Universität der Bundeswehr
Fakultät für Informatik – Institut 2
Prof. Dr. Wolfgang Hommel
Michael Grabatin, Michael Steinke

WT 2020
Übungsblatt 1

Einführung in die Informatik II

17. und 20.01.2019

Teil I

Wiederholung aus Einführung 1

1 Scala

- Scala ist eine funktionale Programmiersprache:

  Das Bedeutet, dass jede Funktion ein Wert ist, wodurch Funktionen zum Beispiel auch als Parameter an andere Funktionen übergeben werden können. Funktionen, die Funktionen als Parameter engegennehmen werden als Funktionen höherer Ordung (higher order functions) bezeichnet. Zudem erlaubt Scala anonyme Funktionen (z. B. `(x: Int) => x + 1`), das Verschachteln von Funktionsdefinitionen (z. B. die Definition einer Hilfsfunktion in einer anderen Funktion) und Currying.

  Durch Currying können Funktionen partiell angewendet werden, dabei werden mehrere Parameterblöcke verwendet. Danach kann die Funktion partiell angewendet werden indem nur der erste Parameterblock spezifiziert wird.

- Scala ist eine objektorientierte Programmiersprache:

  Der objektorientierte Teil von Scala ist für „Einführung in die Informatik 1“ und „Einführung in die Informatik 2“ nicht weiter relevant. An einigen Stellen müssen allerdings objektorientierte Teile von Scala verwendet werden ohne genauer auf ihre Funktionsweise einzugehen. Objektorientierung wird in der Vorlesung „Objektorientierte Programmierung“ näher betrachtet.

- Scala hat ein statisches Typsystem:

  Eine statische Typisierung bedeutet, dass Scala schon beim Kompilieren festlegt, welchen Typ ein Wert oder eine Variable hat. Der Typ kann dazu entweder im Code angegeben werde (z. B. `: Int`) oder in vielen Fällen vom Typinferenzsystem automatisch bestimmt werden.

- Scala läuft in der Java Virtual Machine (JVM):

  Scala-Code wird, wie auch Java-Code, zum Ausführen nicht direkt zu Maschienencode kompiliert, sondern in einen allgemeinen Bytecode, der dann von der JVM interpretiert wird. Im Gegensatz dazu gibt es Sprachen wie zum Beispiel C, die zum Ausführen direkt in Maschienencode kompiliert werden oder Sprachen wie Perl und Python, die in der Regel immer von einem Interpreter zur Laufzeit interpretiert werden.

  Da Scala und Java zur Ausführung die JVM verwenden, können von Scala aus auch Java-Code oder Java-Bibliotheken verwendet werden. Umgekehrt ist dies in vielen Fällen ebenfalls möglich.

2 Syntax

Um Code leichter lesbar zu machen, existieren für die meisten Sprachen Konventionen, die vorschreiben, wie unter anderem Datei-, Funktions- und Variablennamen zu wählen, wie Programmblöcke zu formatieren sind und

wie die Dokumentation eines Programmes geschrieben werden soll. Scala definiert einen offiziellen Style Guide `http://docs.scala-lang.org/style`, der möglichst eingehalten werden sollte. Im Folgenden sind die wichtigsten Punkte zusammengefasst.

2.1 Basis Syntax und Konventionen

- Case-Sensitive: Scala unterscheidet Groß- und Kleinschreibung. `val test` und `val Test` sind unterschiedliche Werte.
- Alle **Funktionsnamen** sollten mit einem kleinen Kleinbuchstaben beginnen und wenn nötig CamelCasing verwenden. Zum Beispiel `def einfacheTestFunktion`.
- **Klassen- und Objektnamen** sollten mit einem Großbuchstaben beginnen und wenn nötig CamelCasing verwenden.
- Der **Dateiname** einer Scala-Datei sollte identisch zu dem in der Datei spezifizierten Objekt oder Klasse sein. Die Datei, die das `object HelloWorld` enthält, sollte also `HelloWorld.scala` heißen.
- Der Startpunkt für jedes Scala-Programm ist die `main`-Methode, sie muss innerhalb einer Klasse oder eines Objektes definiert werden.

  ```
  1 def main(args: Array[String])
  ```

  Der Parameter `args` enthält die beim Aufruf des Programms übergebenen Parameter.

2.2 Namen

Um in Scala Objekte zu identifizieren, müssen Namen vergeben werden.

2.2.1 Alphanumerische Namen

Alphanumerische Namen können in Scala entweder mit einem Buchstaben oder einem Unterstrich beginnen. Darauf können weitere Buchstaben, Zahlen oder Unterstriche folgen.

2.2.2 Operator Namen

Operatoren können aus beliebigen ASCII-Zeichen bestehen. Zum Beispiel +,:,<,?

2.2.3 Keywords

Scala definiert einige Keywords, die nicht in Objektnamen verwendet werden können. In Tabelle 1 sind diese angegeben.

2.3 Kommentare

Scala hat die gleichen Kommentare wie Java. Einzelne Zeilen können durch `//` auskommentiert werden, ganze Blöcke durch `/* ... */`.

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	null
object	override	package	private
protected	return	sealed	super
this	throw	trait	try
true	type	val	var
while	with	yield	
-	:	=	=>
<-	<:	<%	>:
#	@		

Tabelle 1: Durch Scala reservierte Keywords

2.4 Pakete

Scala-Code kann zur besseren Übersichtlichkeit und Wiederverwendbarkeit in Packages zusammengefasst werden. Der Name eines Packages wird über die erste Codezeile in einer Datei festgelegt zum Beispiel:

```
1 package test.test1.HelloWorld
```

Um ein Paket zu verwenden, muss es importiert werden. Dazu wird das `import`-Statement verwendet, dass an beliebiger Stelle, jedoch vor der ersten Verwendung des importierten Codes, stehen kann.

```
1 import test.test1.HelloWorld
```

3 Werte & Variablen

Werte und Variablen können verwendet werden, um Daten im Speicher des Programmes abzulegen. Der Unterschied zwischen Werten und Variablen besteht darin, dass Werte unveränderlich sind, Variablen aber während der Programmausführung modifiziert werden können.

3.1 Definition

Werte werden allgemein durch `val <Name> [: <Typ>] = <Wert>` definiert.

```
1 val x : Int = 1
2 val y = 2
3 y = 2 //Error: reassignment to val
```

Variablen werden allgemein durch `var <Name> [: <Typ>] = <Wert>` definiert.

```
1 var x : Int = 1
2 var y = 2
3 y = 3 //Ok
```

3.2 Datentypen

In Scala stehen einige Datentypen von vornherein zur Verfügung, einige davon sind in Tabelle 2 aufgeführt. Im Gegensatz zu Java, das zwischen primitiven Typen und Typ-Objekten unterscheidet, gibt es in Scala nur Typ-Objekte. Diese sind, wie in Abbildung 1 aus `http://docs.scala-lang.org/tutorials/tour/unified-types.html` dargestellt, hierarchisch aufgebaut.

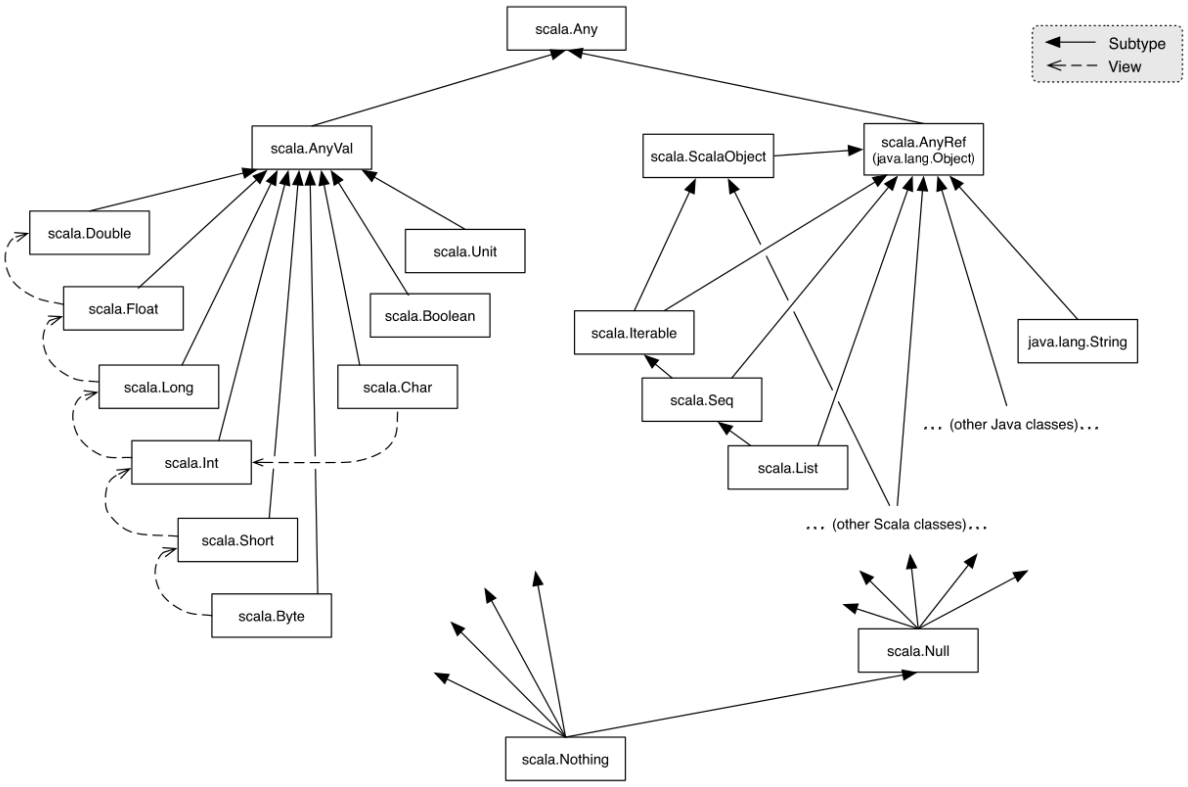

Abbildung 1: Scala Typ-Hierarchie

Datentyp	Speicherung	Wertebereich
Boolean		`true` oder `false`
Byte	8bit, signed (vorzeichenbehaftet), Zweierkomplement	-2^7 bis $2^7 - 1$
Short	16bit, signed (vorzeichenbehaftet), Zweierkomplement	-2^15 bis $2^{15} - 1$
Int	32bit, signed (vorzeichenbehaftet), Zweierkomplement	-2^31 bis $2^{31} - 1$
Long	64bit, signed (vorzeichenbehaftet), Zweierkomplement	-2^63 bis $2^{63} - 1$
Float	32bit, IEEE 754 single precision	`Float.MinValue` bis `Float.MaxValue`
Double	64bit, IEEE 754 double precision	`Double.MinValue` bis `Double.MaxValue`
Char	16bit, unsigned (ohne Vorzeichen)	0 bis $2^{16} - 1$
String	Eine unveränderliche Reihung von Chars	
Unit	Entspricht keinem Wert	
Null	Leere Referenz, Untertyp jedes anderen Referenztyps	
Nothing	Untertyp jedes anderen Typs	
Any	Supertyp jedes anderen Typs	
AnyRef	Supertyp jedes anderen Referenztypes	

Tabelle 2: Auswahl an Scala-Datentypen

4 Bedingungen `if` & `else`

Durch `if` kann der Programmfluss entsprechend angegebener Bedinungen gesteuert werden. Eine Bedingung kann jeder Ausdruck sein, der sich zu `true` oder `false` auswerten lässt.

Die einfachste Form einer Bedingung ist ein einzelnes `if`. Nach der Ausführung des folgenden Blocks hat die Variable `x` mindestens den Wert 10.

```
1 if(x < 10) {
2     x = 10
3 }
```

Über einen `else`-Block kann eine `if`-Anweisung erweitert werden. Der `else`-Teil wird ausgeführt, wenn die Bedingung am Anfang des `if`-Blocks nicht zutrifft. Das folgende Beispiel überprüft ob `x` kleiner ist als 10 und setzt dann `x` auf 10, oder setzt `x` auf 100, falls `x` größer ist als 10.

```
1 if(x < 10) {
2     x = 10
3 } else {
4     x = 100
5 }
```

Eine weitere Möglichkeit ist es mehrere `if`-Bedingungen durch ein `else if` aneinander zu hängen. Ein solcher Block kann beliebig viele `else if`-Teile besitzen, allerdings darf immer nur ein reiner `if`- und ein reiner `else`-Zweig in einem Block vorkommen.

```
1 if(x < 10) {
2     x = 10
3 } else if (x == 10) {
4     x = 50
5 } else {
6     x = 100
7 }
```

Sollten bei einem `if`, `else if`-Konstrukt mehrere Bedingungen zutreffen, wird immer nur die erste zutreffende ausgeführt.

Im Allgmeinen hat ein `if`-Block die folgende Form. Eckige Klammern gehören dabei nicht zur Scala-Syntax, sondern zeigen an, dass der durch sie umschlossene Teil optional ist. Ein Stern hinter einer geschlossenen eckigen Klammer zeigt an, dass der vorhergehende Block beliebig oft wiederholt werden kann.

```
if(<Bedingung>){
    <Anweisung>
}
[
else if(<Bedingung>) {
    <Anweisung>
}
]*
[
else {
    <Anweisung>
}
]
```

5 Pattern Matching

Eine Alternative zu `if` ist das Pattern Matching

```
1 if(x == 1) "eins"
2 else if(x == 2) "zwei"
```

```
else if(x == 3) "drei"
else "mehr"

x match {
    case 1 => "eins"
    case 2 => "zwei"
    case 3 => "drei"
    case _ => "mehr"
}
```

Pattern Matching ist dabei in einigen Fällen übersichtlicher. Das ist besonders offensichtlich bei der Bearbeitung von Listen, wo durch eine rekursive Funktion und Pattern Matching sich ein übersichtlicher Programmfluss ergibt.

```
def sum(l: List[Int]): Int = l match {
  case x::xs => x + sum(xs)
  case _ => 0
}
```

Der Ausdruck `x::xs` wird in dem obigen Pattern Matching verwendet, um das erste Element auf den Wert `x` zu matchen und die Restliste auf `xs` zu matchen. (Die Restliste kann dabei eventuell Leer sein.) Der Default-Case wird als Basisfall der Rekursion verwendet, um die Rekursion zu beenden, falls der erste Case nicht mehr gematched werden kann (also kein Element zum abtrennen mehr vorhanden ist).

Teil II

Übung

6 Basisoperationen

Hinweis: Diese Aufgabe enthält im Wesentlichen nur leicht modifizierte Beispiele aus der Vorlesung. Für diese Aufgaben wird auch **keine Musterlösung** zur Verfügung gestellt.

a) Welche Ausgabe erzeugt der folgende Code-Ausschnitt? Geben Sie die erste und die letzte Ziffer der ausgegebenen Folge an.

```
1  var x = 10
2
3  do {
4      println(x)
5      x -= 1
6  } while(x > 0)
```

b) Welche Ausgabe erzeugt der folgende Code-Ausschnitt? Wenn Sie sich nicht sicher sind, probieren Sie den Code in einem Scala-Worksheet oder einer Scala-REPL selber aus.

```
1  var x = 100
2
3  def incCounter : Int = {
4      x += 1
5      return x
6  }
7
8  def a(u: => Int) = {
9      for(i <- 1 to 10) {
10         println(u)
11     }
12 }
13
14 def b(u: Int) = {
15     for(i <- 1 to 10) {
16         println(u)
17     }
18 }
19
20 a(incCounter)
21 b(incCounter)
22 println(x)
```

c) Vervollständigen Sie die folgenden, mit ____ (mehrere Underscores) markierten, teilweise implementierten Funktionen:

```
1  // Legen Sie ein Array mit 10 Einträgen vom Wert 1 an
2  var a = Array.fill(_____)(_____)
3
4  // Diese Funktion soll testen, ob das Array die Länge 0 hat oder
       nicht
5  def hasLengthZero(a: _________________): Boolean = {
6      if(_________________) {
```

```
        return true
    } else {
        return false
    }
}

// Diese Funktion soll alle Elemente des Arrays ausgeben
def printArrayElements(a: Array[Int]): _____ = {
    for(i <- _____________) {
        println(a(i))
    }
}

// Diese Funktion soll ein Array zurückgeben, welches alle Einträge
   des
// übergebenen Arrays in umgekehrter Reihenfolge enthält.
def reverseArray(a: Array[Int]): Array[Int] = {
    var newArray = ___________________________________
    for(i <- _____________) {

    }
    return _______________
}
```

d) Erstellen Sie eine Funktion, die ein zweidimensionales Array zurück gibt. Diese Funktion soll als Eingabe ein eindimensionales Array an Werten und einem Parameter für die maximale Zeilenlänge des Arrays haben.

```
// Array der Länge 18 mit zufällig gewählten Einträgen
var test : Array[Int] = Array.tabulate____________________
// Ergebnis> test: Array[Int] = Array(51, 76, 0, 28, 59, 33, 77, 54,
   10, 33, 33, 8, 1, 32, 6, 96, 82, 63)

def createMatrix(a: Array[Int], maxLength: Int): Array[_______________
   ] = {
    // Anzahl der benötigten Zeilen berechnen
    var rows : Int = Math.ceil(a.length.toDouble / maxLength).toInt

    // Zweidimensionales Array der Größe rows * maxLength anlegen
    // Alle Zellen sollen mit 0 initialisiert werden
    var erg : Array[Array[Int]] = ____________________________

    for(i <- 0 until rows) {
        for(j <- 0 until maxLength) {
            if(i*maxLength + j < a.length) {
                erg____________ = _______
            }
        }
    }
    return erg
}

createMatrix(test, 5)
// Ergebnis> res: Array[Array[Int]] = Array(Array(51, 76, 0, 28, 59),
    Array(33, 77, 54, 10, 33), Array(33, 8, 1, 32, 6), Array(96, 82,
   63, 0, 0))
```

7 Schleifen

Gegeben sei eine ganze Zahl $n >= 5$ und eine wie folgt deklarierte und initialisierte Reihung A:

```
1 var A = Array.range(0, n)
```

Die folgenden vier Programmstücke $(P1)$ - $(P4)$ arbeiten auf A.

Programmstück (P1):

```
1  for (i <- 0 to (k - 1) / 2) {
2    val h = A(i)
3    A(i) = A(k - 1 - i)
4    A(k - 1 - i) = h
5  }
6
7  for (i <- 0 to (n - 1 - k) / 2) {
8    val h = A(k + i)
9    A(k + i) = A(n - 1 - i)
10   A(n - 1 - i) = h
11 }
12
13 for (i <- 0 to (n - 1) / 2) {
14   val h = A(i)
15   A(i) = A(n - 1 - i)
16   A(n - 1 - i) = h
17 }
```

Programmstück (P2):

```
1  for (i <- 0 to (k - 1)) {
2    val h = A(i)
3    A(i) = A(k - 1 - i)
4    A(k - 1 - i) = h
5  }
6
7  for (i <- 0 to (n - 1 - k)) {
8    val h = A(k + i)
9    A(k + i) = A(n - 1 - i)
10   A(n - 1 - i) = h
11 }
12
13 for (i <- 0 to (n - 1)) {
14   val h = A(i)
15   A(i) = A(n - 1 - i)
16   A(n - 1 - i) = h
17 }
```

Programmstück (P3):

```
1    for (i <- 0 to k - 1) {
2      val h = A(0)
3      for (j <- 0 to n - 2) {
4        A(j) = A(j + 1)
5      }
6      A(n - 1) = h
7    }
```

Programmstück (P4):

```
1  for (i <- 0 to n - 1) {
2    val h = A(0)
3    for (j <- 0 to n - 2) {
4      A(j) = A(j + 1)
5    }
6    A(n - 1) = h
7  }
```

a) Führen Sie die Programmstücke für geeignete Testdaten ($n = 5$ und $k = 3$) von Hand aus.

b) Je zwei der Programmstücke liefern das gleiche Ergebnis. Welche?

c) Erläutern Sie, warum das (auch für andere n und k) so ist, indem Sie erklären, was die Programmstücke und die in ihnen enthaltenen Schleifen bewirken.