	Datentypen		AnPr	V 1.0
	Name	Klasse	Datum	

1 Warum wir Datentypen benötigen

Jede Programmiersprache muss die Daten im Speicher ablegen, der ausschließlich mit Binärdaten arbeiten kann. Die Festlegung des Datentyps ermöglicht es dem Rechner daher:

- Festzustellen, wie viel Speicherplatz pro Variable zu reservieren ist
- Korrekt auf den reservierten Speicherplatz zuzugreifen
- Die Bits entsprechend den Konventionen des Datentyps korrekt zu interpretieren

Wenn wir beispielsweise folgende Bitkombination nehmen:

1	0	1	1	0	0	0	1

ohne den Datentypen festzulegen, ist es nicht möglich die Bits zu interpretieren. Würden wir diese Bitkombination als die niederwertigsten Bits von einem Interger-, Byte- oder Character- Datentyp setzen, würden bei einer Ausgabe folgende Daten angezeigt werden:

Datentyp:	**Ausgabe:**
`int`	177
`byte`	-79
`char`	±

Um nun eine Variable nutzen zu können, müssen wir sie deklarieren, wodurch ein Speicherplatz der richtigen Größe reserviert wird und wir müssen sie initialisieren, wodurch erstmalig („initial“) ein Wert festgelegt wird. Dies wird meist in einer Codezeile erledigt:

```
int iValue = 10;
```

Datentypen werden meist in zwei Gruppen eingeteilt – primitive und zusammengesetzte Datentypen.

1.1 Primitive Datentypen

Hier zur Wiederholung nochmal eine Aufstellung der primitiven Datentypen in Java:

Inhaltstyp	Datentyp	Wertebereich	Speicherbedarf
Wahrheits-wert		**true/false**	
ganze Zahl		**-128 bis 127**	
		-32.768 bis 32.767	
		-2.147.483.648 bis 2.147.483.647	
		–9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	
Gleitkom-mazahl		**+/-1,4*10^{-45} bis +/-3,4*10^{+38} (7 signifikante Stellen)**	
		+/-4,9*10^{-324} bis +/-1,7*10^{+308} (15 signifikante Stellen)	
einzelnes Zeichen		**65.536 unterschiedliche Zeichen**	**16 bit / 2 Byte**

Wichtig für Java ist, dass es für jeden primitiven Datentyp eine zugeordnete Klasse gibt – die sogenannte Wrapperklasse. Diese wird benötigt um:

- Datentyp spezifische Informationen bereitzustellen, wie z.B.
  - maximaler Wert: `Integer.MAX_VALUE`
  - Größenbedarf im Speicher: `Double.SIZE`
- Hilfsfunktionen für den Umgang mit dem Datentyp, wie z.B.
  - Parsen eines Strings in eine Integer Zahl: `Integer.parseInt("12345")`
- Nutzung von Zahlen als Objekte, wenn ausschließlich Objekte akzeptiert werden, wie bspw.
  - `ArrayList<Integer>`

1.2 Zusammengesetzte Datentypen

In Java ist alles, was kein primitiver Datentyp ist, ein zusammengesetzter Datentyp und somit ein Objekt einer Klasse. Dies gilt für die grundlegenden Elemente wie:

- Arrays
- Strings

genauso wie für Klassen, welche wir selbst realisieren. Objekte in Variablen solcher Datentypen haben eine wichtige Eigenschaft, welche primitive Datentypen (in Java) nicht haben. Man kann Eigenschaften und Methoden nutzen, indem man mit einem Punktoperator nach dem Variablennamen auf diese Eigenschaften und Methoden zugreifen kann. Dadurch bieten zusammengesetzte Datentypen einen bequemeren Umgang als primitive Datentypen.

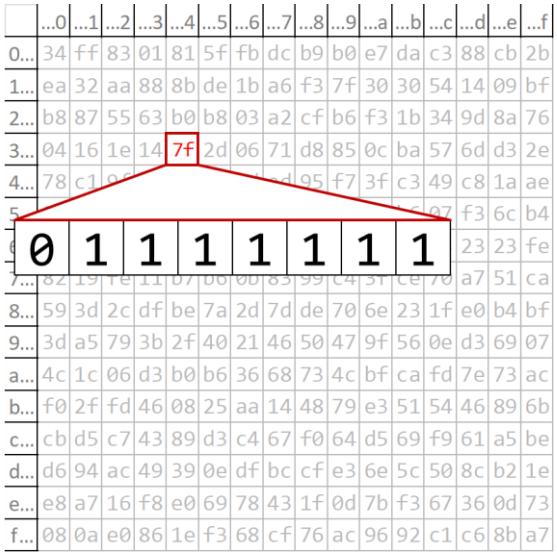
1.3 Konstanten vs. immutable Objects

Konstanten sind Variablen, welche nur initialisiert werden können. Diese werden mit dem Schlüsselwort „final“ deklariert. Nach der erstmaligen Belegung mit einem Wert ist es nicht mehr möglich, diesen Wert zu ändern. Konstanten werden meist ausschließlich in Großbuchstaben benamt. Nun gibt es bei zusammengesetzten Datentypen jedoch noch einen weiteren wichtigen Begriff, der oft mit Konstanten verwechselt wird – immutable. Dies bedeutet, dass ein Objekt, welches einmal erzeugt wurde, nicht mehr verändert werden kann. Während sich das Wort „Konstante“ auf die Zuweisung der Variable bezieht, bezieht sich immutable auf die interne Veränderung des zugewiesenen Objektes. Dadurch ist der Begriff „immutable“ primär für Objekte relevant, nicht für primitive Datentypen, da hier eine „innere“ Veränderung nicht möglich ist (man kann hier lediglich sagen, dass final auch gleichzeitig immutable ist). Ein wichtiger Vertreter von immutable Objekten ist der String. Ein String, der einmal erzeugt wurde, kann ist nicht mehr veränderbar.

2 Beispielhaftes Speichermodell für Datentypen

Um das Verhalten von Datentypen in Java zu verstehen, sehen wir uns vor Augen führen, dass die Informationen im Speicher des Rechners abgelegt werden müssen. Hierzu folgender Code:

```
byte bValue = 127;
```



Dies bewirkt, dass der Rechner:

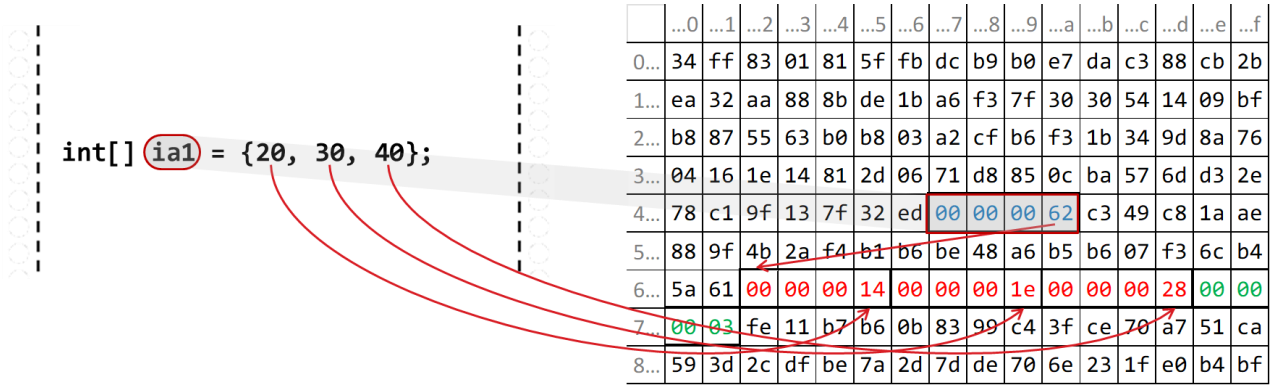
- Eine Speicherzelle von einem Byte reserviert – oder auch „allokiert“ (hier die Beispieladresse 0x34)
- Dort die Bitkombination für die Zahl 127 hinterlegt
- Der Name der Variablen „bValue“ für den Rest der Gültigkeit mit dem Speicherplatz der Adresse 0x34 verbunden wird

Dies ist notwendig, da wir zum Compilezeitpunkt nicht wissen können, wo zur Laufzeit dieser freie Speicherplatz sich befinden wird. Der Variablenname ist also eine Referenz auf einen Speicherplatz, der erst in der Zukunft feststeht.

Alle primitiven Datentypen funktionieren in Java identisch. Der Variablenname zeigt auf den (ersten) Speicherplatz des allokierten Datentyps. Dort steht immer der Wert, welcher dieser Variablen zugewiesen wurde. Wenn wir bspw. zwei Variablen vergleichen, dann wird genau der Wert verglichen, der unter den Adressen dieser Variablen gefunden wird.

Ganz anders wird dies bei zusammengesetzten Datentypen umgesetzt. Hier finden wir in der Speicherzelle, auf die die Variable zeigt, nicht den eigentlichen Wert, sondern wiederum eine Referenz auf den Speicherbereich, in dem sich das eigentliche Objekt befindet.

Hier ein vereinfachtes Beispiel für ein Array mit drei Werten. Die Arrayvariable trägt den Wert 0x62. Dies wiederum ist die Adresse des Bereiches, in dem sich die Hexwerte der drei Dezimalzahlen 20, 30 und 40 befinden. Weiterhin finden wir in diesem Bereich noch andere Informationen über das Array, wie bspw. die Größe:

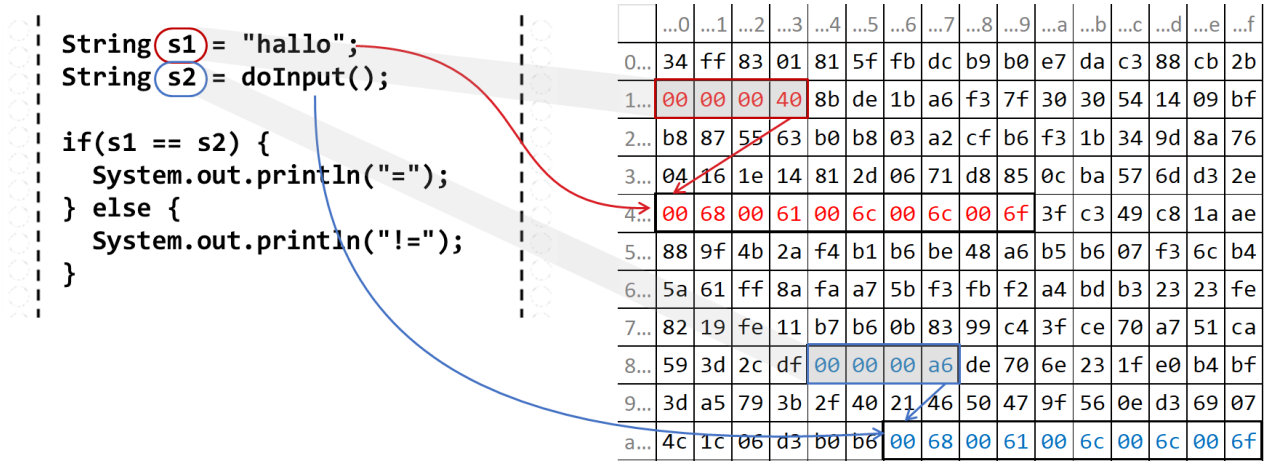


	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...a	...b	...c	...d	...e	...f
0...	34	ff	83	01	81	5f	fb	dc	b9	b0	e7	da	c3	88	cb	2b
1...	ea	32	aa	88	8b	de	1b	a6	f3	7f	30	30	54	14	09	bf
2...	b8	87	55	63	b0	b8	03	a2	cf	b6	f3	1b	34	9d	8a	76
3...	04	16	1e	14	81	2d	06	71	d8	85	0c	ba	57	6d	d3	2e
4...	78	c1	9f	13	7f	32	ed	00	00	00	62	c3	49	c8	1a	ae
5...	88	9f	4b	2a	f4	b1	b6	be	48	a6	b5	b6	07	f3	6c	b4
6...	5a	61	00	00	00	14	00	00	00	1e	00	00	00	28	00	00
7...	00	03	fe	11	b7	b6	0b	83	99	c4	3f	ce	70	a7	51	ca
8...	59	3d	2c	df	be	7a	2d	7d	de	70	6e	23	1f	e0	b4	bf

Somit erklärt sich auch, warum in Java nur Objektvariablen „null“ sein können. Nur hier können wir auf eine nicht existente Adresse zeigen (meist ist das der Wert 0x00). Bei einem primitiven Datentyp (bspw. byte) sind alle Bitkombinationen mit einem tatsächlichen Wert verknüpft. Der Wert 0x00 ist bei byte der Dezimalwert 0.

2.1 Vergleich von Objekten

Dieses Konzept der Referenzvariablen bringt nun ein „Problem“ beim Umgang mit zusammengesetzten Datentypen mit sich. Wenn wir zwei String Variablen, welche den gleichen Inhalt haben aber auf unterschiedlichen Adressen liegen, miteinander vergleichen, kommt trotz inhaltlicher Gleichheit ein „`false`“ beim Vergleich heraus. Gehen wir von folgendem Code und der dazugehörigen vereinfachten Speicherbelegung aus:



	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...a	...b	...c	...d	...e	...f
0...	34	ff	83	01	81	5f	fb	dc	b9	b0	e7	da	c3	88	cb	2b
1...	00	00	00	40	8b	de	1b	a6	f3	7f	30	30	54	14	09	bf
2...	b8	87	55	63	b0	b8	03	a2	cf	b6	f3	1b	34	9d	8a	76
3...	04	16	1e	14	81	2d	06	71	d8	85	0c	ba	57	6d	d3	2e
4...	00	68	00	61	00	6c	00	6c	00	6f	3f	c3	49	c8	1a	ae
5...	88	9f	4b	2a	f4	b1	b6	be	48	a6	b5	b6	07	f3	6c	b4
6...	5a	61	ff	8a	fa	a7	5b	f3	fb	f2	a4	bd	b3	23	23	fe
7...	82	19	fe	11	b7	b6	0b	83	99	c4	3f	ce	70	a7	51	ca
8...	59	3d	2c	df	00	00	00	a6	de	70	6e	23	1f	e0	b4	bf
9...	3d	a5	79	3b	2f	40	21	46	50	47	9f	56	0e	d3	69	07
a...	4c	1c	06	d3	b0	b6	00	68	00	61	00	6c	00	6c	00	6f

Die beiden Variablen s1 und s2 haben den gleichen Inhalt, die Charwerte `'h','a','l','l'` und `'o'`. Wenn wir die beiden vergleichen, so vergleichen wir eben die Werte, auf die die beiden Variablen verweisen, nämlich die Adressen 0x40 und 0xa6. Diese sind unterschiedlich, unabhängig davon, dass die Charwerte identisch sind. Aus diesem Grund müssen wir beim Inhaltlichen Vergleich von Objekten immer auf Vergleichsmethoden wie bspw. „equals()“ zurückgreifen.

2.2 Übergabeverhalten von Objekten

In Java werden Variablen bei Unterprogrammaufruf immer als Wert übergeben („call by value“). Die Art des „Wertes“ bei Variablen sind aber bei primitiven und zusammengesetzten Datentypen unterschiedlich. Bei primitiven Datentypen ist der übergebene Wert immer die Information, mit der wir arbeiten wollen. Bei int ist dies eben die Zahl, mit der wir bswp. rechnen wollen. Bei einer Arrayvariable ist der Wert die Adresse des Arrays, wie wir es weiter oben gesehen haben. Insofern verhält sich die Übergabe einer Objektvariable in Java wie ein „call by reference“.

In C oder C++ können wir die Art des calls explizit festlegen, da wir hier für jede Art von Variablen die Adresse ermitteln können und somit immer selbst entscheiden können, ob das aufgerufene Unterprogramm die Adresse, oder den eigentlichen Wert verarbeiten wollen.

Um das Verhalten verstehen zu können, sehen wir uns hier folgendes Programm an:

```
public static void main(String[] args) {
  int iVal = 20;
  addValue(iVal);
  System.out.println("#2:" + iVal);

  int[] iaVal = {20, 30, 40};
  addValue(iaVal);
  System.out.println("#4:" + iaVal[0]);
}

public static void addValue(int iVal) {
  iVal++;
  System.out.println("#1:" + iVal);
}

public static void addValue(int[] iaVal) {
  iaVal[0]++;
  System.out.println("#3:" + iaVal[0]);
}
```

Geben Sie basierend auf den Überlegungen am Speichermodell die Ausgaben #1 bis #4 an:

Referenz:	**Ausgabe:**
#1	
#2	
#3	
#4	