Vecka 2. Program och kontrollstrukturer

Programmering, grundkurs (pgk)

Björn Regnell

Datavetenskap, LTH, Lunds universitet
https://lunduniversity.github.io/pgk

EDAA45, Lp1-2, HT2025

Kompilerad den 22 augusti 2025

2 Program och kontrollstrukturer

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

Om förra veckan och kommande två veckor

Förra veckan

Viktiga övergripande mål:

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

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

Fråga på resurstider och labbar

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

Kommande 2 veckor

Övergripande mål:

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

Läsvecka 2:

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

Läsvecka 3:

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

OBS! Notera **schemaavvikelser** – veckorna är inte identiska, se schema i TimeEdit.

Studieteknik

Hur studerar du?

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

Hur ska du studera programmering?

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

Det går inte att förstå allt på en gång!

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

Från förra veckan: `val` `var` `def`

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

Från förra veckan: `val` `var` `def`

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

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

Från förra veckan: `val` `var` `def`

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

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

En **konstighet i REPL**:

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

Från förra veckan: `if then else`

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

Förenkla dessa if-uttryck:

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

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

Från förra veckan: `if then else`

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

Förenkla dessa if-uttryck:

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

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

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

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

Från förra veckan: `if then else`

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

Förenkla dessa if-uttryck:

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

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

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

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

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

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

Från förra veckan: `if then else`

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

Förenkla dessa if-uttryck:

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

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

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

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

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

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

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

Från förra veckan: `if then else`

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

Förenkla dessa if-uttryck:

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

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

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

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

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

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

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

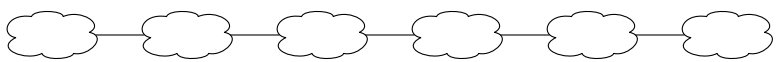
Repetera även **de Morgans lagar** från vecka 1 så du har koll på dem.

Datastrukturer

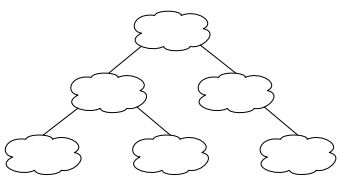
Vad är en datastruktur?

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

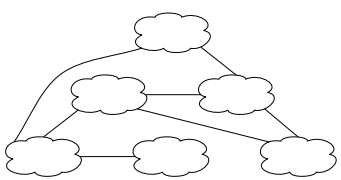
Sekvens



Träd



Graf



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

Några samlingar i `scala.collection`

- En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.
- En **sekvens** (eng. *sequence*) är en samling där alla element är ordnade.
- Exempel på **färdiga samlingar** i Scalas standardbibliotek där elementen är organiserade internt på **olika** vis så att samlingen får olika egenskaper som passar **olika användningsområden**:
  - `scala.collection.immutable.Vector`, sekvens med snabb access **överallt**.
  - `scala.collection.immutable.List`, sekvens med snabb access **i början**.
  - `scala.collection.immutable.Set`, `scala.collection.mutable.Set`, mängd med unika element; ej i sekvens men snabb innehållstest.
  - `scala.collection.immutable.Map`, `scala.collection.mutable.Map`, mängd med par av nyckel & tillhörande värde, snabb access via nyckel.
  - `scala.collection.mutable.ArrayBuffer`, förändringsbar sekvens kan ändra storlek.
  - `scala.Array`, förändringsbar sekvens som **inte** kan ändra storlek. Alla element är lagrade efter varandra i minnet: snabbast access av alla samlingar, men har speciella begränsningar.

Olika strukturer för att hantera data

- **Tupel** (eng. *tuple*)
  - samla flera datavärden t.ex. `(1, "hej", true)` i element `_1`, `_2`, `_3`
  - elementen kan vara av **olika** typ
- **Enumeration** (även kallad *uppräkning*) (eng. *enumeration*)
  - Namnge uppräknade värden t.ex. `enum Color { case Red, Black }`
  - Värdena har ordningsnummer och är alla av **samma** typ (här `Color`)
- **Klass** (eng. *class*)
  - samlar data i **attribut** med (väl valda!) namn
  - attributen kan vara av **olika** typ
  - definierar även **metoder** som använder attributen (kallas även **operationer** på data)
- **Färdig samling**
  - speciella klasser som samlar data i element av **samma** typ
  - exempel: `scala.collection.immutable.Vector`
  - har ofta *många* färdiga **bra-att-ha-metoder**, se snabbreferensen `https://fileadmin.cs.lth.se/pgk/quickref.pdf`
- **Egenimplementerade samlingar**
  - → fördjupningskurs

Vad är en vektor?

En **vektor**[1] (eng. *vector*) är en **sekvens** som är **snabb** att **indexera** i. Åtkomst av element i en sekvens som t.ex. heter `xs` sker i Scala med `xs.apply(platsnummer)`:

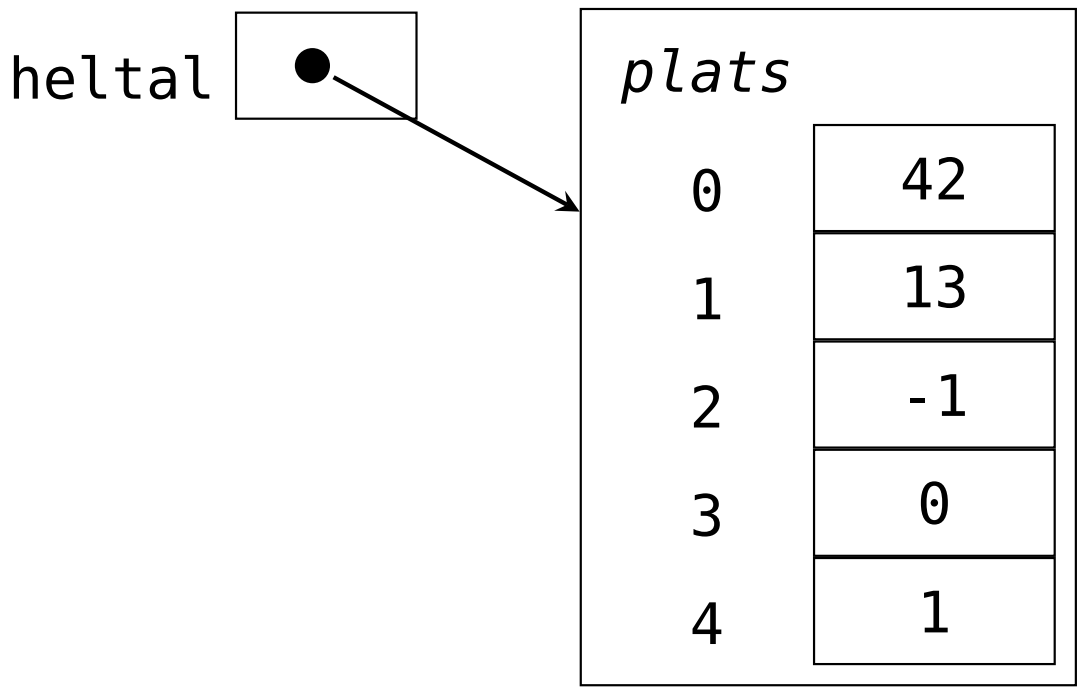
```
1  scala> val heltal = Vector(42, 13, -1, 0, 1)
2  val heltal: scala.collection.immutable.Vector[Int] = Vector(42, 13, -1, 0, 1)
3
4  scala> heltal.apply(0)   // platsnummer räknas från noll
5  val res0: Int = 42
6
7  scala> heltal(1)         // man kan i Scala skippa .apply före (
8  val res1: Int = 13
9
10 scala> heltal(5)         // ger körtidsfel då sjätte platsen inte finns
11 java.lang.IndexOutOfBoundsException: 5
12   at scala.collection.immutable.Vector.checkRangeConvert(Vector.scala:132)
```

Utelämnar du `.apply` så skapar kompilatorn automatiskt ett anrop av `apply`.

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

En konceptuell bild av en vektor

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

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



En samling strängar

- En vektor kan lagra **många** värden av samma typ.
- Elementen kan vara till exempel heltal eller strängar.
- Eller faktiskt vad som helst. (En s.k. *generisk* samling.)

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

Notera typen `Matchable` som betyder ”**nästan vilken typ som helst**”

(Mer om `Matchable` senare.)

Kontrollstrukturer

Vad är en kontrollstruktur?

- En **kontrollstruktur** påverkar i vilken ordning (sekvens) satser exekveras och uttryck evalueras.

  Exempel på **inbyggda** kontrollstrukturer:

  `for`-**`do`**-sats
  `while`-**`do`**-sats
  `for`-**`yield`**-uttryck

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

  Exempel: `upprepa` som du använt i Kojo

  ```
  upprepa(4){fram; höger}
  ```

Mitt första program: en oändlig loop på ABC80

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

Mitt första program: en oändlig loop på ABC80

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

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

Loopa genom elementen i en vektor

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

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

`for ... do ...` gör så att följande händer:

- Plocka ut **varje element** ur samlingen.
- **Namnet** före pilen (här g) **refererar** till ett **nytt** värde för varje runda i loopen.
- Detta namn motsvarar en **lokal `val`**-variabel.

Bygg ny samling från befintlig med for-yield-uttryck

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

```
for g <- grönsaker yield s"god $g"
```

```
1  scala> val grönsaker = Vector("gurka","tomat","paprika","selleri")
2
3  scala> val åsikter = for g <- grönsaker yield s"god $g"
4  val åsikter: Vector[String] =
5    Vector(god gurka, god tomat, god paprika, god selleri)
```

Samlingen Range håller reda på intervall

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

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

- Men alla värden däremellan skapas inte förrän de behövs:

```
1  scala> val jättestortIntervall = Range(0, Int.MaxValue)
2  val jättestortIntervall: Range.Exclusive = Range 0 until 2147483647
3
4  scala> jättestortIntervall.end
5  val res1: Int = 2147483647
6
7  scala> jättestortIntervall.toVector
8  java.lang.OutOfMemoryError: Java heap space
```

Loopa med Range

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

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

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

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

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

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

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

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

Loopa med Range skapad med `to`

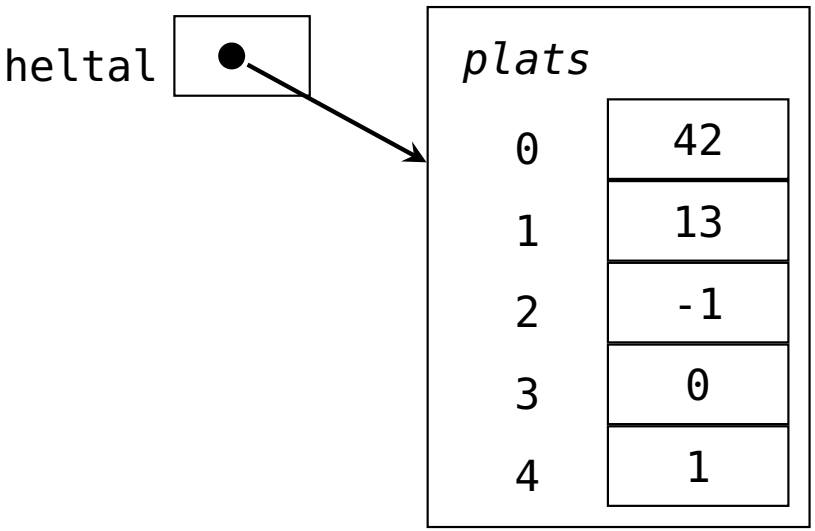
Med `to` efter ett heltal får du en Range till och **med** sista:

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

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

Vad är en `Array`?

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

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



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

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

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

Några likheter mellan `Vector` och `Array`

- Båda är samlingar som kan innehålla många element.
- Med båda kan man snabbt accessa vilket element som helst: `xs(2)`

Några viktiga skillnader:

Vector

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

Array

- Är **föränderlig**: `xs(2) = 42`
- Är **snabb** om man bara vill läsa eller skriva på befintliga platser.
- Kan **ej** ändra storlek: tillägg eller borttagning mitt i kräver **långsam** kopiering av resten.

Fristående applikation

Kompilering i terminalen

När du ska skriva kod i en editor, kompilera i terminalen och köra ditt program som en **fristående applikation**, så behövs:

- En editor: **VS Code** med tillägget **Scala (Metals)**
- Körmiljön **OpenJDK**
- Kommandoverktyg i terminalen: `scala` eller `scala-cli`
- Installera så här: `https://lunduniversity.github.io/pgk/#verktyg`
- Läs mer i Appendix C.
- Tips om du kör Windows: installera nya Windows Terminal

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

Scala Command Line Interface (CLI)

- Utvecklingen av ett nytt kommandogränssnitt (eng. *Command Line Interface (CLI)*) för Scala startades 2022 i ett öppenkällkodsprojekt som leds av Virtuslab.
- I augusti 2024 blev **`scala-cli`** det nya **`scala`**
- Du kan nu ersätta `scala-cli` med `scala`
- Läs mer i Appendix C och F, samt här: `https://scala-cli.virtuslab.org/`
- Se vad Scala CLI kan göra med underkommandot `help`

```
scala help
```

Ett minimalt fristående program i Scala

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

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

Kompilera och kör i terminalen:

```
> scala run hej.scala
Compiling project (Scala 3.7.2, JVM (21))
Compiled project (Scala 3.7.2, JVM (21))
Hej Scala!
```

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

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

Loopa genom en samling med en `while`-sats

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

scala> xs.size
val res0: Int = 4

scala> var i = 0
val i: Int = 0

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

Strängargument till i ett program med primitiv main

Skriv och spara nedan kod i filen `helloargs1scala`

```
> code helloargs1.scala
```

```
object HelloScalaArgs:
  def main(args: Array[String]): Unit = // en primitiv main-metod utan @main
    var i = 0
    while i < args.size do
      println(args(i))
      i = i + 1
```

En primitiv `main`-metod har ej `@main` och måste vara i ett objekt.
Kompilera och kör med programargument efter `--`

```
1 > scala run helloargs1.scala -- morot gurka tomat
2 morot
3 gurka
4 tomat
```

Typsäkra argument till i ett program med @main

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

```
> code helloargs2.scala
```

```
@main def hej(heltal: Int, resten: String*): Unit =  // notera * efter String
  for i <- 0 until heltal do println(resten(i))
```

Med `@main` behövs inget objekt.
Kompilera och kör med programargument efter `--`

```
1 > scala run helloargs2.scala -- 2 morot gurka tomat
2 morot
3 gurka
4 > scala run helloargs2.scala -- aj morot gurka tomat
5 Illegal command line: java.lang.NumberFormatException: For input string: "aj"
```

Med `@main` genereras automatiskt en primitiv main som kollar att argumenten har rätt typ.

För kännedom: Scala-**skript**

- Scala-kod kan köras som ett **skript**.
- Ett skript finns i en enda fristående fil med ändelsen `.sc`
- Skript behöver inget huvudprogram.
- Skript har automatiskt alla programargument i strängsekvensen `args`

```
// spara detta i filen 'myscript.sc'
println("Hej alla mina argument:")
for a <- args do println(s"Hej: $a")
```

```
> scala run myscript.sc -- ett två tre
Hej alla mina argument:
Hej: ett
Hej: två
Hej: tre
```

För kännedom: Scala-**skript**

- Scala-kod kan köras som ett **skript**.
- Ett skript finns i en enda fristående fil med ändelsen `.sc`
- Skript behöver inget huvudprogram.
- Skript har automatiskt alla programargument i strängsekvensen `args`

```
// spara detta i filen 'myscript.sc'
println("Hej alla mina argument:")
for a <- args do println(s"Hej: $a")
```

```
> scala run myscript.sc -- ett två tre
Hej alla mina argument:
Hej: ett
Hej: två
Hej: tre
```

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

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

Algoritmer: stegvisa lösningar

Vad är en algoritm?

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

Exempel:

- baka en kaka

Vad är en algoritm?

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

Exempel:

- baka en kaka
- räkna ut din pensionsprognos

Vad är en algoritm?

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

Exempel:

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

Vad är en algoritm?

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

Exempel:

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

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

Varför?

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

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

Algoritm:

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

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

Algoritm:

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

   `annars`
   - Skriv ”Grattis!”

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

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

Algoritm:

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

   `annars`
   - Skriv ”Grattis!”

Hittar du buggen?

Algoritm-exempel: HIGHSCORE

Problem: Kolla om high-score i ett spel

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

Algoritm:

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

   `annars`
   - Skriv ”Grattis!”

Hittar du buggen?

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

HIGHSCORE implementerad i Scala

```
import scala.io.StdIn.readLine

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

HIGHSCORE implementerad i Scala

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

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

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

HIGHSCORE implementerad i Scala

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

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

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

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

Algoritmexempel: N-FAKULTET

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

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

Algoritmexempel: N-FAKULTET

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

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

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

Algoritmexempel: MIN

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

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

Algoritmexempel: MIN

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

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

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

Funktioner skapar struktur

Mall för funktionsdefinitioner

```
def funktionsnamn(parameterdeklarationer): returtyp = uttryck
```

Mall för funktionsdefinitioner

`def` funktionsnamn(parameterdeklarationer): returtyp = uttryck

Exempel:

```
def öka(i: Int): Int = i + 1
```

Mall för funktionsdefinitioner

`def` funktionsnamn(parameterdeklarationer): returtyp = uttryck

Exempel:

```
def öka(i: Int): Int = i + 1
```

Returtypen kan härledas av kompilatorn:

```
def öka(i: Int) = i + 1
```

Men för att få hjälp av kompilatorn är det bra att ange returtyp!

Mall för funktionsdefinitioner

`def` funktionsnamn(parameterdeklarationer): returtyp = uttryck

Exempel:

```
def öka(i: Int): Int = i + 1
```

Returtypen kan härledas av kompilatorn:

```
def öka(i: Int) = i + 1
```

Men för att få hjälp av kompilatorn är det bra att ange returtyp!
Om flera parametrar använd kommatecken. Om flera satser använd indentering (och eventuell valfria klammerparenteser).

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

Mall för funktionsdefinitioner

`def` funktionsnamn(parameterdeklarationer): returtyp = uttryck

Exempel:

```
def öka(i: Int): Int = i + 1
```

Returtypen kan härledas av kompilatorn:

```
def öka(i: Int) = i + 1
```

Men för att få hjälp av kompilatorn är det bra att ange returtyp!
Om flera parametrar använd kommatecken. Om flera satser använd indentering (och eventuell valfria klammerparenteser).

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

Ovan funktion har **sidoeffekten** att skriva ut en smiley.

Bättre många små abstraktioner som gör en sak var

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

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

Bättre många små abstraktioner som gör en sak var

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

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

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

Bättre många små abstraktioner som gör en sak var

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

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

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

- Denna bättre `isHighscore` är nu en **äkta funktion** som alltid ger samma svar för samma inparametrar och **saknar sidoeffekter**; dessa funktioner är ofta lättare att förstå.
- Funktioner som ger ett booleskt värde kallas för **predikat**.

Vad är ett block?

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

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

Vad är ett block?

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

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

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

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

Namn i block blir **lokala**

Synlighetsregler:

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

```
1  scala> def a = { val lokaltNamn = 42; println(lokaltNamn) }
2  scala> a
3  42
4
5  scala> println(lokaltNamn)
6  1 |println(lokaltNamn)
7    |        ^^^^^^^^^^
8    |        Not found: lokaltNamn
9
10 scala> def b = { val x = 42; { val x = 76; println(x) }; println(x) }
11 scala> def c = { val x = 42; { val b = x + 1; println(b) } }
12 scala> b  // vad händer?
13 scala> c  // vad händer?
```

Parameter och argument

Skilj på parameter och argument!

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

```
scala> val ettArgument = 42

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

scala> öka(ettArgument)
```

Speciell syntax: anrop med s.k. **namngivet argument**

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

Namngivna argument kan ges i valfri ordning; då riskerar man inte fel ordning.

Procedurer

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

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

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

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

scala> :type x
Unit

scala> println(x)    // vad händer?
```

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

”Ingenting” *är* faktiskt någonting i Scala

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

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

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

Problemlösning: nedbrytning i abstraktioner som sen kombineras

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

Exempel på **funktionell nedbrytning**

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

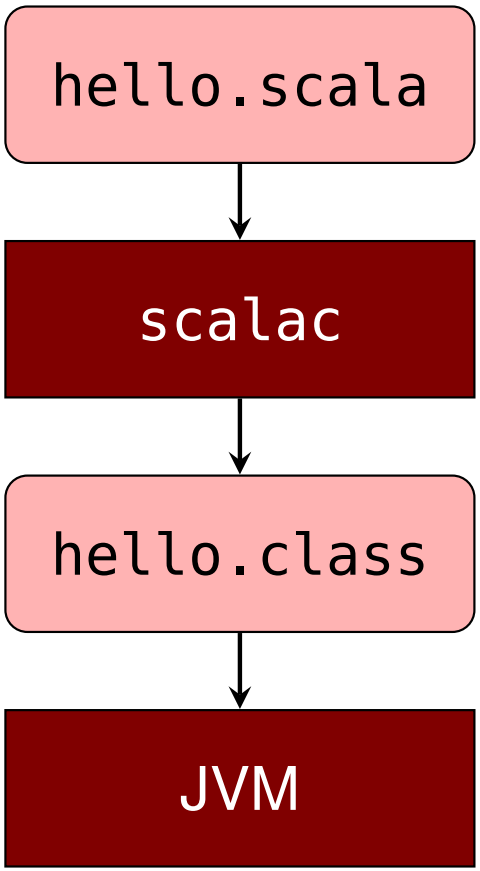
```
// skapa abstraktioner som bygger på varandra

def kvadrat = upprepa(4){fram; höger}

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

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

// huvudprogram

sudda; sakta(200)
rutnät
```

Varför abstraktion?

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

Katalogstruktur för kodfiler med paket

Från källkod till maskinkod med JVM



Källkodsfil

Kompilatorn skapar *abstrakt* maskinkod (s.k. bytekod)

`.class`-fil med bytekod

Java Virtual Machine
Översätter bytekod till *konkret* maskinkod som passar din specifika CPU **under körning** (s.k. interpretering)

Paket

```
package greeting

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

- Paket (eng. *package*) skapar namnrymder och i en hierarkisk struktur.
- Paket kan vara **nästlade**: ofta finns paket i paket i paket.
- Paket är speciellt bra om man har mycket kod i många kodfiler.
- Kompilatorn placerar maskinkoden i kataloger enligt paketstrukturen.[2]
  Är du nyfiken, kolla underkataloger i `.scala-build`:

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

[2] Katalogstrukturen för källkoden *måste* i många andra språk, t.ex. Java, *exakt motsvara paketstrukturen*, men detta är inte nödvändigt i Scala – alla Scala-kodfiler kan ligga i samma katalog på toppnivå eller i underkatalog med valfritt namn, oavsett hur din kod använder **`package`**.

Import

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

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

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

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

...gör så att namnet syns **direkt**, och man slipper skriva hela vägen till namnet:

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

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

Jar-filer

- `jar`-filer liknar `zip`-filer och används för att sammanföra många kompilerade kodfiler i **en komprimerad fil** för enkel distribution och körning.
- Du använder jar-filer med optionen `--jar`

```
scala run . --jar introprog.jar
```

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

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

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

Att göra denna vecka

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

Veckans övning: `programs`

- Kunna skapa, kompilera och köra en enkel applikation i terminalen.
- Kunna skapa samlingarna Range, Array och Vector med heltal och strängar.
- Kunna indexera i en indexerbar samling, t.ex. Array och Vector.
- Kunna anropa operationerna size, mkString, sum, min, max på samlingar som innehåller heltal.
- Känna till skillnader och likheter mellan samlingarna Range, Array och Vector.
- Förstå skillnaden mellan en while-sats och ett for-uttryck.
- Kunna skapa samlingar med heltalsvärden som resultat av enkla for-uttryck.
- Förstå skillnaden mellan en algoritm i pseudo-kod och dess implementation.
- Kunna implementera algoritmerna SUM, MIN, MAX med en indexerbar samling och en while-sats.

Labb läsvecka 3: `irritext`

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

Alla kodar loss!

Vi ses nästa vecka!