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

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 deklareras 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*[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 heapminnet[2,3] 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

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

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 = scala.io.StdIn.readLine(s"Gissa tal mellan [1, $max]: ").toInt
  val hemlis = (math.random * övreGräns + 1).toInt
  def skrivLedtrådOmEjRätt(gissning: Int): Unit =
    if (gissning > hemlis) println("För stort :(")
    else if (gissning < hemlis) println("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.[4]

[4] 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 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) 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(uttryck)`
mot krullparenteser: `f{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*)

docs.scala-lang.org/style/declarations.html#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:

```
scala> def räkna(x: Int, y: Int, f: (Int, Int) => Int) = f(x , y)

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

scala> def sub(a: Int, b: Int) = a - b

scala> räkna(21, 21, add _)
res1: Int = 42

scala> räkna(21, 21, sub _)
res1: Int = 0
```

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

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

Anonyma funktioner

- Man behöver inte sätta namn på funktioner. De kan i stället skapas med hjälp av **funktionsliteraler**.[5]
- En funktionsliteral har inget namn, men ...
  1. en parameterlista,
  2. sedan den reserverade krummeluren **=>**
  3. och sedan ett uttryck (eller ett block).
- Exempel:

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

[5] Även kallade ”lambda-värden” efter den s.k. lambdakalkylen.

Applicera funktioner på element i samlingar

Stegade funktioner, ”Curry-funktioner”

Begränsningar i Java

- Av dessa funktionskoncept...
  - överlagring
  - utelämna 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** men inget annat i Java 7,
- medan även **anonyma funktioner** (”lambda”) går att göra (med vissa begränsningar) i Java 8.
- 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** som hör ihop och ge dem en egen **namnrymd**.
- Medlemmarna kan vara t.ex.:
  - **`val`**
  - **`var`**
  - **`def`**
- Ett sådant objekt kallas även för **modul**.[6]

[6] 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

Vad är ett tillstånd?

Lata variabler och fördröjd evaluering

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

En funktion som finns inuti ett objekt är en **metod**.

Funktioner är objekt

Programmeringsparadigm

Funktioner är äkta objekt i Scala

Scala visar hur man kan **förena** (eng. *unify*) objekt-orientering och funktionsprogrammering på ett elegant & pragmatiskt sätt:

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

Rekursion

Rekursiva funktioner

Rekursiva datastrukturer