# 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

- 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
```

- Men om parameterlista saknas får man **inte** använda `()` vid anrop:

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

# Funktioner med default-argument

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

```
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 default-argumentet, så är det default-argumentet som appliceras på parametern.

# Stackminne och Heapminne

Minnet är uppdelat i två delar:

- **Stackminne**: På stacken 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.
- **Heapminne**: I heapminnet[2,3] sparas alla objekt (data) som allokeras under körning. Heapminnet 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 stackminne.

```
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 stackminne.

```
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 stackminne.

```
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 stackminne.

```
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()
```

**Stacken**

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

# Lokala funktioner

# Värdeanrop och namnanrop

# Uppdelad parameterlista

# Skapa din egen kontrollstruktur

# Funktioner med namngivna argument

# Funktioner är äkta värden i Scala

- En funktioner är ett äkta värde; vi kan till exempel tilldela en variabel ett funktionsvärde:
- Funktioner har en typ precis som alla värden:

# Anonyma funktioner

# Applicera funktioner på element i samlingar

# Stegade funktioner, ”Curry-funktioner”

# 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**.[4]

[4] 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.
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