Vecka 3. Funktioner och abstraktion

Programmering, grundkurs (pgk)

Björn Regnell

Datavetenskap, LTH, Lunds universitet
https://cs.lth.se/pgk

EDAA45, Lp1-2, HT2023

Kompilerad den 7 november 2023

3 Funktioner och abstraktion

- Abstraktion
- Vad är en funktion?
- Hur ser det ut i minnet när funktioner anropas?
- En funktion är ett värde
- Äkta funktioner
- Anonyma funktioner
- Skapa din egen kontrollstruktur
- Kort om rekursion
- Automatisera kompilering
- Veckans uppgifter

Hämta beställda bokpaket!

- Du som beställt **bokpaket** men **ännu inte** hämtat ut det:

  Gå till datavetenskaps **expedition** på andra våningen i E-huset trapphus A på expeditionstid eller mejla `birger.swahn@cs.lth.se` för överenskommelse om annan tid.

Kursombud

- Om du är intresserad: fyll i **enkät** om **kursombud** i Canvas.
- Kursombud träffar vid behov kursansvarig på rast mellan föreläsningar eller chattar med varandra och kursansvarig på Discord **under kursens gång**.
- Kursombud träffar kursansvarig och programledning **efter kursen** och diskuterar **kursutvärderingen** CEQ.
- Det vore bra med minst 2st D:are och minst 2st C:are och gärna minst 1st W:are.
- Läs mer om studierådet här:
  `https://www.dsek.se/committees/srd`

Abstraktion

Vad är abstraktion?

- **Abstraktion** innebär att skapa en förenklad **modell** ur konkreta detaljer
- Vi ”hittar på” nya **begrepp** som ger oss återanvändbara ”byggblock” för våra tankar och vår kommunikation
- Vi får ett abstrakt **namn** som kan användas i stället för en massa **konkreta detaljer**
- Skilj på abstraktionens **namn** (begrepp, koncept), dess **användning** (anrop) och dess detaljerade **beskrivning** (definition, implementation)
- **Funktioner** (som du redan känner från matematiken) är en av våra **viktigaste** abstraktionsmekanismer

`https://sv.wikipedia.org/wiki/Abstraktion`
`https://en.wikipedia.org/wiki/Abstraction`

Exempel på abstraktionsmekanismer inom datavetenskapen

Vi kommer att behandla flera olika, alltmer **kraftfulla** abstraktionsmekanismer i denna kurs:

- Funktioner
- Objekt
- Klasser
- Arv
- Generiska strukturer
- Kontextuella abstraktioner

Dessa abstraktionsmekanismer blir **extra kraftfulla** om de **kombineras**!

Vad är en funktion?

Om veckans tema: Funktioner

- Funktioner är en av de viktigaste abstraktionsmekanismerna inom datavetenskapen
- Du kan redan massor om funktioner, bl.a. från matematiken.
- Denna vecka ska vi fördjupa förståelsen:
  - överlagring
  - enhetlig access
  - defaultargument
  - namngivna argument
  - lokala funktioner
  - funktioner som äkta värden
  - anonyma funktioner
  - klammerparentes vid ensam paramenter
  - multipla parameterlistor
  - egendefinierade kontrollstrukturer
  - fördröjd evaluering (”call-by-name”)
  - stegade funktioner (”Curry-funktioner”)
  - fångad variablelrymd (”closure”)

Om veckans tema: Funktioner

Funktion: deklaration och anrop

`def` funktionsnamn(parameterdeklarationer): returtyp = uttryck

- En funktion har ett **huvud** och efter = kommer dess **kropp**.
- En **namngiven** funktion **deklareras** med nyckelordet **`def`**
- En funktion kan ha **parametrar** som deklareras i huvudet.
- **Kroppen** ska vara ett **uttryck** (ev. ett block med flera uttryck).
- **Parametrar** binds till **argument** vid **anrop**.
- Uttrycket i funktionens kropp **evalueras** vid **varje anrop**.
- Värdet av uttrycket blir funktionen **returvärde**.

Funktion: deklaration och anrop

`def` funktionsnamn(parameterdeklarationer): returtyp = uttryck

- En funktion har ett **huvud** och efter = kommer dess **kropp**.
- En **namngiven** funktion **deklareras** med nyckelordet **`def`**
- En funktion kan ha **parametrar** som deklareras i huvudet.
- **Kroppen** ska vara ett **uttryck** (ev. ett block med flera uttryck).
- **Parametrar** binds till **argument** vid **anrop**.
- Uttrycket i funktionens kropp **evalueras** vid **varje anrop**.
- Värdet av uttrycket blir funktionen **returvärde**.

Exempel:

```
def öka(a: Int, b: Int): Int = a + b
```

Funktion: deklaration och anrop

`def` funktionsnamn(parameterdeklarationer): returtyp = uttryck

- En funktion har ett **huvud** och efter = kommer dess **kropp**.
- En **namngiven** funktion **deklareras** med nyckelordet **`def`**
- En funktion kan ha **parametrar** som deklareras i huvudet.
- **Kroppen** ska vara ett **uttryck** (ev. ett block med flera uttryck).
- **Parametrar** binds till **argument** vid **anrop**.
- Uttrycket i funktionens kropp **evalueras** vid **varje anrop**.
- Värdet av uttrycket blir funktionen **returvärde**.

Exempel:

```
def öka(a: Int, b: Int): Int = a + b
```

```
scala> öka(42, 1)
val res0: Int = 43
```

Deklarera funktioner, överlagring

- En parameter, och sedan två parametrar:

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

- Båda funktionerna ovan kan finnas samtidigt! Trots att de har **samma namn** är de **olika funktioner**; kompilatorn kan skilja dem åt med hjälp av de **olika parameterlistorna**.
- Detta kallas **överlagring** (eng. *overloading*) av funktioner.
- Överlagring ger **flexibilitet i användningen**; vi slipper hitta på nytt namn så som öka2 vid 2 parametrar.

Funktioner med defaultargument

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

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

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

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

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

- Om ett argument utelämnas och parametern deklarerats med defaultargument så appliceras detta. Kompilatorn fyller alltså i argumentet åt oss, om det är entydigt vilken parameter som avses.

Funktioner med namngivna argument

- Genom att använda **namngivna argument** behöver man inte hålla reda på ordningen på parametrarna, bara man känner till parameternamnen.
- Namngivna argument går fint att **kombinera** med defaultargument.

```
scala> def namn(förnamn: String,
                efternamn: String,
                förnamnFörst: Boolean = true,
                ledtext: String = "Namn:"): String =
          if förnamnFörst then s"$ledtext $förnamn $efternamn"
          else s"$ledtext $efternamn, $förnamn"

scala> namn(ledtext = "Name:", efternamn = "Coder", förnamn = "Kim")
val res0: String = Name: Kim Coder
```

Enhetlig access

- Om en funktion **deklareras med** tom parameterlista `()` så ska den **anropas med** tom parameterlista.

```
scala> def tomParameterlista() = 42

scala> tomParameterlista()
val res1: Int = 42

scala> tomParameterlista
1 |tomParameterlista
  |^^^^^^^^^^^^^^^^^
  |method tomParameterlista must be called with () argument
```

- En parameterlös funktion deklarerad **utan** `()` ska anropas **utan** `()`.

```
scala> def ingenParameterlista = 42
scala> ingenParameterlista()
1 |ingenParameterlista()
  |^^^^^^^^^^^^^^^^^^^
  |method ingenParameterlista does not take parameters
```

Enhetlig access

- Om en funktion **deklareras med** tom parameterlista `()` så ska den **anropas med** tom parameterlista.

```
scala> def tomParameterlista() = 42

scala> tomParameterlista()
val res1: Int = 42

scala> tomParameterlista
1 |tomParameterlista
  |^^^^^^^^^^^^^^^^^
  |method tomParameterlista must be called with () argument
```

- En parameterlös funktion deklarerad **utan** `()` ska anropas **utan** `()`.

```
scala> def ingenParameterlista = 42
scala> ingenParameterlista()
1 |ingenParameterlista()
  |^^^^^^^^^^^^^^^^^^^
  |method ingenParameterlista does not take parameters
```

- Deklaration utan `()` möjliggör **enhetlig access**: implementationen kan ändras från **`val`** till **`def`** eller tvärtom, **utan** att **användandet** påverkas.

Hur ser det ut i minnet när funktioner anropas?

Anropsstacken och objektheapen

Minnet som innehåller ett programs data är uppdelat i två delar:

- **Anropsstacken**:
  - På anropsstacken läggs en **aktiveringspost** (eng. *stack frame*[1], *activation record*) för varje funktionsanrop med plats för **parametrar** och **lokala variabler**.
  - Aktiveringsposten **raderas** när **returvärdet** har levererats.
  - Stacken **växer** vid **nästlade funktionsanrop**, då en funktion i sin tur anropar en annan funktion.
- **Objektheapen**: I objektheapen[2,3] sparas alla objekt (data) som allokeras under körning. Heapen städas då och då av **skräpsamlaren** (eng. *garbage collector*), och minne som inte används längre frigörs.

[1] en.wikipedia.org/wiki/Call_stack
[2] en.wikipedia.org/wiki/Memory_management
[3] Ej att förväxlas med datastrukturen heap sv.wikipedia.org/wiki/Heap

Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
scala> def h(x: Int, y: Int) = { val z = x + y; println(z) }

scala> def g(a: Int, b: Int) = { val x = 1; h(x + 1, a + b) }

scala> def f() = { val n = 5; g(n, 2 * n) }

scala> f()
```

Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
scala> def h(x: Int, y: Int) = { val z = x + y; println(z) }

scala> def g(a: Int, b: Int) = { val x = 1; h(x + 1, a + b) }

scala> def f() = { val n = 5; g(n, 2 * n) }

scala> f()
```

Stacken

variabel	värde	Aktiveringspost för anrop av...

Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
scala> def h(x: Int, y: Int) = { val z = x + y; println(z) }

scala> def g(a: Int, b: Int) = { val x = 1; h(x + 1, a + b) }

scala> def f() = { val n = 5; g(n, 2 * n) }

scala> f()
```

Stacken

variabel	värde	Aktiveringspost för anrop av...
n	5	f

Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
scala> def h(x: Int, y: Int) = { val z = x + y; println(z) }

scala> def g(a: Int, b: Int) = { val x = 1; h(x + 1, a + b) }

scala> def f() = { val n = 5; g(n, 2 * n) }

scala> f()
```

Stacken

variabel	värde	Aktiveringspost för anrop av...
n	5	f
a	5	g
b	10	
x	1	

Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
scala> def h(x: Int, y: Int) = { val z = x + y; println(z) }

scala> def g(a: Int, b: Int) = { val x = 1; h(x + 1, a + b) }

scala> def f() = { val n = 5; g(n, 2 * n) }

scala> f()
```

Stacken

variabel	värde	Aktiveringspost för anrop av...
n	5	f
a b x	5 10 1	g
x y z	2 15 17	h

Vad är en stack trace?

När du letar buggar vid körtidsfel har du nytta av att **noga studera utskriften av anropsstacken** (eng. *stack trace*):

```
1  // Program i filen bmi.scala
2
3  @main
4  def bmi(heightCm: Int, weightKg: Int) =
5    safeDiv(weightKg, heightCm * heightCm)
6
7  def safeDiv(numerator: Int, denominator: Int): (Int, String) =
8    if denominator == 0 then (numerator / denominator, "")  // ser du buggen?
9    else (0, "division by zero")
```

```
1  > scala-cli run bmi.scala -- 0 42
2  Exception in thread "main" java.lang.ArithmeticException: / by zero
3          // HÄR KOMMER STACK TRACE pga körtidsfel - se nästa bild
```

Hur läsa en stack trace?

```
1  Exception in thread "main" java.lang.ArithmeticException: / by zero
2          at bmi$package$.safeDiv(bmi.scala:8)
3          at bmi$package$.bmi(bmi.scala:5)
4          at bmi.main(bmi.scala:3)
```

- En **stack trace** skrivs ut efter en krasch p.g.a. körtidsfel.
- Körtidsfel känns igen med ordet **Exception**.
- Först kommer en beskrivning av felet som orsakat kraschen, här: `java.lang.ArithmeticException: \ by zero`
- Därefter visas anropsstacken.
- För varje funktionsanrop anges: **`klass.metod(kodfil:radnummer)`**
- Main-funktioner läggs i ett singelobjekt i ett speciellt paket
- Singelobjekt i Scala kodas som en Java-klass med dollar-tecken efter namnet, eftersom det inte finns singelobjekt i JVM.

Lokala funktioner

Med lokala funktioner kan delproblem lösas med nästlade abstraktioner.

```
def gissaTalet(max: Int, min: Int = 1): Unit =
  def gissat = io.StdIn.readLine(s"Gissa talet mellan $min och $max: ").toInt

  val hemlis = (math.random() * (max - min) + min).toInt

  def skrivLedtrådOmEjRätt(gissning: Int): Unit =
    if gissning > hemlis then println(s"$gissning är för stort :(")
    else if gissning < hemlis then println(s"$gissning är för litet :(")

  def inteRätt(gissning: Int): Boolean =
    skrivLedtrådOmEjRätt(gissning)
    gissning != hemlis


  def loop: Int = { var i = 1; while inteRätt(gissat) do i += 1; i }

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

Lokala, nästlade funktionsdeklarationer är tyvärr inte tillåtna i många andra språk, t.ex. Java.[4]

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

En funktion är ett värde

Funktioner är äkta värden i Scala

- En funktion är ett **äkta värde**.
- Vi kan till exempel tilldela en variabel ett **funktionsvärde**.

Funktioner är äkta värden i Scala

- En funktion är ett **äkta värde**.
- Vi kan till exempel tilldela en variabel ett **funktionsvärde**.
- Med hjälp enbart funktionsnamnet får vi funktionen som ett **värde** (inga argument appliceras än):

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

scala> val f = add
val f: (Int, Int) => Int = Lambda7210/0x0000000841e4e040

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

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

Funktioner är äkta värden i Scala

- En funktion är ett **äkta värde**.
- Vi kan till exempel tilldela en variabel ett **funktionsvärde**.
- Med hjälp enbart funktionsnamnet får vi funktionen som ett **värde** (inga argument appliceras än):

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

scala> val f = add
val f: (Int, Int) => Int = Lambda7210/0x0000000841e4e040

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

- Ett funktionsvärde har en **typ** precis som alla värden:
  `f: (Int, Int) => Int`
- Ett funktionsvärde har till skillnad från en funktionsdeklaration inget namn (variabeln `f` har ett namn men inte själva funktionen). Den kallas därför en **anonym** funktion eller **lambda** (mer om detta snart).

Funktionsvärden kan vara argument

En funktion kan ha en annan funktion som parameter:

```
1  scala> def tvåGånger(x: Int, f: Int => Int) = f(f(x))
2
3  scala> def öka(x: Int) = x + 1
4
5  scala> def minska(x: Int) = x - 1
6
7  scala> tvåGånger(42, öka)
8  val res1: Int = 44
9
10 scala> tvåGånger(42, minska)
11 val res1: Int = 40
```

Applicera funktioner på element i samlingar med `map`

```
def öka(x: Int) = x + 1

def minska(x: Int) = x - 1

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

Applicera funktioner på element i samlingar med `map`

```
def öka(x: Int) = x + 1

def minska(x: Int) = x - 1

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

Metoden **`map`** fungerar på alla Scala-samlingar och tar **en funktion som argument** och applicerar denna funktion på alla element och **skapar en ny samling** med resultaten:

```
1  scala> xs.map(öka)
2  val res0: ???   // vad blir resultatet?
3
4  scala> xs.map(minska)
5  val res1: ???   // vad blir resultatet?
```

En funktion som har funktionsvärden som indata (eller utdata) kallas en **högre ordningens funktion** (eng. *higher-order function*).

Applicera funktioner på element i samlingar med `map`

```
def öka(x: Int) = x + 1

def minska(x: Int) = x - 1

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

Metoden **`map`** fungerar på alla Scala-samlingar och tar **en funktion som argument** och applicerar denna funktion på alla element och **skapar en ny samling** med resultaten:

```
1  scala> xs.map(öka)
2  val res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
3
4  scala> xs.map(minska)
5  val res1: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
```

En funktion som har funktionsvärden som indata kallas en **högre ordningens funktion** (eng. *higher-order function*).

Äkta funktioner

Äkta funktioner

- En **äkta** (eng. *pure*) funktion är en funktion som ger ett resultat som **enbart** beror av dess argument. Alltså som funktioner i matematiken.
- En äkta (matematisk) funktion är **referentiellt transparent** (eng. *referentially transparent*), vilket innebär att varje anrop kan bytas ut mot funktionskroppen där parametrarna ersatts med motsvarande argument.
- En äkta funktion har **inga sidoeffekter**, t.ex. utskrift, skriva/läsa filer, eller uppdateringar av variabler **synliga utanför** funktionen.
- Exempel:

```
def add(x: Int, y: Int): Int = x + y                // äkta funktion
def rnd(n: Int): Int = (math.random() * n).toInt  // oäkta funktion
```

  - Uttrycket `add(41, 1)` kan ersättas med 41 +1 som i sin tur kan ersättas med 42 utan att det påverkar resultatet. Resultatet av `add(41, 1)` blir **samma varje gång** funktionen appliceras med dessa argument
  - Uttrycket `rnd(42)` kan **inte** bytas ut mot ett specifikt uttryck som säkert ger samma resultat varje gång. Alltså: *ej referentiell transparens*.

Exempel på oäkta funktioner: slumptal

- Funktioner vars värden på något sätt beror av slumpen är **inte** äkta funktioner.
- Även om samma argument ges vid upprepad applicering, så kan ju resultatet bli olika.
- Studera dokumentationen för `scala.util.Random` här:
  https://www.scala-lang.org/api/current/scala/util/Random.html
- Du har nytta av funktionen `Random.nextInt` och slumptalsfrö (eng. *random seed*) i veckans uppgifter.

Slumptalsfrö: få samma slumptal varje gång

- Om man använder slumptal kan det vara svårt att leta buggar, eftersom det blir **olika varje gång** man kör programmet och buggen kanske bara uppstår ibland.
- Med klassen `scala.util.Random` kan man skapa **pseudo**-slumptalssekvenser.

Slumptalsfrö: få samma slumptal varje gång

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

```
val seed = 42
val rnd = util.Random(seed) // skapa ny slumpgenerator med frö 42
val r = rnd.nextInt(6) // ger slumptal mellan 0 till och med 5
```

Slumptalsfrö: få samma slumptal varje gång

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

```
val seed = 42
val rnd = util.Random(seed) // skapa ny slumpgenerator med frö 42
val r = rnd.nextInt(6) // ger slumptal mellan 0 till och med 5
```

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

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

Anonyma funktioner

Anonyma funktioner

- Man behöver inte ge funktioner namn. De kan i stället skapas med hjälp av **funktionslitteraler**.[5]
- En funktionslitteral har ...
  1. en parameterlista (utan funktionsnamn, utan returtyp),
  2. sedan den reserverade teckenkombinationen **=>**
  3. och sedan ett uttryck (eller ett block).

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

Anonyma funktioner

- Man behöver inte ge funktioner namn. De kan i stället skapas med hjälp av **funktionslitteraler**.[5]
- En funktionslitteral har ...
  1. en parameterlista (utan funktionsnamn, utan returtyp),
  2. sedan den reserverade teckenkombinationen **=>**
  3. och sedan ett uttryck (eller ett block).
- Exempel:

```
(x: Int, y: Int) => x + y                    // vilken typ?
```

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

Anonyma funktioner

- Man behöver inte ge funktioner namn. De kan i stället skapas med hjälp av **funktionslitteraler**.[5]
- En funktionslitteral har ...
  1. en parameterlista (utan funktionsnamn, utan returtyp),
  2. sedan den reserverade teckenkombinationen **=>**
  3. och sedan ett uttryck (eller ett block).
- Exempel:

```
(x: Int, y: Int) => x + y                    // vilken typ?
```

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

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

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

Applicera anonyma funktioner på element i samlingar

Anonym funktion skapad med funktionslitteral direkt i anropet:

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

Applicera anonyma funktioner på element i samlingar

Anonym funktion skapad med funktionslitteral direkt i anropet:

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

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

Eftersom kompilatorn här kan härleda typerna så behövs de inte:

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

Applicera anonyma funktioner på element i samlingar

Anonym funktion skapad med funktionslitteral direkt i anropet:

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

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

Eftersom kompilatorn här kan härleda typerna så behövs de inte:

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

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

```
scala> xs.map(_ + 1)
res2: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
```

Platshållarsyntax för anonyma funktioner

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

```
_ + 1
```

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

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

Platshållarsyntax för anonyma funktioner

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

```
_ + 1
```

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

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

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

```
_ + _
```

Platshållarsyntax för anonyma funktioner

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

```
_ + 1
```

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

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

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

```
_ + _
```

... expanderas till:

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

Exempel på platshållarsyntax med `reduceLeft`

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

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

Predikat, med och utan namn

- En funktion som har `Boolean` som returtyp kallas för ett **predikat**.
- Exempel:

```
def isTooLong(name: String): Boolean = name.length > 10

def isTall(heightInMeters: Double, limit: Double = 1.78): Boolean =
  heightInMeters > limit
```

- Predikat ges ofta ett namn som börjar på `is` eller `has` så att man lätt kan se att det är ett predikat när man läser kod som anropar funktionen.
- Många av samlingsmetoderna i Scalas standardbibliotek tar predikat som funktionsargument. Exempel med predikat som anonym funktion:

```
scala> val parts = Vector(3, 1, 0, 5).partition(_ > 1)
val parts: (Vector[Int], Vector[Int]) =
  (Vector(3, 5),Vector(1, 0))
```

- Studera snabbreferensen och försök hitta samlingsmetoder som tar predikat som funktionsargument. `http://cs.lth.se/pgk/quickref`
  I anropsexempel med predikat-argument används bokstaven `p`.

Funktionsvärde vid tom parameterlista: använd ”thunk”

- Om du vill ha funktionen som ett värde så skriv bara namnet och inte parameterlistan (samma exempel som tidigare):

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

scala> val f = add     // inget anrop sker
val f: (Int, Int) => Int = Lambda7210/0x0000000841e4e040@1ce2db23
```

- Vid **tom parameterlista** behövs anonym funktion som **fördröjer anrop**:

```
scala> def a() = 42
def a(): Int

scala> val b = a
1 |val b = a
  |         ^
  |         method a must be called with () argument

scala> val b = () => a()   // anonym funktion, fördröjd evaluering
val b: () => Int = Lambda7214/0x0000000841e50440@565d794
```

- Notera typen: `()` **`=>`** `Int` Ett sådant funktionsvärde kallas **thunk**
  `https://en.wikipedia.org/wiki/Thunk`

Skapa din egen kontrollstruktur

Hur fungerar egentligen `upprepa` i Kojo?

```
upprepa(10) {
  println("hej")
}
```

Hur fungerar egentligen `upprepa` i Kojo?

```
upprepa(10) {
  println("hej")
}
```

Vi ska nu se hur vi, genom att kombinera ett antal koncept, kan skapa egna kontrollstrukturer likt upprepa ovan:

- klammerparentes vid ensam paramenter
- multipla parameterlistor
- namnanrop (fördröjd evaluering)

Multipla parameterlistor

Vi har tidigare sett att man kan ha mer än en parameter:

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

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

Man kan även ha **mer än en parameterlista**:

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

scala> add(21)(21)
res1: Int = 42
```

(eng. *multiple parameter lists*)

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

Värdeanrop och namnanrop

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

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

Värdeanrop och namnanrop

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

```
scala> def byValue(n: Int): Unit = for i <- 1 to n do print(" " + n)

scala> byValue(21 + 21)
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42

scala> byValue({print(" hej"); 21 + 21})
 hej 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4
```

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

```
scala> def byName(n: => Int): Unit = for i <- 1 to n do print(" " + n)

scala> byName({print(" hej"); 21 + 21})
 hej hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej 42 hej
```

Kluring: Varför skrivs ”hej” ut en extra gång i början?

Värdeanrop och namnanrop

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

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

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

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

Kluring: Varför skrivs ”hej” ut en extra gång i början? ledtråd: `1 to n`

Klammerparenteser vid ensam parameter

Så här har vi sett nyss att man man göra:

```
1 scala> def twice(action: => Unit): Unit = { action; action }
2
3 scala> twice( { print("hej"); print("san ") } )
4 hejsan hejsan
```

Det ser rätt klyddigt ut med { (och) } eller vad tycker du?

Klammerparenteser vid ensam parameter

Så här har vi sett nyss att man man göra:

```
1 scala> def twice(action: => Unit): Unit = { action; action }
2
3 scala> twice( { print("hej"); print("san ") } )
4 hejsan hejsan
```

Det ser rätt klyddigt ut med `{ (` och `) }` eller vad tycker du?
Men... För alla funktioner `f` gäller att:
det är helt ok att byta ut vanliga parenteser: `f(uttryck)`
mot krullparenteser: `f{uttryck}`
om parameterlistan har **exakt en** parameter.

Man kan alltså skippa det yttre parentesparet för bättre läsbarhet:

```
scala> twice { print("hej"); print("san ") }
```

Skapa din egen kontrollstruktur

- Genom att **kombinera multipla parameterlistor** med **namnanrop** med **klammerparentes vid ensam parameter** kan vi skapa vår egen kontrollstruktur: `upprepa`

Skapa din egen kontrollstruktur

- Genom att **kombinera multipla parameterlistor** med **namnanrop** med **klammerparentes vid ensam parameter** kan vi skapa vår egen kontrollstruktur: `upprepa`

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

Hur då?

Skapa din egen kontrollstruktur

- Genom att **kombinera multipla parameterlistor** med **namnanrop** med **klammerparentes vid ensam parameter** kan vi skapa vår egen kontrollstruktur: `upprepa`

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

Hur då? Till exempel så här:

```
def upprepa(n: Int)(block: => Unit) = for i <- 0 until n do block
```

Skapa din egen kontrollstruktur

- Genom att **kombinera multipla parameterlistor** med **namnanrop** med **klammerparentes vid ensam parameter** kan vi skapa vår egen kontrollstruktur: `upprepa`

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

Hur då? Till exempel så här:

```
def upprepa(n: Int)(block: => Unit) = for i <- 0 until n do block
```

```
gurka gurka gurka tomat tomat gurka gurka gurka gurka to
```

Kolon vid ensam parameter

Du kan från Scala 3.3 i stället för klammerparentes vid ensam parameter använda kolon för att få färre ”krullisar” (eng. *fewer braces*).

```
upprepa(42):
  if math.random() < 0.5
  then print(" gurka")
  else print(" tomat")
```

Denna förenklade syntax föregicks av långa diskussioner innan den till slut accepterades.[6]

[6] Den nyfikne kan läsa förslaget före omröstning här: `https://docs.scala-lang.org/sips/fewer-braces.html`

Stegade funktioner, ”Curry-funktioner”

Om en funktion har multipla parameterlistor kan man skapa **stegade funktioner**, även kallat **partiellt applicerade** funktioner (eng. *partially applied functions*) eller **”Curry”-funktioner**.

```
scala> def add(x: Int)(y: Int) = x + y

scala> val öka = add(1)
val öka: Int => Int = Lambda7339/0x0000000841eb7040@19c8add7

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

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

Funktion med fångad variabelrymd: *closure*

```
def f(x: Int): Int => Int =
  val a = 42 + x
  def g(y: Int): Int = y + a
  g
```

Funktionen g **fångar** den lokala variabeln a i ett **funktionsobjekt**.

Funktion med fångad variabelrymd: *closure*

```
def f(x: Int): Int => Int =
  val a = 42 + x
  def g(y: Int): Int = y + a
  g
```

Funktionen g **fångar** den lokala variabeln a i ett **funktionsobjekt**.

```
scala> val funkis = f(1)
val funkis: Int => Int = Lambda7356/0x0000000841ed2840@1bda2

scala> funkis(2)
val res0: Int = 45
```

Funktion med fångad variabelrymd: *closure*

```
def f(x: Int): Int => Int =
  val a = 42 + x
  def g(y: Int): Int = y + a
  g
```

Funktionen g **fångar** den lokala variabeln a i ett **funktionsobjekt**.

```
scala> val funkis = f(1)
val funkis: Int => Int = Lambda7356/0x0000000841ed2840@1bda2

scala> funkis(2)
val res0: Int = 45
```

Ett funktionsobjekt med ”fångade” variabler kallas **closure**.
(Mer om funktioner som objekt senare.)

Översikt av begrepp vi gått igenom hittills

1. överlagring
2. utelämna tom parameterlista (enhetlig access)
3. defaultargument
4. namngivna argument
5. lokala funktioner
6. funktioner som äkta värden
7. anonyma funktioner
8. klammerparentes vid ensam paramenter
9. multipla parameterlistor
10. namnanrop (fördröjd evaluering)
11. egendefinierade kontrollstrukturer
12. stegade funktioner (”Curry-funktioner”)
13. fångad variablelrymd i funktionsobjekt (”closure”)

Kort om rekursion

Rekursiva funktioner

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

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

scala> fakultet(5)
val res0: Int = 120
```

- För varje nytt anrop läggs en ny aktiveringspost på stacken.
- I aktiveringsposten sparas varje returvärde som gör att `5 * (4 * (3 * (2 * 1)))` kan beräknas.
- Rekrusionen avbryts när man når **basfallet**, här `n < 2`
- En rekursiv funktion **måste** ha en returtyp.

Loopa med rekursion

```
def gissaTalet(max: Int, min: Int = 1): Unit =
  def gissat =
    io.StdIn.readLine(s"Gissa talet mellan [$min, $max]: ").toInt

  val hemlis = (math.random() * (max - min) + min).toInt

  def skrivLedtrådOmEjRätt(gissning: Int): Unit =
    if gissning > hemlis then println(s"$gissning är för stort :(")
    else if (gissning < hemlis) println(s"$gissning är för litet :(")

  def ärRätt(gissning: Int): Boolean =
    skrivLedtrådOmEjRätt(gissning)
    gissning == hemlis

  def loop(n: Int = 1): Int = if ärRätt(gissat) then n else loop(n + 1)

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

Rekursiva datastrukturer

- Datastrukturena Lista och Träd är exempel på datastrukturer som passar bra ihop med rekursion.
- Båda dessa datastrukturer kan beskrivas rekursivt:
  - En lista består av ett huvud och en lista, som i sin tur består av ett huvud och en lista, som i sin tur...
  - Ett träd består av grenar till träd som i sin tur består av grenar till träd som i sin tur, ...
- Dessa datastrukturer bearbetas med fördel med rekursiva algoritmer.
- I denna kursen ingår rekursion endast ”för kännedom”: du ska veta vad det är och kunna skapa en enkel rekursiv funktion, t.ex. fakultets-beräkning. Du kommer jobba mer med rekursion och rekursiva datastrukturer i fortsättningskursen.

Automatisera kompilering

Kompilera om det som ändrats vid varje sparning

- Den kreativa programmeringsprocessen innehåller många korta cykler av koda, ändra, testa.
- Det blir **många omkompileringar** och då vill man gärna slippa skriva samma kommando om och om igen.
- Vid **varje liten ändring** vill man **kompilera om** det som ändrats och se om det fortfarande kompilerar utan fel.
- Då kan du använda:
  `scala-cli compile . --watch`
  Ändringar bevakas och kompileras om direkt.

Veckans uppgifter

Mål med övning functions

- Kunna skapa och använda funktioner med en eller flera parametrar, default-argument, och namngivna argument.
- Kunna förklara nästlade funktionsanrop med aktiveringsposter på stacken.
- Kunna förklara skillnaden mellan äkta och ”oäkta” funktioner.
- Kunna applicera en funktion på alla element i en samling.
- Kunna använda funktioner som äkta värden.
- Kunna skapa och använda anonyma funktioner (ä.k. lambda-funktioner).
- Känna till att funktioner kan ha uppdelad parameterlista.
- Känna till att det går att partiellt applicera argument på funktioner med uppdelad parameterlista för att skapa s.k. stegade funktioner (ä.k. curry-funktioner).
- Känna till rekursion och kunna beskriva vad som kännetecknar en rekursiv funktion.
- Känna till att det går att skapa egna kontrollstrukturer med hjälp av namnanrop.
- Känna till skillnaden mellan värdeanrop och namnanrop.
- Kunna tolka en stack trace.

Mål med laboration irritext

- Kunna skapa ett större program med din egen kod efter dina egna idéer.
- Kunna använda en editor och terminalen för att iterativt editera, kompilera, och testa din kod.
- Kunna använda variabler i kombination med alternativ och repetetition i flera nivåer.
- Kunna stegvis förbättra din kod för att underlätta förändring och öka läsbarheten.
- Kunna skapa och använda abstraktioner för att generalisera och möjliggöra återanvändning av kod.

Ni ska spela **varandras** textspel i din **samarbetsgrupp**.
Läs labbinstruktioner:`http://cs.lth.se/pgk/compendium/`

Tips till ditt textspel.

```
"Yes".toLowerCase.startsWith("y")    // true
"hejsan".contains("ejsa")            // true
"42".toInt                           // 42
"?".toIntOption.getOrElse(42)        // 42
Thread.sleep(1000)                   // ger lagom irriterande fördröjning (1 sekund)

val i = 42
s"Livets mening är $i!" // dollar $ före namn vid stränginterpolering med s""
s"Livets mening är inte ${i-1}!"  // klamrar ${} vid evaluering av uttryck

"""|en sträng som spänner över
   |flera rader där marginalen fram till vertikalstreck
   |är bortplockad med stripMargin (kan kombineras med s-interpolatorn)
""".stripMargin

math.random() < 0.8                  // true i 80% av fallen
scala.util.Random.nextInt(42)      // ger slumptal mellan 0 och 41
scala.io.StdIn.readLine("prompt>") // ger sträng som användaren skriver

val x = try { "?".toInt } catch { case e: Exception => 42 }  // förhindrar krasch
Thread.sleep(1000)    // sova i 1000 milliskeunder
```

Kolla **snabbreferensen** vad mer du kan göra med strängar!

Exempel på en början till ett textspel

Här finns en exempel på en enkel *början* på ett textspel som du stegvis kan ändra och bygga ut till något du själv vill göra: `https://github.com/lunduniversity/introprog/tree/master/workspace/w03_irritext`

- Vilka begrepp och principer ger koden träning i?

Jobba så här

- Skriv koden i editorn vs code som du startar med `code .`
- Dra igång 3 olika terminalfönster:
  1. `scala-cli compile . --watch`
     så att din kod kompileras om automatiskt vid varje sparning med Ctrl+S.
  2. `scala-cli repl .`
     så att du kan göra mindre undersökningar rad för rad, medan du tänker.
     Ctrl+D avslutar REPL, så du kan börja om efter kodändring.
  3. `scala-cli run .`
     för att se om programmet funkar som det ska. Om programmet väntar på input kan du behöva avbryta för att köra om efter ändring.
- Börja enkelt och bygg vidare steg för steg.
- Bygg om koden allteftersom den växer genom att införa nya abstraktioner med väl valda namn (s.k. ”refaktorisering”) .
- Fixa **alla** kompileringsfel | | körtidsfel **innan** du går vidare.
- Fokusera på kodens **läsbarhet**.