

# EDAA45 Programmering, grundkurs

## Läsvecka 3: Funktioner, objekt

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

### 3 Funktioner, objekt

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

# Kursombud

# Fastställa kursombud

- Glädjande nog är det många intresserade!
- Instruktioner från studierådet:
  - min 2 max 4 D-are
  - min 2 max 4 W-are
- Vi lottar med lite lajvkodning inspirerat av:

```
1 scala> val kursombud = Vector("Kim Finkodare", "Robin Schnellhacker")  
2 scala> scala.util.Random.shuffle(kursombud).take(1)
```

# Funktioner

# Deklarera funktioner, överlagring

- En parameter, och sedan två parametrar:

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

- Båda funktionerna ovan kan finnas samtidigt! Trots att de har samma namn är de **olika** funktioner; kompilatorn kan skilja dem åt med hjälp av de olika parameterlistorna.
- Detta kallas **överlagring** (eng. *overloading*) av funktioner.

# Tom parameterlista och inga parametrar

- Om en funktion deklarerats med tom parameterlista ( ) kan den anropas på två sätt: med och utan tomma parenteser.

```
1 scala> def tomParameterLista() = 42
2
3 scala> tomParameterLista()
4 res2: Int = 42
5
6 scala> tomParameterLista
7 res3: Int = 42
```

Denna flexibilitet är grunden för **enhetlig access**: namnet kan användas enhetligt oavsett om det är en funktion eller en variabel.

- Om parameterlista saknas får man **inte** använda ( ) vid anrop:

```
1 scala> def ingenParameterLista = 42
2
3 scala> ingenParameterLista
4 res4: Int = 42
5
6 scala> ingenParameterLista()
7 <console>:13: error: Int does not take parameters
```

# Funktioner med defaultargument

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

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

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

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

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

- Om argumentet utelämnas och det finns ett defaultargumentet, så är det defaultargumentet som appliceras.



# Funktioner med namngivna argument

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

```
1 scala> def namn(förnamn: String,  
2               efternamn: String,  
3               förnamnFörst: Boolean = true,  
4               ledtext: String = ""): String =  
5     if (förnamnFörst) s"$ledtext: $förnamn $efternamn"  
6     else s"$ledtext: $efternamn, $förnamn"  
7  
8 scala> namn(ledtext = "Name", efternamn = "Coder", förnamn = "Kim")  
9 res0: String = Name: Kim Coder
```

# Anropsstacken och objektheapen

Minnet är uppdelat i två delar:

- **Anropsstacken:** På stackminnet läggs en **aktiveringspost** (eng. *stack frame*<sup>1</sup>, *activation record*) för varje funktionsanrop med plats för parametrar och lokala variabler. Aktiveringsposten raderas när returvärdet har levererats. Stacken växer vid nästlade funktionsanrop, då en funktion i sin tur anropar en annan funktion.
- **Objektheapen:** I heapminnet<sup>2,3</sup> sparas alla objekt (data) som allokeras under körning. Heapen städas vid tillfälle av skräpsamlaren (eng. *garbage collector*), och minne som inte används längre frigörs.  
[stackoverflow.com/questions/1565388/increase-heap-size-in-java](https://stackoverflow.com/questions/1565388/increase-heap-size-in-java)

---

<sup>1</sup>[en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)

<sup>2</sup>[en.wikipedia.org/wiki/Memory\\_management](https://en.wikipedia.org/wiki/Memory_management)

<sup>3</sup>Ej att förväxlas med datastrukturen heap [sv.wikipedia.org/wiki/Heap](https://sv.wikipedia.org/wiki/Heap)

# Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
1 scala> :paste
2 def f(): Unit = { val n = 5; g(n, 2 * n) }
3 def g(a: Int, b: Int): Unit = { val x = 1; h(x + 1, a + b) }
4 def h(x: Int, y: Int): Unit = { val z = x + y; println(z) }
5
6 scala> f()
```

# Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
1 scala> :paste
2 def f(): Unit = { val n = 5; g(n, 2 * n) }
3 def g(a: Int, b: Int): Unit = { val x = 1; h(x + 1, a + b) }
4 def h(x: Int, y: Int): Unit = { val z = x + y; println(z) }
5
6 scala> f()
```

## Stacken

variabel	värde	Vilken aktiveringspost?
----------	-------	-------------------------

# Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
1 scala> :paste
2 def f(): Unit = { val n = 5; g(n, 2 * n) }
3 def g(a: Int, b: Int): Unit = { val x = 1; h(x + 1, a + b) }
4 def h(x: Int, y: Int): Unit = { val z = x + y; println(z) }
5
6 scala> f()
```

## Stacken

variabel	värde	Vilken aktiveringspost?
n	5	f

# Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
1 scala> :paste
2 def f(): Unit = { val n = 5; g(n, 2 * n) }
3 def g(a: Int, b: Int): Unit = { val x = 1; h(x + 1, a + b) }
4 def h(x: Int, y: Int): Unit = { val z = x + y; println(z) }
5
6 scala> f()
```

## Stacken

variabel	värde	Vilken aktiveringspost?
n	5	f
a	5	g
b	10	
x	1	

# Aktiveringspost

Nästlade anrop ger växande anropsstack.

```
1 scala> :paste
2 def f(): Unit = { val n = 5; g(n, 2 * n) }
3 def g(a: Int, b: Int): Unit = { val x = 1; h(x + 1, a + b) }
4 def h(x: Int, y: Int): Unit = { val z = x + y; println(z) }
5
6 scala> f()
```

## Stacken

variabel	värde	Vilken aktiveringspost?
n	5	f
a	5	g
b	10	
x	1	
x	2	h
y	15	
z	17	

# Lokala funktioner

Med lokala funktioner kan delproblem lösas med nästlade abstraktioner.

```
def gissaTalet(max: Int): Unit = {  
  def gissat = io.StdIn.readLine(s"Gissa talet mellan [1, $max]: ").toInt  
  val hemlis = (math.random * max + 1).toInt  
  def skrivLedtrådOmEjRätt(gissning: Int): Unit =  
    if (gissning > hemlis) println(s"$gissning är för stort :(")  
    else if (gissning < hemlis) println(s"$gissning är för litet :(")  
  def inteRätt(gissning: Int): Boolean = {  
    skrivLedtrådOmEjRätt(gissning)  
    gissning != hemlis  
  }  
  def loop: Int = { var i = 1; while(inteRätt(gissat)){ i += 1 }; i }  
  
  println(s"Du hittade talet $hemlis på $loop gissningar :)")  
}
```

Lokala, nästlade funktionsdeklarationer är tyvärr inte tillåtna i Java.<sup>4</sup>

---

<sup>4</sup>[stackoverflow.com/questions/5388584/does-java-support-inner-local-sub-methods](https://stackoverflow.com/questions/5388584/does-java-support-inner-local-sub-methods)



# Värdeanrop och namnanrop

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

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

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

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

# Klammerparenteser vid ensam parameter

Så här har vi sett nyss att man man göra:

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

Men...

För alla funktioner  $f$  gäller att:

det är helt ok att byta ut vanliga parenteser:

$f(\text{uttryck})$

mot krullparenteser:

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

**om** parameterlistan har **exakt en** parameter.

Men man kan alltså göra så här också:

```
scala> loop{ 21 + 21 }

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

# Uppdelad parameterlista

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

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

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

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

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

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

- Detta kallas även **multipla parameterlistor** (eng. *multiple parameter lists*)

# Skapa din egen kontrollstruktur

- Genom att **kombinera uppdelad parameterlista** med **namnanrop** med **klammerparentes vid ensam parameter** kan vi skapa vår egen kontrollstruktur:

```
scala> def upprepa(n: Int)(block: => Unit) = {  
    var i = 0  
    while (i < n) { block; i += 1 }  
}
```

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

gurka gurka gurka tomat tomat gurka gurka gurka gurka t

# Funktioner är äkta värden i Scala

- En funktioner är ett äkta värde.
- Vi kan till exempel tilldela en variabel ett funktionsvärde.
- Med hjälp av blank+understreck efter funktionsnamnet får vi funktionen som ett **värde** (inga argument appliceras än):

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

scala> val f = add _

scala> f
f: (Int, Int) => Int = <function2>

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

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

# Funktionsvärden kan vara argument

- En funktion kan ha en annan funktion som parameter:

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

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

```
1 scala> tvåGånger(42, öka)
2 res1: Int = 43
```

# Applicera funktioner på element i samlingar med map

```
1 scala> def öka(x: Int) = x + 1
2
3 scala> def minska(x: Int) = x - 1
4
5 scala> val xs = Vector(1, 2, 3)
6
7 scala> xs.map(öka)
8 res0: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
9
10 scala> xs.map(minska)
11 res1: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
12
13 scala> xs map öka
14 res2: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)
15
16 scala> xs map minska
17 res3: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
```

Funktioner som tar andra funktioner som parametrar kallas

**högre ordningens funktioner.**

# Anonyma funktioner

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

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

---

<sup>5</sup>Även kallat "lambda-värde" eller bara "lambda" efter den s.k. lambdakalkylen. [en.wikipedia.org/wiki/Anonymous\\_function](http://en.wikipedia.org/wiki/Anonymous_function)



# Anonyma funktioner

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

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

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

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

---

<sup>5</sup>Även kallat "lambda-värde" eller bara "lambda" efter den s.k. lambdakalkylen. [en.wikipedia.org/wiki/Anonymous\\_function](http://en.wikipedia.org/wiki/Anonymous_function)

# Applicera anonyma funktioner på element i samlingar

- Anonym funktion skapad med funktionsliteral direkt i anropet:

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

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

```
1 scala> xs.map(x => x - 1)
2 res1: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
3
4 scala> xs map (x => x - 1)
5 res2: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2)
```

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

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

# Platshållarsyntax för anonyma funktioner

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

```
_ + 1
```

Ovan expanderar 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 understreckt representerar används **endast en gång**.

```
_ + 1
```

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

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

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

```
_ + _
```

... expanderas till:

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

# Exempel på platshållarsyntax med samlingsmetoden reduceLeft

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

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

# Stegade funktioner, "Curry-funktioner"

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

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

scala> val öka = add(1) _
öka: Int => Int = <function1>

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

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

# Översikt begrepp vi gått igenom hittills

- överlagring
- utelämna tom parameterlista (enhetlig access)
- defaultargument
- namngivna argument
- lokala funktioner
- namnanrop (fördröjd evaluering)
- klammerparentes vid ensam parameter
- uppdelad parameterlista
- egendefinierade kontrollstrukturer
- funktioner som äkta värden
- anonyma funktioner
- stegade funktioner ("Curry-funktioner")

# Begränsningar i Java

- Av alla dessa funktionskoncept...
  - överlagring
  - utelämnna tom parameterlista (principen om enhetlig access)
  - defaultargument
  - namngivna argument
  - lokala funktioner
  - namnanrop (fördröjd evaluering)
  - klammerparentes vid ensam paramenter
  - uppdelad parameterlista
  - egendefinierade kontrollstrukturer
  - funktioner som äkta värden
  - anonyma funktioner
  - stegade funktioner ("Curry-funktioner")
- ...kan man endast göra **överlagring** i Java 7,
- medan även **anonyma funktioner** ("lambda") går att göra (med vissa begränsningar) i Java 8.  
[en.wikipedia.org/wiki/Anonymous\\_function#Java\\_Limitations](http://en.wikipedia.org/wiki/Anonymous_function#Java_Limitations)
- En av de saker jag saknar mest i Java: **lokala funktioner!**

Det är **kombinationen** av alla koncept som **skapar uttryckskraften** i Scala.



# Objekt

# Objekt som modul

- Ett **object** användas ofta för att samla **medlemmar** (eng. *members*) som **hör ihop** och ge dem en egen **namnrymd** (eng. *name space*).
- Medlemmarna kan vara t.ex.:
  - **val**
  - **var**
  - **def**
- Ett sådant objekt kallas även för **modul**.<sup>6</sup>

---

<sup>6</sup>Även paket som skapas med **package** har en egen namnrymd och är därmed också en slags modul. Objekt kan alltså i Scala användas som ett alternativ till paket; en skillnad är att objekt kan ha tillstånd och att objekt inte skapar underkataloger vid kompilering (det finns iofs s.k. **package object**) [en.wikipedia.org/wiki/Modular\\_programming](http://en.wikipedia.org/wiki/Modular_programming)

# Singelobjekt och metod

Ett Scala-**object** är ett s.k. **singelobjekt** (eng. *singleton object*) och finns bara i **en** enda upplaga.

Minne för objektets variabler allokeras första gången objektet refereras.

En funktion som finns i ett objekt kallas en **metod** (eng. *method*).

```
object mittBankkonto {  
  val kontonr: Long      = 1234567L  
  var saldo: Int         = 1000  
  def ärSkuldsatt: Boolean = saldo < 0  
}
```

```
scala> mittBankkonto.saldo -= 25000
```

```
scala> mittBankkonto.ärSkuldsatt  
res0: Boolean = true
```

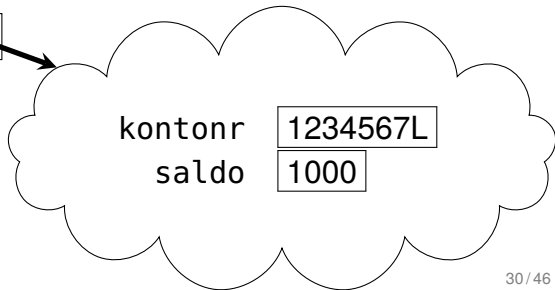
(Vi ska i nästa vecka se hur man med s.k. klasser kan skapa många upplagor av samma typ av objekt, så att vi kan ha flera olika bankkonto.)

# Vad är ett tillstånd?

Ett objekts **tillstånd** är den samlade uppsättningen av värden av alla de variabler som finns i objektet.

```
object mittBankkonto {  
  val kontonr: Long      = 1234567L  
  var saldo: Int         = 1000  
  def ärSkuldsatt: Boolean = saldo < 0  
}
```

mittBankkonto



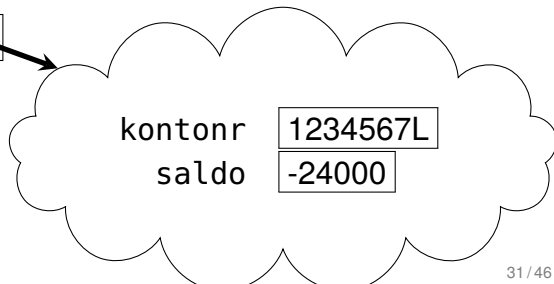
# Tillståndsändring

När en variabel tilldelas ett nytt värde sker en **tillståndsändring**. Ett **förändringsbart objekt** (eng. *mutable object*) har ett **förändringsbart tillstånd** (eng. *mutable state*).

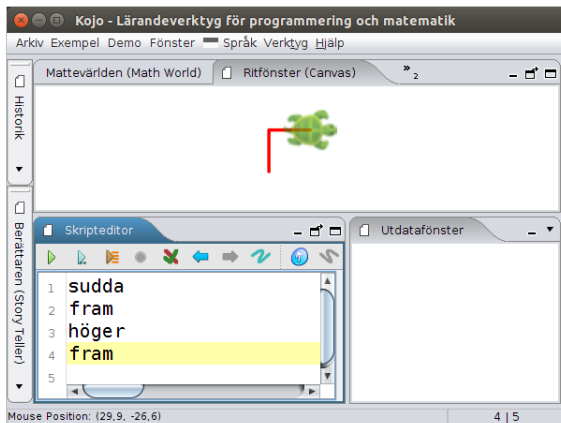
```
scala> mittBankkonto.saldo -= 25000
```

```
scala> mittBankkonto.saldo  
res1: Int = -24000
```

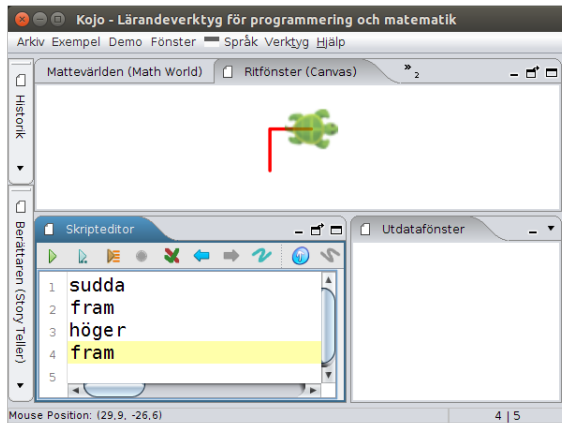
mittBankkonto



# Vad rymmer sköldpaddan i Kojo i sitt tillstånd?



# Vad rymmer sköldpaddan i Kojo i sitt tillstånd?



position, riktning, pennfärg, pennbredd, penna uppe/nere, fyllfärg

# Lata variabler och fördröjd evaluering

Med nyckelordet **lazy** före **val** skapas en s.k. "lat" (eng. *lazy*) variabel.

```
1 scala> val striktVektor = Vector.fill(1000000)(math.random)
2 striktVektor: scala.collection.immutable.Vector[Double] =
3   Vector(0.7583305221813246, 0.9016192590993339, 0.770022134260162, 0.156677181
4
5 scala> lazy val latVektor = Vector.fill(1000000)(math.random)
6 latVektor: scala.collection.immutable.Vector[Double] = <lazy>
7
8 scala> latVektor
9 res0: scala.collection.immutable.Vector[Double] =
10   Vector(0.5391685014341797, 0.14759775960530275, 0.722606095900537, 0.9025572
```

En **lazy val** initialiseras **inte** vid deklarationen utan när den **refereras första gången**. Yttrycket som anges i deklarationen evalueras med s.k. **fördröjd evaluering** (även "lat" evaluering).



# Vad är egentligen skillnaden mellan `val`, `var`, `def` och `lazy val`?

```
object slump {  
  val förAlltidSammaReferens = math.random  
  var kanÄndrasMedTilldelning = math.random  
  def evaluerasVidVarjeAnrop = math.random  
  lazy val fördröjdInit      = Vector.fill(1000000)(math.random)  
}
```

# Vad är egentligen skillnaden mellan `val`, `var`, `def` och `lazy val`?

```
object slump {  
  val förAlltidSammaReferens = math.random  
  var kanÄndrasMedTilldelning = math.random  
  def evaluerasVidVarjeAnrop = math.random  
  lazy val fördröjdInit      = Vector.fill(1000000)(math.random)  
}
```

Lat evaluering är en viktig princip inom funktionsprogrammering som möjliggör effektiva, oföränderliga datastrukturer där element allokeras först när de behövs.  
[en.wikipedia.org/wiki/Lazy\\_evaluation](http://en.wikipedia.org/wiki/Lazy_evaluation)

# Funktioner är objekt

# Programmeringsparadigm

[en.wikipedia.org/wiki/Programming\\_paradigm](http://en.wikipedia.org/wiki/Programming_paradigm):

- **Imperativ programmering**: programmet är uppbyggt av sekvenser av olika satser som läser och **ändrar** tillstånd
- **Objektorienterad programmering**: en sorts imperativ programmering där programmet består av objekt som kapslar in tillstånd och erbjuder operationer som läser och **ändrar** tillstånd.
- **Funktionsprogrammering**: programmet är uppbyggt av samverkande (matematiska) funktioner som **undviker** föränderlig data och tillståndssändringar. Oföränderliga datastrukturer skapar effektiva program i kombination med lat evaluering och rekursion.

# Funktioner är äkta objekt i Scala

Scala visar hur man kan **förena** (eng. *unify*)  
**objekt-orientering** och **funktionsprogrammering**:

**En funktion är ett objekt av funktionstyp  
som har en `apply`-metod.**

# Funktioner är äkta objekt i Scala

Scala visar hur man kan **förena** (eng. *unify*)  
**objekt-orientering** och **funktionsprogrammering**:

**En funktion är ett objekt av funktionstyp  
som har en `apply`-metod.**

```
scala> object öka extends (Int => Int) {  
    def apply(x: Int) = x + 1  
}
```

```
scala> öka(1)  
res0: Int = 2
```

```
scala> öka.    // tryck TAB  
andThen  apply  compose  toString
```

Mer om **extends** senare i kursen...

# Rekursion

# Rekursiva funktioner

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

```
scala> def fakultet(n: Int): Int =  
        if (n < 2) 1 else n * fakultet(n - 1)  
  
scala> fakultet(5)  
res0: Int = 120
```

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



# Loopa med rekursion

```
def gissaTalet(max: Int): Unit = {  
  def gissat = io.StdIn.readLine(s"Gissa talet mellan [1, $max]: ").toInt  
  val hemlis = (math.random * max + 1).toInt  
  def skrivLedtrådOmEjRätt(gissning: Int): Unit =  
    if (gissning > hemlis) println(s"$gissning är för stort :(")  
    else if (gissning < hemlis) println(s"$gissning är för litet :(")  
  def ärRätt(gissning: Int): Boolean = {  
    skrivLedtrådOmEjRätt(gissning)  
    gissning == hemlis  
  }  
  def loop(n: Int = 1): Int = if (ärRätt(gissat)) n else loop(n + 1)  
  
  println(s"Du hittade talet $hemlis på ${loop()} gissningar :)")  
}
```

# Rekursiva datastrukturer

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

# SimpleWindow

# Färdiga, enkla funktioner för att rita finns i klassen `cslib.window.SimpleWindow`

På labben ska du använda `cslib.window.SimpleWindow`

- Paketet `cslib` innehåller paketet `window` som innehåller Java-klassen `SimpleWindow`.
- Med `SimpleWindow` kan man skapa ritfönster.
- Ladda ner <http://cs.lth.se/pgk/cslib> och lägg sedan jar-filen den katalog där du startar REPL med:  
`scala -cp cslib.jar`

# Färdiga, enkla funktioner för att rita finns i klassen `cslib.window.SimpleWindow`

På labben ska du använda `cslib.window.SimpleWindow`

- Paketet `cslib` innehåller paketet `window` som innehåller Java-klassen `SimpleWindow`.
- Med `SimpleWindow` kan man skapa ritfönster.
- Ladda ner <http://cs.lth.se/pgk/cslib> och lägg sedan jar-filen den katalog där du startar REPL med:  
`scala -cp cslib.jar`

```
$ scala -cp cslib.jar  
scala> val w = new SimpleWindow(200,200,"hejsan")
```

# Färdiga, enkla funktioner för att rita finns i klassen `cslib.window.SimpleWindow`

På labben ska du använda `cslib.window.SimpleWindow`

- Paketet `cslib` innehåller paketet `window` som innehåller Java-klassen `SimpleWindow`.
- Med `SimpleWindow` kan man skapa ritfönster.
- Ladda ner <http://cs.lth.se/pgk/cslib> och lägg sedan jar-filen den katalog där du startar REPL med:  
`scala -cp cslib.jar`

```
$ scala -cp cslib.jar  
scala> val w = new SimpleWindow(200,200,"hejsan")
```

Studera dokumentationen för `cslib.window.SimpleWindow` här: <http://cs.lth.se/pgk/api/>

# Veckans övning och laboration

# Övning functions

- Kunna skapa och använda funktioner med en eller flera parametrar, default-argument, namngivna argument, och uppdelad parameterlista.
- Kunna använda funktioner som äkta värden.
- Kunna skapa och använda anonyma funktioner (s.k. lambda-funktioner).
- Kunna applicera en funktion på element i en samling.
- Förstå skillnader och likheter mellan en funktion och en procedur.
- Förstå skillnader och likheter mellan en värde-anrop och namnanrop.
- Kunna skapa en procedur i form av en enkel kontrollstruktur med fördröjd evaluering av ett block.
- Kunna skapa och använda objekt som moduler.
- Förstå skillnaden mellan äkta funktioner och funktioner med sidoeffekter.
- Kunna skapa och använda variabler med fördröjd initialisering och förstå när de är användbara.
- Kunna förklara hur nästlade funktionsanrop fungerar med hjälp av begreppet aktiveringspost.
- Kunna skapa och använda lokala funktioner, samt förstå nyttan med lokala funktioner.
- Känna till att funktioner är objekt med en `apply`-metod.
- Känna till stegade funktioner och kunna använda partiellt applicerade argument.
- Känna till rekursion och kunna förklara hur rekursiva funktioner fungerar.



# Lab blockmole

- Kunna kompilera Scalaprogram med `scalac`.
- Kunna köra Scalaprogram med `scala`.
- Kunna definiera och anropa funktioner.
- Kunna använda och förstå default-argument.
- Kunna ange argument med parameternamn.
- Kunna definiera objekt med medlemmar.
- Förstå kvalificerade namn och import.
- Förstå synlighet och skuggning.