

# EDAA45 Programmering, grundkurs

## Läsvecka 2: Kodstrukturer

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

## 2 Kodstrukturer

- Studieteknik
- Datastrukturer och kontrollstrukturer
- Huvudprogram med `main` i Scala och Java
- Algoritmer: stegvisa lösningar
- Funktioner skapar struktur
- Kodfilernas katalogstruktur
- Att göra denna vecka

# Studieteknik

# Hur studerar du?

- Vad är bra **studieteknik**?
- Hur lär **du** dig bäst? Olika personer har olika preferenser.
  - Ta reda på vad som funkar bäst för dig.
  - En kombination av flera sinnen är bäst: läsa+prata+skriva...
  - Aktivera dig! Inte bara passivt läsa utan också aktivt göra.
- Hur skapa **struktur**? Du behöver ett sammanhang, ett **system av begrepp**, att **placera in** din nya kunskap i.
- Hur uppbåda **koncentration**? Steg 1: Stäng av mobilen!
- Hur vara **disciplinerad**? Studier först, nöje sen!
- Du måste **planera och omplanera** för att säkerställa **tillräckligt mycket egen pluggtid** då du är pigg och koncentrerad för att det ska funka!
- Programmering **kräver** en **pigg och koncentrerad hjärna!**

# Hur ska du studera programmering?

## ■ När du gör **övningarna**:

- Ta fram föreläsningssbilderna i pdf och kolla igenom dem.
- Är det något i föreläsningssbilderna du inte förstår: ta upp det i samarbetsgrupperna eller på resurstiderna.
- Om något är knepigt:
  - Hitta på egna REPL-experiment och undersök hur det funkar.
  - Följ ev. länkar i föreläsningssbilderna, eller googla själv på wikipedia, stackoverflow, ...

## ■ Innan du gör **laborationerna**:

- Kolla igenom målen för veckans **övning** och dubbelkolla så att du har uppnått dem.
- Gör förberedelserna **i god tid innan** labben.
- Läs igen **hela** labbinstruktionen **innan** labben.
- Om du tror att du behöver det för att hinna med: gör delar av labben redan innan redovisningstillfället.

## På rasten: träffa din samarbetsgrupp

### ■ Träffas i samarbetsgrupperna och bestäm/gör/diskutera:

1 När ska ni träffas nästa gång?

2 Bläddra igenom föreläsningsbilderna från w01 i pdf.

3 Vilka **koncept** är fortfarande (mest) **grumliga**?

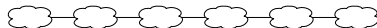
Alltså: Vilka koncept från förra veckan vill ni på nästa möte jobba mer med i gruppen för att alla ska förstå grunderna?

# 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**<sup>1</sup> (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.

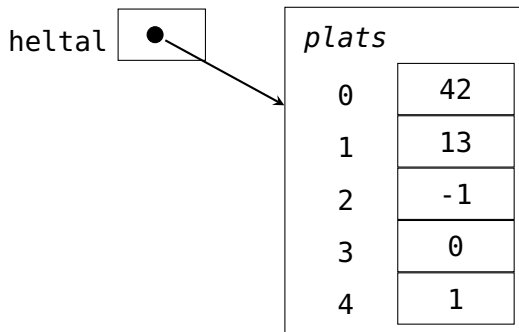
---

<sup>1</sup>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
<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:

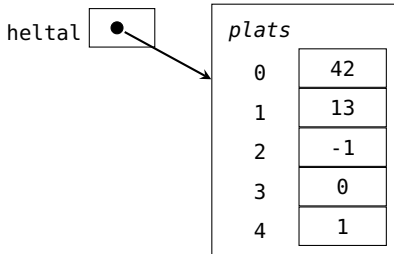
```
scala> 1 to 6
res2: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5, 6)

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:

```
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**.<sup>2</sup>
- 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
```

---

<sup>2</sup>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 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

# 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

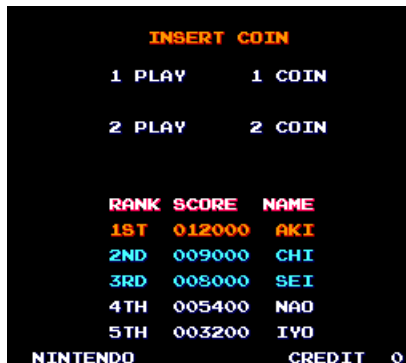


# Vad är en algoritm?

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

## Exempel:

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



# Algoritm-exempel: HIGHSCORE

**Problem:** Uppdatera high-score i ett spel

**Varför?**

# Algoritm-exempel: HIGHSCORE

**Problem:** Uppdatera high-score i ett spel

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

**Algoritm:**

# Algorithm-exempel: HIGHSCORE

**Problem:** Uppdatera 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!"

# Algorithm-exempel: HIGHSCORE

**Problem:** Uppdatera 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?")
    val highscore = readLine("Vad var highscore före senaste spelet?")
    val msg = if (points > highscore) "GRATTIS!" else "Försök igen!"
    println(msg)
  }
}
```

# 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?")
    val highscore = readLine("Vad var highscore före senaste spelet?")
    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  
?

# 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*

# 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))
```

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 }
```

# 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
```

# Block kan ha lokala variabler

Synlighetsregler etc

# Funktioner

# Parameter och argument



# 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/Void_type)

[https://en.wikipedia.org/wiki/Unit\\_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

<<kvadrat, stapel, rutnät i kojo>>

# Kodfilernas katalogstruktur

# Paket

<<paket skapar katalogstruktur för kodfiler>>

# Jar-filer

<<liknar zip-filer; packar ihop bytekod i en enda fil för enkel distribution>>

## Att göra i Vecka 2: 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: Gör klart samarbetskontrakt och visa för handledare på resurstid.
- 7 **Koda på resurstiderna** och få hjälp och tips!