Dominik Gruntz
Daniel Kröni

Arbeitsblatt: Scala Funktionen

Scala vereint die objektorientierte Programmierung mit der funktionalen Programmierung. In diesem Arbeitsblatt lernen Sie Scala von der funktionalen Seite kennen.

Funktion = Methode als Objekt
In Java definiert man mit folgender Syntax eine Methode:

```
public void call() {
  System.out.println(":-(");
}
```

Das einzige was man damit machen kann, ist diese aufzurufen '`call();`'. Das kennen Sie bereits.

In Scala definiert man das so:

```
def call(): Unit = println(":-)")
```

Diese Methode kann man ebenfalls aufrufen '`call()`'. Aber zusätzlich kann man sie wie ein Objekt verwenden. Das heisst man kann Methodenobjekte als Argument einer anderen Methode übergeben, in Variablen oder gar Datenstrukturen speichern oder von anderen Methoden als Resultat zurückgegeben lassen! Eine Methode die als Objekt verwendet wird, nennt man in Scala "Funktion".

Betrachten wir nun die Methode `async`, die Ihr Argument in einem neuen Thread ausführt:

```
def async(action: () => Unit): Unit = {
  new Thread() {
    override def run(): Unit = {
      action() // Hier wird die übergebene Funktion aufgerufen
    }
  }.start()
}
```

Das ganze Konstrukt kann nun so aufgerufen werden:

```
async(call) // call wird als Funktion der Methode async als Argument übergeben
```

Der Parameter action hat als Typ eine *Funktion* '`() => Unit`', das erkennen Sie am Pfeil '`=>`'. In diesem Beispiel ist es der Typ der Funktionen, die keine Argumente nehmen (die leere Argumentliste '`()`') und als Resultat '`Unit`' (void) zurückgeben.

Natürlich kann man auch interessantere Funktionstypen deklarieren!

```
def everySecond(action: Int => Unit): Unit
```

Der action Parameter muss nun eine Funktion sein, die als Argument ein `Int` nimmt und wiederum `Unit` zurückgibt.

Aufgabe 1

a) Definieren Sie die Methode `clock`, die als Argument ein Int i nimmt und damit einen "Tick(i)" String auf der Konsole ausgibt:

```
clock(3) // gibt "Tick(3)" auf der Konsole aus.
```

b) Implementieren Sie die Methode `everySecond`. Sie soll in einem neuen Thread die übergebene action jede Sekunde mit einem aufsteigenden Zähler aufrufen. Testen Sie Ihre Implementierung indem Sie `everySecond` mit `clock` als Argument aufrufen.

Ab jetzt macht programmieren wieder Spass!

Exkurs: Anonyme Funktionen

Funktionen muss man nur einen Namen geben, damit man sie referenzieren kann. Wenn eine Funktion aber nur einmal gebraucht wird und übersichtlich genug ist, kann gleich eine anonyme Funktion (ohne Namen, auch Lambda Expression genannt) verwendet werden:

```
everySecond(i => println("Tick(" + i + ")"))
```

Wenn der Rumpf der anonymen Funktion mehr als einen Ausdruck beinhaltet, müssen Sie diesen mit geschwungenen Klammern zusammenhalten:

```
everySecond(i => {
  println("Tock()")
  println("Tick(" + i + ")")
})
```

Aufgabe 2

a) Implementieren Sie die Methode `time` zur Messung der Ausführungszeit ihres Arguments. Die Methode `time` kann so verwendet werden:

```
def time(block: () => Unit): Unit = { ... }

time(() => {
   Thread.sleep(100)
})                  // Ausgabe: [101ms]
```

b) [*optional*] Implementieren Sie eine generische Version der `time` Funktion, die das Resultat der zu messenden Funktion zurückgibt:

```
def time[A](block: () => A): A = { ... }

val a = time(() => {
  Thread.sleep(100)
  "Done"
})
```

Exkurs: By-name Parameter

Die Aufrufe der Funktion `time` sind bezüglich Ästhetik nicht einwandfrei.

Viel schöner wäre es, wenn `time` so aufgerufen werden könnte:

```
time {
  Thread.sleep(100)
}                   // Ausgabe: [101ms]
```

Kein Problem in Scala. Statt dem Funktionstypen '`() => Unit`' können Sie einfach '`=> Unit`' definieren was soviel bedeutet wie "eine Funktion *ohne* Parameterliste" (statt einer Funktion mit einer *leeren* Parameterliste).

```
def time(block: => Unit): Unit = { ... }
```

"!alacS tim ssapS leiV".reverse

Aufgabe 3 (optional)

Der Umgang mit Atomic Variables in Java ist umständlich. Sie definieren eine solche Variable und schreiben dann jedes Mal den Code um mit einem optimistischen Loop das CAS zu probieren bis es klappt:

```
private final AtomicInteger balance = new AtomicInteger(0);

public void deposit(int amount) {
  while (true) {
    int currentValue = balance.get();
    int newValue = currentValue + amount;
    if (balance.compareAndSet(currentValue, newValue)) {
      return;
    }
  }
}
```

a) Definieren Sie die Scala Klasse `NiceAtomicInt`, die innerhalb einen `j.u.c.a.AtomicInteger` verwendet und sich von so verwenden lässt:

```
val balance = new NiceAtomicInt(0)
balance.modify(b => b + 10)
```

Die optimistische Schleife wird nur noch einmal programmiert und zwar in der Klasse `NiceAtomicInt` und nicht bei jeder Verwendung.

b) Definieren Sie die nun die Klasse `NiceAtomic[A]`, die nicht mehr auf `Int` eingeschränkt ist, sondern über einen Typparameter `A` einen beliebigen Inhaltstyp annehmen kann.
Hinweis: Sie wollen dazu eine `j.u.c.a.AtomicReference` verwenden.