ANHANG

D Anmerkungen zur Java-Syntax

D.1 Java-Datentypen

In Java gibt es zwei Arten von Datentypen: primitive Datentypen und Objekttypen. Primitive Datentypen werden direkt in Variablen gespeichert und weisen eine Wertesemantik auf (Werte werden kopiert, wenn sie einer anderen Variablen zugewiesen werden). Objekttypen werden gespeichert, indem Referenzen auf das Objekt (und nicht das Objekt selbst) gespeichert werden. Wird ein solcher Datentyp einer anderen Variablen zugewiesen, wird nur die Referenz und nicht das Objekt kopiert.

D.1.1 Primitive Datentypen

Die folgende Tabelle listet alle primitiven Datentypen der Programmiersprache Java auf:

Tabelle D.1
Die primitiven Datentypen in Java.

Typbezeichnung	Beschreibung	Beispiel-Literale		
Integer (ganze Zahlen)				
`byte`	Integer in Bytegröße (8 Bit)	24	-2	
`short`	Kleine Integer (16 Bit)	137	-119	
`int`	Integer (32 Bit)	5409	-2003	
`long`	Größere Integer (64 Bit)	423266353L	55L	
Reelle Zahlen				
`float`	Gleitkommazahl einfacher Genauigkeit	43.889F		
`double`	Gleitkommazahl doppelter Genauigkeit	45.632.4e5	0.4	
Andere Datentypen				
`char`	Ein einzelnes Zeichen (16 Bit)	'm'	'?'	'\u00F6'
`boolean`	Ein boolescher Wert (`true` oder `false`)	`true`	`false`	

Hinweise:

- Eine Zahl ohne Dezimalpunkt wird im Allgemeinen als **int** interpretiert, doch bei einer Zuweisung (sofern der Wert passt) automatisch in **byte**, **short** oder **long** umgewandelt. Du kannst ein Literal als **long** deklarieren, indem du an die Zahl ein **L** anhängst (das kleingeschriebene **L** (**l**) kann ebenfalls verwendet werden, sollte jedoch wegen der Verwechslungsgefahr mit der Eins vermieden werden).
- Eine Zahl mit Dezimalpunkt ist vom Typ **double**. Du kannst das Literal als **float** kennzeichnen, indem du ein **F** oder **f** an die Zahl anhängst.
- Ein Zeichen kann als einzelnes Unicodezeichen in einfachen Anführungsstrichen oder als vierstelliger Unicode-Wert mit vorangestelltem **\u** angegeben werden.
- Die beiden booleschen Literale lauten **true** und **false**.

Da die Variablen der primitiven Datentypen nicht auf Objekte verweisen, gibt es auch keine Methoden, die mit ihnen verknüpft sind. Wenn sie jedoch in einem Kontext verwendet werden, der einen Objekttyp erfordert, kann durch Autoboxing ein primitiver Wert in ein entsprechendes Objekt umgewandelt werden.

Die folgende Tabelle zeigt eine Übersicht über die kleinsten und größten Werte der numerischen Datentypen.

Tabelle D.2 Minima und Maxima der numerischen Datentypen in Java.

Typ	Minimum	Maximum
byte	-128	127
short	-32768	32767
int	-2147483648	2147483647
long	-9223372036854775808	9223372036854775807
	Positives Minimum	**Positives Maximum**
float	1.4e–45	3.4028235e38
double	4.9e–324	1.7976931348623157e308

D.1.2 Objekttypen

Alle Typen, die nicht im Abschnitt *Primitive Datentypen* aufgeführt wurden, sind Objekttypen. Dazu gehören Klassen- und Schnittstellentypen aus der Java-Standardbibliothek (wie **String**) und benutzerdefinierte Typen.

Eine Variable eines Objekttyps hält eine Referenz (oder „Zeiger") auf ein Objekt. Zuweisungen und die Übergabe von Parametern weisen Referenzsemantik auf (d.h., die Referenz und nicht das Objekt wird kopiert). Nachdem eine Variable einer anderen zugewiesen wurde, weisen beide Variablen auf dasselbe Objekt. Man spricht davon, dass die zwei Variablen Aliase für dasselbe Objekt sind.

Klassen sind die Schablonen für Objekte, die die Zustandsfelder und Methoden definieren, über die hinterher jedes Objekt verfügt.

Felder verhalten sich wie Objekttypen – sie besitzen ebenfalls Referenzsemantik.

D.2 Java-Operatoren

D.2.1 Arithmetische Ausdrücke

In Java gibt es eine ganze Reihe von Operatoren für die arithmetischen und logischen Ausdrücke.

[]	.	++	--	(Parameter)	
++	--	+	-	!	~
new	(Typanpassung)				
*	/	%			
+	-				
<<	>>	>>>			
<	>	>=	<=	instanceof	
==	!=				
&					
^					
|					
&&					
||					
?:					
=	+=	-=	*=	/=	%=	>>=	<<=	>>>=	&=	|=	^=

Tabelle D.3
Java-Operatoren (Operatoren mit höchster Priorität stehen ganz oben).

Tabelle D.3 listet alle Operatoren auf, einschließlich der Typanpassung (Cast) und der Parameterübergabe. Die meisten Operatoren sind entweder binäre Operatoren (mit linkem und rechtem Operanden) oder unäre Operatoren (mit nur einem Operanden). Die wichtigsten arithmetischen Binäroperationen sind

- `+` Addition
- `-` Subtraktion
- `*` Multiplikation
- `/` Division
- `%` Modulo (oder Rest-nach-Division)

Die Ergebnisse von Divisionen oder Modulo-Operationen hängen davon ab, ob es sich bei den Operanden um Integer oder Gleitkommazahlen handelt. Bei zwei Integerwerten liefert die Division ein Integerergebnis zurück und verwirft jeglichen Rest, bei zwei Gleitkommzahlen ist das Ergebnis eine Gleitkommazahl:

`5 / 3` Ergebnis: **`1`**

`5.0 / 3` Ergebnis: **`1.6666666666666667`**

(Es fällt auf, dass nur einer der Operanden eine Gleitkommazahl sein muss, um ein Gleitkommaergebnis zu produzieren.)

Wenn in einem Ausdruck mehr als ein Operator verwendet wird, dann greifen die *Prioritätsregeln*, um die Reihenfolge der Auswertung festzulegen. In Tabelle D.3 stehen die Operatoren mit der höchsten Priorität ganz oben, sodass zum Beispiel Multiplikation, Division und Modulus Vorrang vor Addition und Subtraktion haben. Das bedeutet, dass die folgenden zwei Beispiele beide als Ergebnis 100 haben:

```
51 * 3 - 53
154 - 2 * 27
```

Binäre Operatoren mit der gleichen Prioritätsstufe werden von links nach rechts ausgewertet und unäre Operatoren mit der gleichen Prioritätsstufe von rechts nach links.

Wenn es erforderlich ist, die normale Auswertungsreihenfolge zu ändern, können Klammern verwendet werden. So haben die beiden folgenden Beispiele als Ergebnis 100:

```
(205 - 5)  / 2
2 * (47 + 3)
```

Die wichtigsten unären Operatoren sind -, **!**, **++**, **--**, **[]** und **new**. Dir ist sicher aufgefallen, dass sowohl **++** als auch **--** in den beiden obersten Reihen der Tabelle D.3 zu finden sind. Die Operatoren in der obersten Reihe weisen ihren einzigen Operanden zur Linken auf, und die in der Reihe darunter zur Rechten.

D.2.2 Boolesche Ausdrücke

In booleschen Ausdrücken werden Operatoren verwendet, um mittels der Kombination von Operanden einen Wert zu produzieren, der entweder wahr oder falsch ist. Solche Ausdrücke sind normalerweise in den Testausdrücken von *if-Anweisungen* und Schleifen zu finden.

Die relationalen Operatoren kombinieren in der Regel ein Paar arithmetischer Operanden, die Tests auf Gleichheit und Ungleichheit können aber auch mit Objektreferenzen durchgeführt werden. Zu den relationalen Operatoren in Java gehören:

==	Gleich	!=	Nicht gleich
<	Kleiner als	<=	Kleiner gleich
>	Größer als	>=	Größer gleich

Die binären logischen Operatoren kombinieren zwei boolesche Ausdrücke, um als Ergebnis einen anderen booleschen Wert zu produzieren. Die Operatoren lauten

&&	Und
||	Oder
^	Exklusives Oder

Außerdem übernimmt

! Nicht

einen einzelnen booleschen Ausdruck und ändert ihn von **`true`** in **`false`** und umgekehrt.

Sowohl **&&** als auch **||** sind etwas ungewöhnlich in ihrer Anwendung. Wenn der linke Operand von **&&** falsch ist, dann spielt der Wert des rechten Operanden keine Rolle mehr und wird nicht ausgewertet. Ähnlich verhält es sich mit dem **||**-Operator. Wenn hierbei der linke Operand wahr ist, wird der rechte Operand nicht mehr ausgewertet. Diese beiden Operatoren werden deshalb auch als Kurzschlussoperatoren bezeichnet.

D.3 Java-Kontrollstrukturen

Kontrollstrukturen nehmen Einfluss auf die Reihenfolge, in der Anweisungen im Rumpf einer Methode oder eines Konstruktors ausgeführt werden. Sie lassen sich in zwei wesentliche Kategorien zusammenfassen: *Auswahlanweisungen* und *Schleifen*.

Eine Auswahlanweisung verfügt über einen Entscheidungspunkt, an dem entschieden wird, welcher Weg durch den Rumpf einer Methode oder eines Konstruktors eingeschlagen werden soll. Eine **`if-else`**-Anweisung ist zum Beispiel mit einer Entscheidung zwischen zwei verschiedenen Anweisungsblöcken verbunden, während eine **`switch`**-Anweisung die Auswahl einer bestimmten Option aus mehreren Optionen erlaubt.

Schleifen bieten die Möglichkeit, Anweisungen entweder eine bestimmte Anzahl von Malen oder unendlich oft zu wiederholen. Für Ersteres bieten sich die **`for-each`**- und die **`for`**-Schleifen an und für Letzteres die **`while`**- und **`do`**-Schleifen.

Für die Praxis solltest du dir jedoch merken, dass Ausnahmen zu der obigen Kategorisierung durchaus häufig vorkommen. So kann zum Beispiel eine **`if-else`**-Anweisung verwendet werden, um zwischen mehreren alternativen Anweisungssätzen zu wählen, wenn im **`else`**-Teil eine weitere **`if-else`**-Anweisung verschachtelt ist. Und eine **`for`**-Schleife kann auch dazu verwendet werden, um einen Anweisungsblock beliebig oft zu durchlaufen.

D.3.1 Auswahlanweisungen

if-else

Die **if-else**-Anweisung gibt es in zwei Varianten, die beide durch die Auswertung eines booleschen Ausdrucks gesteuert werden.

```
if (ausdruck)
{
    anweisungen
}

if (ausdruck)
{
    anweisungen
}
else
{
    anweisungen
}
```

In der ersten Variante wird der Wert des booleschen Ausdrucks verwendet, um zu entscheiden, ob die Anweisungen ausgeführt werden oder nicht. In der zweiten wird der Ausdruck verwendet, um zwischen zwei Anweisungsblöcken zu entscheiden, von denen nur einer ausgeführt wird.

Beispiele:

```
if (field.size() == 0)
{
    System.out.println("Das Feld ist leer.");
}

if (number < 0)
{
    reportError();
}
else
{
    processNumber(number);
}
```

Sehr häufig werden **if-else**-Anweisungen miteinander verbunden, indem ein zweites **if-else** in den **else**-Teil der ersten **if-else**-Anweisung platziert wird. Dies kann beliebig oft erfolgen, es empfiehlt sich jedoch, am Ende immer einen **else**-Teil vorzusehen.

```
if (n < 0)
{
    handleNegative();
}
else if (number == 0)
{
    handleZero();
}
else
{
    handlePositive();
}
```

switch

Die **switch**-Anweisung entscheidet anhand eines einzelnen Wertes, zu welcher der beliebig vielen **case**-Marken gesprungen wird. Zwei Möglichkeiten der Anwendung sind:

```
switch (ausdruck)
{
    case wert: anweisungen;
        break;
    case wert: anweisungen;
        break;
    weitere faelle ausgelassen
    default: anweisungen;
        break;
}

switch (ausdruck)
{
    case wert1:
    case wert2:
    case wert3:
        anweisungen;
        break;
    case wert4:
    case wert5:
        anweisungen;
        break;
    weitere faelle ausgelassen
    default:
        anweisungen;
        break;
}
```

Hinweise:

- Eine **switch**-Anweisung kann eine beliebige Anzahl von **case**-Marken aufweisen.
- Der **break**-Befehl nach jeder **case**-Marke darf in der Regel nicht fehlen, da sonst die Ausführung mit den Anweisungen der nächsten **case**-Marke fortgesetzt wird. Man kann dieses Verhalten allerdings auch bewusst nutzen, wie z.B. in der zweiten der obigen Varianten. Dort wird für die ersten drei Werte der erste Anweisungsblock ausgeführt, während die Werte vier und fünf den zweiten Anweisungsblock ausführen.
- Die **default**-Marke definiert den Standardfall und ist optional. Wenn kein Standardfall vorgegeben wird, kann es passieren, dass kein Fall ausgeführt wird.
- Der **break**-Befehl zum Abschluss des **default**-Falls (oder der letzten **case**-Marke, falls kein **default** vorgegeben wird) ist nicht erforderlich, wird aber als guter Stil betrachtet.

Beispiele:

```
switch(day)
{
    case 1: dayString = "Montag";
        break;
    case 2: dayString = "Dienstag";
        break;
    case 3: dayString = "Mittwoch";
        break;
    case 4: dayString = "Donnerstag";
        break;
    case 5: dayString = "Freitag";
        break;
    case 6: dayString = "Samstag";
        break;
    case 7: dayString = "Sonntag";
        break;
    default: dayString = "ungültiger Tag";
        break;
}

switch(month)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        numberOfDays = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        numberOfDays = 30;
        break;
```

```
        case 2:
            if(isLeapYear())
                numberOfDays = 29;
            else
                numberOfDays = 28;
            break;
    }
```

D.3.2 Schleifen

Java kennt drei Schleifen: **while**, **do-while** und **for**. Die **for**-Schleife gibt es in zwei Varianten. Mit Ausnahme der **for-each**-Schleife wird die Anzahl der Wiederholungen durch einen booleschen Ausdruck gesteuert.

while

Die **while**-Schleife führt einen Anweisungsblock so lange aus, wie ein gegebener Ausdruck zu **true** ausgewertet wird. Der Ausdruck wird vor der Ausführung des Schleifenrumpfes geprüft, sodass der Rumpf unter Umständen nullmal (d.h. gar nicht) ausgeführt wird. Diese Eigenschaft ist ein wichtiges Merkmal der **while**-Schleife.

```
while (ausdruck)
{
    anweisungen;
}
```

Beispiele:

```
System.out.print("Bitte einen Dateinamen eingeben: ");
input = readInput();
while (input == null)
{
    System.out.print("Bitte erneut versuchen: ");
    input = readInput();
}

int index = 0;
boolean found = false;
while (!found && index < list.size())
{
    if(list.get(index).equals(item))
    {
        found = true;
    }
    else
    {
    index++;
    }
}
```

do-while

Die **do-while**-Schleife führt einen Anweisungsblock so lange aus, wie ein gegebener Ausdruck zu **true** ausgewertet wird. Der Ausdruck wird nach der Ausfüh-

rung des Schleifenrumpfes getestet, sodass der Rumpf mindestens einmal ausgeführt wird. Dieser Unterschied zur **while**-Schleife ist besonders wichtig.

```
do
{
    anweisungen;
} while (ausdruck);
```

Beispiel:

```
do
{
    System.out.print("Bitte einen Dateinamen eingeben: ");
    input = readInput();
} while (input == null);
```

for

Die **for**-Schleife gibt es in zwei Varianten. Die erste Variante – auch als **for-each**-Schleife bezeichnet – wird ausschließlich dazu verwendet, um die Elemente einer Sammlung zu durchlaufen. Bei jedem Durchlauf der Schleife wird der Schleifenvariablen der Wert des nächsten Elements aus der Sammlung zugewiesen.

```
for (variablendeklaration : sammlung)
{
    anweisungen;
}
```

Beispiel:

```
for (String note : list)
{
    System.out.println(note);
}
```

Die zweite Variante der **for**-Schleife wird so lange ausgeführt, wie die Auswertung der *Bedingung* **true** ergibt. Vor Eintritt in die Schleife wird eine *Initialisierungsanweisung* genau einmal ausgeführt. Die *Bedingung* wird vor jedem Schleifendurchlauf erneut ausgewertet (sodass die Schleife unter Umständen kein einziges Mal ausgeführt wird). Nach jeder Ausführung des Schleifenrumpfes wird eine *Inkrement*-Anweisung ausgeführt.

```
for (initialisierung; bedingung; inkrement)
{
    anweisungen;
}
```

Beispiel:

```
for (int i = 0; i < text.size(); i++)
{
    System.out.println(text.get(i));
}
```

Beide Varianten der **for**-Schleife werden häufig dazu verwendet, um den Rumpf der Schleife eine bestimmte Anzahl von Malen auszuführen – zum Beispiel einmal für jedes Element in einer Sammlung. Eine **for-each**-Schleife kann nicht verwendet werden, wenn die Sammlung verändert werden muss, während sie durchlaufen wird.