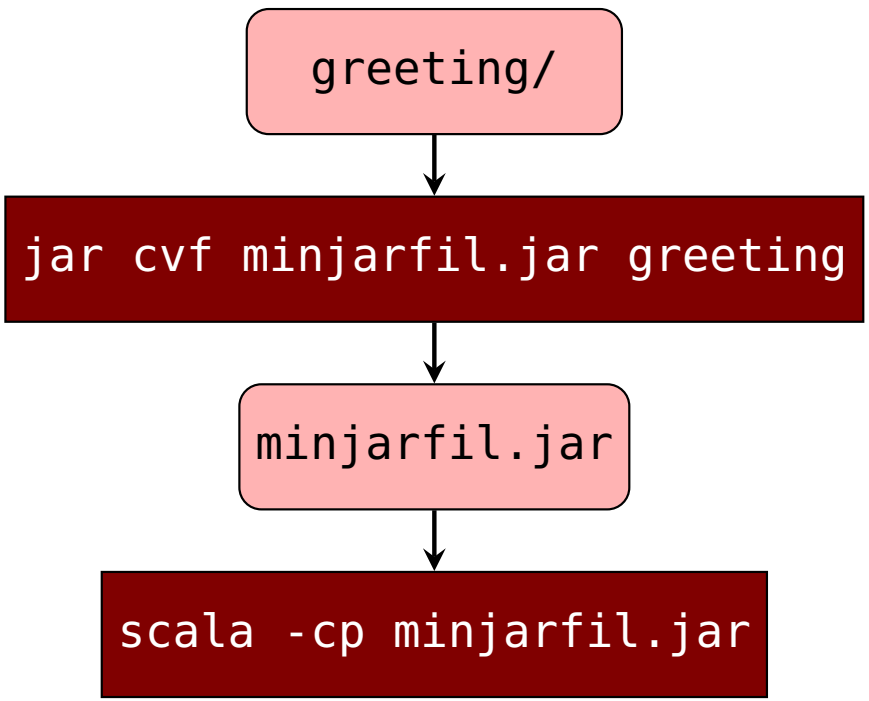
...gör så att kompilatorn ”ser” namnet, och man slipper skriva hela sökvägen till namnet:

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

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

Jar-filer

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



en katalog med filer

En jar-fil med alla filer inpackade

Lägg jar-filen till ”classpath”

Dokumentation

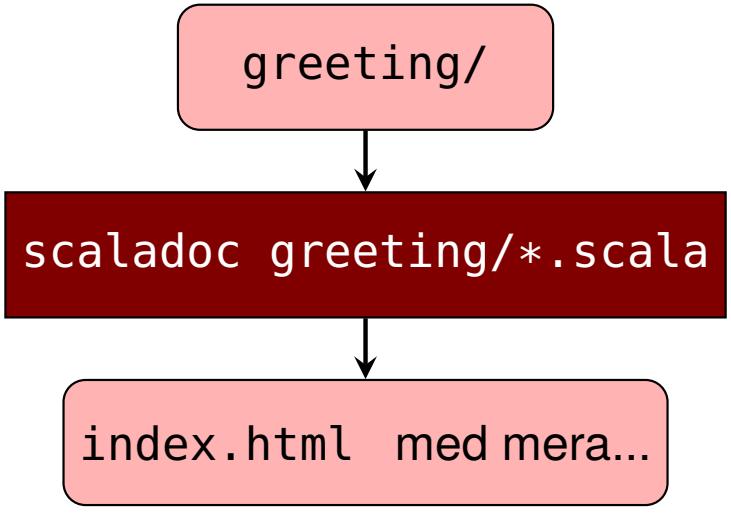
Dokumentation

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

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

/* Flerradskommentarer börjar med
   snedstreck-asterisk
   och slutar med asterisk-snedstreck.  */

/** Dokumentationskommentarer placeras före
 *   t.ex. en funktion och berättar vad den gör
 *   och vad eventuella parametrar används till.
 *   Börjar med snedstreck-asterisk-asterisk.
 *   Varje ny kommentarsrad börjar med asterisk.
 *   Avslutas med asterisk-stjärna.
 */
```

scaladoc

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



en katalog med `.scala`-filer

En webbsajt

Att göra i Vecka 1: Förstå grundläggande kodstrukturer

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

Veckans övning: `w02-programs`

- Kunna skapa samlingarna Range, Array och Vector med heltals- och strängvärden.
- Kunna indexera i en indexerbar samling, t.ex. Array och Vector.
- Kunna anropa operationerna size, mkString, sum, min, max på samlingar som innehåller heltal.
- Känna till grundläggande skillnader och likheter mellan samlingarna Range, Array och Vector.
- Förstå skillnaden mellan en for-sats och ett for-uttryck.
- Kunna skapa samlingar med heltalsvärden som resultat av enkla for-uttryck.
- Förstå skillnaden mellan en algoritm i pseudo-kod och dess implementation.
- Kunna implementera algoritmerna SUM, MIN/MAX på en indexerbar samling med en `while`-sats.
- Kunna köra igång enkel Scala-kod i REPL, som skript och som applikation.
- Kunna skriva och köra igång ett enkelt Java-program.
- Känna till några grundläggande syntaxskillnader mellan Scala och Java, speciellt variabeldeklarationer och indexering i Array.
- Förstå vad ett block och en lokal variabel är.
- Förstå hur nästlade block påverkar namnsynlighet och namnöverskuggning.
- Förstå kopplingen mellan paketstruktur och kodfilstruktur.
- Kunna skapa en jar-fil.
- Kunna skapa dokumentation med scaladoc.