Aktuelle Kapitel Sozialer Web-Technologien – Einführung in Scala

Übung

Prof. Dr.-Ing. Hendrik Gärtner

Scala Allgemein

- Von 2001 bis 2004 von Martin Odersky entwickelt
- wird am Labor für Programmiermethoden an der École polytechnique fédérale de Lausanne entwickelte
- Martin Odersky: Working hard to keep it simple: http://www.youtube.com/watch?v=3jg1AheF4n0 , O'Reilley OSCON, Java 2011, Keynode Vortrag
- Scala steht für „Scalable Language“:
  - kompakt gehaltener Sprachkern
  - bietet die Möglichkeit häufig verwendete Sprachelemente z. B. Operatoren oder zusätzliche Kontrollstrukturen in Benutzerklassen zu implementieren und dadurch den Sprachumfang zu erweitern
  - Geeignet für die Erstellung eigener DSLs (Domain Specific Languages)
- Ist insbesondere für die Entwicklung skalierender, paralleler Anwendungen geeignet
- Mittlerweile in der Industrie verbreitet: Twitter, Amazon, IBM, LinkedIn Novell, TomTom, etc.

Scala - Eigenschaften

- Scala läuft auf der Java Virtual Machine, existierende Deployments können verwendet werden
- Mit Scala können alle Java-Bibliotheken verwendet werden
- Scala unterstützt sowohl objektorientierte als auch funktionale Programmierung
- Scala ist wie Java statisch getypt
- Die Angabe der Typen ist meist optional und wird über Inferenz-Mechanismen ermittelt
- Scala unterscheidet zwischen änderbaren (mutable) und nicht änderbaren (immutable) Variablen

Scala - Einstieg

- Alles in Scala sind Objekte
- Es existieren keine primitiven Typen und statische Methoden
- Hauptklasse wird durch ein sogenanntes Singleton-Objekt realisiert
- Von Singleton-Objekten existiert immer nur eine Instanz
- Wird beim erstmaligen Gebrauch erzeugt

```
object ErsteAnwendung {
  def main(args: Array[String]){
        println("Hello World")}
}
```

Oder: Beinhaltet schon die Main-Methode

```
object ErsteAnwendung extends App{
        println("Hello World")
}
```

Übung

Entwicklung einer Klasse für das Rechnen mit Rationalen Zahlen:

- Erzeugen eines Bruchs durch Angabe von Nenner und Zähler
- Überprüfung der Gültigkeit des Bruchs
- Umwandlung eines Bruchs in eine reelle Zahl
- Addieren des Bruchs
- Negieren des Bruchs
- Subtraktion des Bruchs
- Kürzen des Bruchs
- Subtrahieren des Bruchs
- Infix-Schreibweise und Definition von Operatoren für Rationale Zahlen
- Vergleich von Brüchen (kleiner, max)

Testen der Klasse!

Scala Konstruktoren

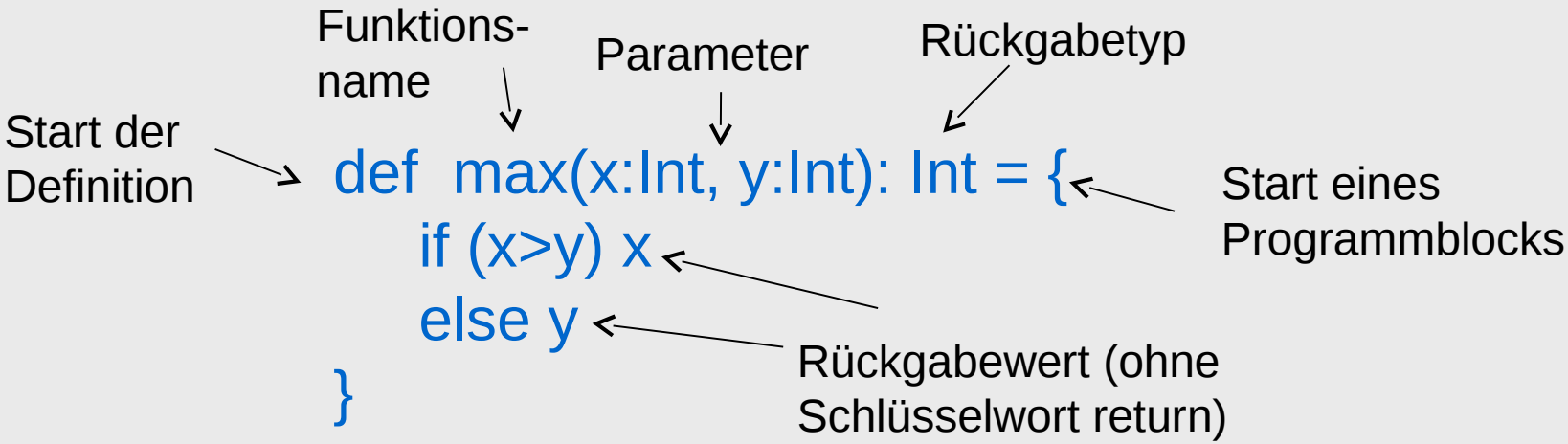
Scala :
```
class Rational (numerator:Int, denominator:Int){
def this (denom:Int) = this(1,denom)
override def toString:String = numerator + “/“ + denominator}
```

Java :
```
public class Rational {
  private int numerator;
  private int denominator;

  public Rational(int n, int d){
    denominator= d;
    numerator=n;}
  public Rational(int d){
    this(1,d);}

  @Override
  public String toString(){ return denominator + "/" + numerator;}
}
```

Scala Funktionsdefinition



- Rückgabetyp kann meist weggelassen werden (ergibt sich aus dem Kontext)
- Beim Überschreiben von Funktionen muss override angegeben werden
- Semikolon zwischen Befehlen, kann – muss aber nicht – gesetzt werden

Basisfunktionen der Klasse Rational

```
class Rational (numerator:Int, denominator:Int){

  require (denominator!=0,“denominator must be != 0“)
                                  // wirft IllegalArgumentException
  println("A Rational was created....")  // ist Teil des Konstruktors
  def this (denom:Int) = this(1,denom)
  def num:Int=numerator  // damit numerator von außen zugänglich ist
  def denom:Int=denominator // damit denominator von außen zugänglich ist
  def value:Double = (num.toDouble / denom) // Konvertierung
  override def toString = num + "/" + denom
}
```

Test-driven-Development

- Für jede Funktion werden ***vor*** der Implementierung Tests geschrieben
- Tests werden bei jeder Änderung automatisiert ausgeführt werden

Einsatz des Testframeworks FunSuite:

- Einbindung der Bibliotheken
  - JUnit4
  - scalatest-2.9… (downloadbar unter: http://www.scalatest.org)
- Import der Bibliotheken
  - import org.scalatest.FunSuite
  - import org.junit.runner.RunWith
  - import org.scalatest.junit.JUnitRunner
- Start von JUnit mit der Annotation
  @RunWith(classOf[JUnitRunner])

Beispieltestprogramm

```
class TestRational extends FunSuite{
    test("Rational Inititalisation:") {
    val x = new Rational(1,2)
    assert(x.toString === "1/2")
  }
    test("Test Add: 1/2 + 1/6 = 2/3") {
    val x = new Rational(1,2)
    val y = x.add(new Rational(1,6))
    expect(“2/3") {y.toString}
  }
test("Test requirement (denominator!=0)"){
      intercept [IllegalArgumentException] {
        new Rational(1,0)}
}
…
}
```

Entwerfen Sie Tests für die Funktionen sub, neg und mul!

Require/Assertions

- Schlägt ein Test mit require, expect oder assert fehl, so wird eine Exception geworfen:
  - Bei require eine IllegalArgumentException,
  - bei expect und assert ein AssertionError

- Spiegelt die unterschiedlichen Intentionen wider:
  - Require wird benutzt um sogen. Preconditions abzuprüfen
  - Assert und expect um die Funktionsfähigkeit einer Funktion zu testen

Implementierung der Methode add

- Addition der Brüche durch Angleichung der Nenner
- Danach addieren der Zähler

$$\frac{a}{b} + \frac{c}{d} = \frac{a * d + c * b}{b * d}$$

```
def add(other:Rational): Rational= {
    new Rational(num * other.denom + denom * other.num,
    denom * other.denom)
}
```

Implementieren Sie eine Funktion neg (negieren), eine Funktion mul (Multiplikation) und eine Funktion sub (subtrahieren) !

Berechnen Sie: 2/3 – 2/4 -1/9 und 2/3*2/4

Funktionen neg und sub

Funktion zum Negieren des Bruchs:

```
def neg:Rational= new Rational(-num,denom)
```

Funktion für die Subtraktion:

```
def sub(other:Rational):Rational = add(other.neg)
```

Funktion für die Multiplikation:

```
def mul(other:Rational):Rational = new
    Rational(num*other.num, denom * other.denom)
```

Ergebnisse: 2/3 – 2/4 -1/9 = 33/108 und 2/3 * 2/4=4/12

Wie kann der Bruch gekürzt werden?

Kürzen des Bruchs

Euklidscher Algorithmus:

```
def gcd(a:Int,b:Int):Int= {    // greatest common divisor
    if (b==0) a
    else gcd(b, a % b)}
```

Beispiel: gcd(1060, 855) = 5

1060 = 1 * 855 + 205
855 = 4 * 205 + 35
205 = 5 * 35 + 30
35 = 1*30 + 5
30 = 6***5** + 0

Integrieren Sie die Funktion gcd in die Klasse Rational!

Integration des Kürzens (1/2)

Variante 1:

```
class Rational (numerator:Int, denominator:Int){
    private def gcd(a:Int,b:Int):Int= if (b==0) a else gcd(b, a % b)
    private val g = gcd(numerator, denominator)
    def num:Int=numerator/g
    def denom:Int = denominator/g
    …}
```

Variante 2:

```
class Rational (numerator:Int, denominator:Int){
    private def gcd(a:Int,b:Int):Int= if (b==0) a else gcd(b, a % b)
    def num:Int=numerator/gcd(numerator,denominator)
    def denom:Int = denominator/gcd(numerator,denominator)
     ….}
```

Integration des Kürzens (2/2)

Variante 3:

```
class Rational (numerator:Int, denominator:Int){
    private def gcd(a:Int,b:Int):Int= if (b==0) a else gcd(b, a % b)
    private val g = gcd(numerator, denominator)
    val num:Int=numerator/g  // ist damit ein public member
    val denom:Int = denominator/g // ist damit ein public member
    …}
```

- Für die Klienten der Klasse Rational ist es unerheblich, welche Variante implementiert wurde. Die Klasse verhält sich immer gleich
- Die Möglichkeit unterschiedliche Implementierungen zu verwenden wird das **„Prinzip der Datenabstraktion“** genannt.
- Datenabstraktion ist eines der wichtigsten Punkte des Softwareengineerings

Definition von Operatoren

Im Prinzip sind die Rationalen Zahlen als Integers implementiert. Es gibt jedoch einen signifikanten Unterschied zwischen Integers und Rationals:

- Zwei Integer-Zahlen x und y werden mit x+y addiert
- Zwei Brüche x und y werden mit y.add(y) addiert

In Scala kann dieser Unterschied eliminiert werden!

Infix-Operator

Jede Methode mit einem Parameter kann als *Infix-Operator* verwendet werden, d.h. er kann in **Infix-Notation**[1] geschrieben werden:

r add s	≈	r.add(s)
r less s	≈	r.less(s)
r max s	≈	r.max(s)

…….

[1]Infix-Notation: Operator steht zwischen den beiden Operanden

Identifier in Scala

Identifier sind in Scala entweder:

- Alphanumerisch (bestehen aus Alphanumerischen Zeichenketten),
- Symbolic (bestehen aus einem oder mehreren Operatorsymbolen (wie +,-,etc.)) oder
- Bestehen aus einer Alphanumerischen Zeichenkette, einem _ (zählt als Alphanumerisch), gefolgt von einem oder mehreren Symbolen

Beispiele:

x1 * +?%& vector_++ counter_=

Also 4 + 5 ist nicht anderes als ein Int-Objekt mit dem Wert 4 und der Methode + die aufgerufen wird mit dem Parameter 5

$$4 + 5 \approx 4.+(5)$$

Operatoren in Rationalen Zahlen

Austausch der Identifier: bspw.

```
def  add(other:Rational):Rational= new Rational(…)   in
def  +(other:Rational):Rational= new Rational(…)
```

oder

```
def neg:Rational= new Rational(-num,denom)   in
def unary_- :Rational= new Rational(-num,denom)
```

Dann können Rationale Zahlen wie Integers behandeln werden.

Vorrang der Operatoren

Die Ausführungsreihenfolge der Operatoren wird durch das erste Zeichen bestimmt: (aufsteigender Reihenfolge)

(alle Buchstaben)

|

^

&

< >

= !

:

+ -

* / %

(all other special characters)

Hausaufgabe

Schreiben Sie eine Funktion, die die 10001. Primzahl findet.

Aufteilung des Problems in zwei Funktionen:

1. Primzahlentest
2. Zählen der gefundenen Primzahlen

Benutzen Sie dazu nur Funktionen und keine Operatoren aus der imperativen Programmierung - also kein var!

Vielen Dank für

Ihre Aufmerksamkeit

Prof. Dr.-Ing. Hendrik Gärtner