Primitive Datentypen*

Hans Jürgen Ohlbach

23. März 2018

Keywords: Ganze Zahlen, Gleitkommazahlen, Boolesche Werte, Buchstaben, Umgang mit einzelnen Bits

Empfohlene Vorkenntnisse: Digitale Arithmetik, Zeichenkodierung, Prozessorarchitektur

Inhaltsverzeichnis

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Einführung

In vielen Programmiersprachen unterscheidet man *primitive Datentypen* und *komplexe Datentypen*. Primitive Datentypen sind i.A. Wahrheitswerte (Boolesche Typen), ganze Zahlen (Integer Typen), Gleitkommazahlen (Float Typen) und Buchstaben (Character Typen). Aus rein mathematischer Sicht sind das eigentlich so einfache Strukturen, dass sich kaum darüber zu schreiben lohnt. Sieht man sich aber an, wie sie auf Computern realisiert sind, dann gibt es da ganz viele Details, die man als Programmierer wissen muss.

In diesem Miniskript wird beschrieben, wie diese Datentypen konkret realisiert sind, und zwar einmal für die Programmiersprache C, und zum anderen für die Programmiersprache Java. Die Programmiersprache C ist sehr nahe an der Prozessorarchitektur, so dass man illustrieren kann, wie diese Datentypen tatsächlich gespeichert und verarbeitet werden. Java ist eine viel abstraktere Sprache, die auf der Java Virtual Machine (JVM) aufbaut. Die Java Virtual Machine legt eine Abstraktionsschicht über die tatsächliche Prozessorarchitektur, in der diese Datentypen teilweise anders verarbeitet werden als in C.

2 Architekturen

Um zu verstehen, wie die primitiven Datentypen in Computern realisiert sind, müssen wir uns zunächst die konkrete Architektur der zugrunde liegenden Maschine vergegenwärtigen. Für die Programmiersprache C ist es die tatsächliche Prozessorarchitektur wie sie die Hardware vorgibt. Für die Programmiersprache Java ist es die Architektur der Java Virtual Machine. Da sie für alle Prozessortypen gleich ist, sieht sie im Detail anders aus als die Hardwarearchitektur.

2.1 Architektur für die Programmiersprache C

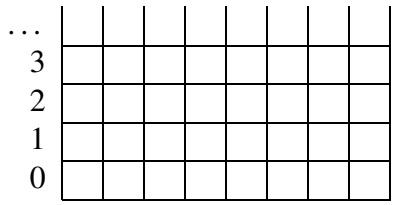
Für die Programmiersprache C ist die Struktur des Arbeitsspeichers entscheidend. Im Prozessor selbst gibt es zusätzlich *Register*, die zwar in C-Programmen unsichtbar sind, die aber trotzdem für das Verständnis der Verarbeitung der Datentypen wichtig sind[1].

[1] Zwischen Registern und Arbeitsspeicher liegen meist noch mehrere Ebenen von Cache-Speichern. Die sind aber für die jetzigen Betrachtungen nicht relevant.

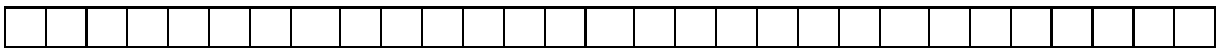
Der Arbeitsspeicher ist in fast allen Rechnern als Folgen von *Bytes*, d.h. 8-Bit Blöcken, organisiert. Jedes Byte hat eine *Adresse*, ab der Adresse 0.

Die Register fassen jedoch mehr Bits. Man hat heute die schon etwas veraltete 32-Bit Architektur und die neuere 64-Bit Architektur. Ein Register ist damit eine Folge von 32 bzw. 64 Bits (4 Bytes bzw. 8 Bytes) Die Registerbreite bestimmt damit auch die Anzahl von Bits, mit denen man eine Adresse im Arbeitsspeicher formulieren kann. In der 32-Bit Architektur sind es 4 Bytes und in der 64-Bit Architektur sind es 8 Bytes. Solche Adressen kann natürlich auch als Daten im Arbeitsspeicher ablegen. Sie nehmen dann 4 bzw. 8 Bytes an Platz ein.

Arbeitsspeicher



Register bei 32-Bit Architektur



Wichtig wird dieser Unterschied, wenn Daten aus dem Arbeitsspeicher in ein Register geladen werden sollen. Soll z.B. ein Byte aus dem Arbeitsspeicher in ein 8 Byte breites Register geladen werden, dann müssen die 7 restlichen Bytes aufgefüllt werden, entweder mit Nullen, oder mit Einsen. Für manche Anwendungen bedeutet es auch, dass es sich eigentlich nicht lohnt, mit sehr kurzen Datentypen zu arbeiten, da sie bei der Verarbeitung sowieso erst mal verlängert werden müssen.

Little-Endian und Big-Endian
Die Standardgröße einer Integerzahl in Binärdarstellung sind 32 Bit, d.h. 4 Bytes.

Beispiel:

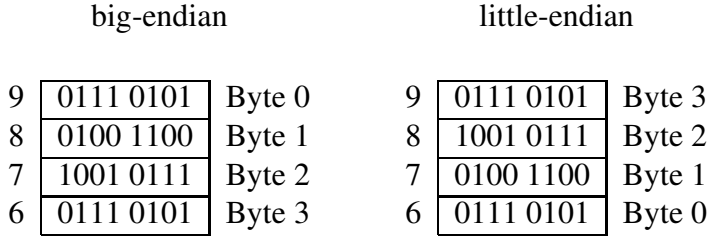
$$\underbrace{01110101}_{\text{Byte 3}}\,\underbrace{10010111}_{\text{Byte 2}}\,\underbrace{01001100}_{\text{Byte 1}}\,\underbrace{01101101}_{\text{Byte 0}}$$

Byte 3 wird als das *höchstwertige* Byte bezeichnet, und Byte 0 als das *niedrigstwertige* Byte. Das entspricht auch der Darstellung der Dezimalzahlen, z.B. 345, wo 3 an der Hunderterstelle ist, und damit den höchsten Wert hat, und 5 an der Einerstelle ist, und damit den niedrigsten Wert hat.

Speichert man diese 4 Bytes im Arbeitsspeicher ab, dann gibt es zwei Möglichkeiten:

big-endian: das höchstwertige Byte kommt an die kleinste Adresse
little-endian: das niedrigstwertige Byte kommt an die kleinste Adresse.

Wird das obige 32-Bit Wort z.B. ab der Adresse 6 gespeichert, dann sehen die beiden Möglichkeiten so aus:



Innerhalb eines Rechners oder einer Rechnerfamilie sieht der Benutzer normalerweise nichts davon, es sei denn sein Programm manipuliert Daten wirklich byteweise. Werden jedoch Daten an andere Rechner übertragen, und das geschieht oft byteweise, dann kann es damit zu Inkompatibilitäten kommen.

Die allermeisten Architekturen basieren auf Intel x86 oder AMD64 / x86-64, und diese speichern little-endian. Nur die IBM z/Architektur, Freescale ColdFire (basierend auf der Motorola 68000 Serie), Xilinx Microblaze, SuperH und Atmel AVR32 speichern big-endian.

2.2 Architektur für die Programmiersprache Java

Die Java Virtual Machine hat eine sog. *Stack-Architektur*. D.h. (fast) jeder Maschinenbefehl erwartet seine Argumente auf dem Stack, und legt sein Ergebnis wieder auf dem Stack ab. Um z.B. 2 Zahlen zu addieren, müssen die Zahlen vorher auf das Stack gelegt werden. Der Addierbefehl holt sich dann die beiden Zahlen (und löscht sie vom Stack) und schreibt die Summme wieder auf das Stack. Daher bestehen die meisten Maschinenbefehle nur aus der Nummer des Befehls selbst, ohne Argumente. Um diese Nummern zu repräsentieren reicht ein Byte. Deshalb werden die Maschineninstruktionen der JVM auch als *Byte Code* bezeichnet.

Was ganz wichtig zu wissen ist: Die Zellen im Stack *sind immer 32 Bit breit*, egal welche Architektur der Prozessor hat. D.h. *alle Datentypen werden in Einheiten von 32 Bit gespeichert*. Egal ob im Arbeitsspeicher little-endian oder big-endian gespeichert wird, sobald die Zahlen in das Stack kopiert werden, und dann verarbeitet werden, werden sie richtig ausgerichtet. Insofern spielt little-endian oder big-endian keine Rolle.

3 Integer Datentypen

Integer Typen sind die Computerversion von als endlichen Bitfolgen gespeicherten ganzen Zahlen in Binärdarstellung. Sie werden entweder in ein, zwei, vier oder acht Bytes gespeichert. Darüber hinaus gibt es noch Bibliotheken, mit denen man sehr große ganze Zahlen, so groß, bis der Arbeitsspeicher voll ist, speichern und verarbeiten kann.

Man unterscheidet sog. *unsigned* und *signed* Darstellung. „Unsigned“ bedeutet, dass die Zahl immer als positive Zahl oder 0 interpretiert wird. „Signed“ bedeutet Zweierkomplementdarstellung von positiven und negativen Zahlen.

3.1 Unsigned Integer

Unsigned (vorzeichenlose) Integer gibt es in der Programmiersprache C, und zwar als folgende Typen:

Name	Bytes	Größte Zahl
unsigned char	1	$2^8 - 1 =$ 255
unsigned short	2	$2^{16} - 1 =$ 65 535
unsigned int	4	$2^{32} - 1 =$ 4 294 967 295
unsigned long	8	$2^{64} - 1 =$ 18 446 744 073 709 551 615

Der Typ `char` war ursprünglich für die 7-Bit ASCII-Darstellung von Buchstaben gedacht. Er kann aber genauso gut für kleine Zahlen benutzt werden.

Arithmetik
Die Arithmetik innerhalb eines Typs ist im Prinzip wie die Mathematik sie vorgibt. Der einzige Unterschied ist das Verhalten bei Überläufen, d.h. wenn die Zahlen zu groß werden. C gibt dann keine Warnung aus, sondern schneidet einfach die führenden Bits, die zu viel sind, ab.

Beispiel:

```
#include<stdio.h>
int main(){
    unsigned char i = 255;
    unsigned char j = 2;
    unsigned char k = i+j;
    printf("%u",k);}
```

Die Direktive %u steht für die Ausgabe einer unsigned Integer Variablen.

Das Ergebnis ist 1, warum?

Zahl	Bitfolge
255	1111 1111
+ 2	0000 0010
257	1 0000 0001

Die führende 1 passt nicht in ein Byte und wird daher abgeschnitten.

Typumwandlung
Man kann problemlos einen kleinen Typ in einen großen Typ umwandeln. Z.B.

```
int main(){
    unsigned char i = 255;
    unsigned int j = i;
    printf("%u",j);}
```

(`#include<stdio.h>` sparen wir uns im Folgenden.)

Hier wird aus dem einen Byte von i ein 4-Byte Integer j gemacht, indem 24 führende Nullen hinzugefügt werden.

Kritischer ist die Umwandlung von großen Typen nach kleinen Typen.

```
int main(){
    unsigned int i = 257;
    unsigned char j = i;
    printf("%u",j);}
```

Für `int i = 257` werden 9 Bits benötigt. Kopiert man diese in einen 8-Bit `unsigned char` Typ, wird das führende Bit, welches nicht passt, abgeschnitten. Das Ergebnis ist daher 1.

3.2 Signed Integer

Signed Integer sind vorzeichenbehaftet. Negative ganze Zahlen werden dabei in *Zweierkomplementdarstellung* gespeichert. Der Grund ist, dass man dann Subtraktion als Addition mit dem Zweierkomplement realisieren kann. Für eine Bitfolge erhält man das Zweierkomplement, indem man von links nach rechts alle Bits umdreht, bis auf die rechteste 1, diese und alle Nachfolgenden Nullen bleiben unverändert.

Beispiel: $001101100 \rightarrow 110010100$.

Am linkesten Bit erkennt man, ob es sich um eine negative oder positive Zahl handelt. Ist es 0, dann ist es eine positive Zahl, ist es 1, dann ist es eine negative Zahl. Eine negative Zahl kann man auf genau die gleiche Weise wieder positiv machen: von links nach rechts all Bits umdrehen, bis auf die rechteste Eins. Die und die folgenden Nullen bleiben wie sie sind.

Folgende Signed-Integer Typen sind verfügbar:

Name	Bytes	Zahlenbereich
char/byte	1	$-128 - 127$
short	2	$-32768 - 32767$
int	4	$-2147483648 - 2147483647$
long	8	$-9,223,372,036,854,775,808 - 9,223,372,036,854,775,807.$

Der 1-Byte Type heißt in C `char` und in Java `byte`.

Arithmetik
Sind die Zahlen klein genug für den jeweiligen Typ, dann ist die Arithmetik in C und Java wiederum so wie sie die Mathematik beschreibt. Wenn die Zahlen allerdings zu groß werden, dann werden die internen Abläufe im Prozessor deutlich.

Zuweisungen: Eine Zuweisung `char a = 128;` in C erzeugt zunächst die Bitfolge 1000 0000, welche aber als Zweierkomplement der Zahl -128 entspricht. C führt die Anweisung ohne Warnung

aus. In Java würde `byte a = 128;` nicht compiliert werden. Mit der expliziten Typumwandlung `byte a = (byte)128;` wird es compiliert, und das Ergebnis ist ebenfalls -128.

Die Zuweisung `char a = 256;` in C erzeugt zunächst die Bitfolge 1 0000 0000 mit 9 Bits. Jetzt wird das führende Bit, welches nicht in ein Byte passt, einfach abgeschnitten. Das Ergebnis ist a = 0. In Java erhält man mit `byte a = (byte)256;` ebenfalls a = 0.

Arithmetische Operationen: Die Arithmetik ist standardmäßig auf 4-Byte Integer abgestimmt, In Java sowieso, weil die Einträge im Stack 4 Bytes lang sind, in C auch, sogar bei 64-Bit Architektur des Prozessors. Das äußert sich z.B. in folgendem Programm:

```
int main(){
    char i = 127;
    printf("%i",i+1);}
```

Der Ausdruck ist 128, und zwar weil `i+1` vom Typ int ist, obwohl i vom Typ char ist.

Anders wird es in folgendem Programm:

```
int main(){
    char i = 127;
    char j = i+1;
    printf("%i",j);}
```

Jetzt ist der Ausdruck -128. Die int-Bitfolge 0000 0000 1000 0000 im Ergebnis von i+1 wird jetzt explizit auf das 1-Byte `char j` reduziert. In dieser Darstellung ist das Ergebnis die negative Zahl -128.

D.h. generell werden die arithmetischen Operationen automatisch in der 4-Byte Darstellung durchgeführt, und dann wieder auf den gewünschten Typ reduziert, indem die führenden Bits, die zuviel sind, abgeschnitten werden. Das gilt auch, wenn mit int-Zahlen direkt gearbeitet wird. Hier werden Überläufe sofort wenn sie auftauchen, abgeschnitten, meist ohne Warnung.

In Java gilt im Prinzip das gleiche wie in C. Jedoch ist der Compiler penibler, und lässt manche Operationen nicht zu.

Folgendes Programm wird es gar nicht compiliert

```
public static void main(String[] args) {
    byte a = 127;
    byte b = a+1;
    System.out.println(b);}}
```

weil der Ausdruck `a+1` vom Typ `int` ist.

Erst mit *expliziter* Typumwandlung lässt sich der Compiler auf die Umwandlung ein:

```
public static void main(String[] args) {
    byte a = 127;
```

```
    byte b = (byte)(a+1);
    System.out.println(b);}}
```

Der Ausdruck ist dann, wie im C-Programm, -128.

3.3 Wrapperklassen in Java und Autoboxing

Zu jeder Integerklasse in Java gibt es eine sog. *Wrapperklasse*. Die Wrapperklassen kapseln die jeweilige Zahl eines primitiven Datentyps in ein Objekt. Z.B. `Integer drei = new Integer(3);` kapselt die int-Zahl 3 in ein Objekt. Die Anweisung erzeugt daher zwei Speicherzellen, eine mit der Zahl 3, und eine mit der Adresse dieser Zelle.

Konkret gibt es u.A. folgende Wrapperklassen:

primitiver Typ	Wrapperklasse
byte	Byte
short	Short
int	Integer
long	Long

Um die Schreibarbeit mit den Wrapperklassen zu vereinfachen, hat man eine abkürzende Schreibweise eingeführt. Anstelle von `Integer drei = new Integer(3);` kann man einfach schreiben: `Integer drei = 3;` (sog. *Autoboxing*). Nach außen wirken beide Versionen zunächst gleich.

Allerdings gibt es doch einen Unterschied zwischen der Konstruktor-Version `Integer i = new Integer(3);` und der Autoboxing-Version `Integer i = 3;`. Java hat einen internen Vorrat von Integer-Objekten, nämlich für die Zahlen -128 bis 127. Mit der Autoboxing-Version wird, falls möglich, die interne Version genommen. Mit der Konstruktor-Version wird immer ein neues Objekt erzeugt.

Den Unterschied merkt man bei folgendem Programm:

```
public class Test {
    public static void main(String[] args) {
        int k = 127;
        Integer i = k;
        Integer j = k;
        System.out.println(i == j);}}
```

Da $k \leq 128$, zeigt sowohl i, als auch j auf das gleiche Objekt. Daher wird `true` ausgedruckt[2].

Für `k = 128` gibt es kein passendes Integer Objekt mehr im Vorrat. Daher werden für i und j *zwei* neue Objekte erzeugt. Der Ausdruck ist dann `false`.

Anders ist es bei folgendem Programm:

[2] `i == j` testet, ob die Adressen von i und j gleich sind.

```
public class Test {
    public static void main(String[] args) {
        int k = 1;
        Integer i = new Integer(k);
        Integer j = new Integer(k);
        System.out.println(i == j);}}
```

Hier werden auf jeden Fall zwei neue Objekte erzeugt. Der Ausdruck ist dann auch `false`.

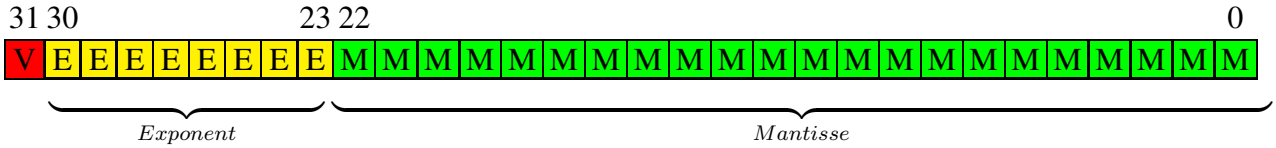
Einen Vorrat von Wrapper-Objekten für die Zahlen von -128 bis 128 gibt es nicht nur für die int Zahlen, sondern genauso für alle andere Integer-Typen.

4 Gleitkommazahlen

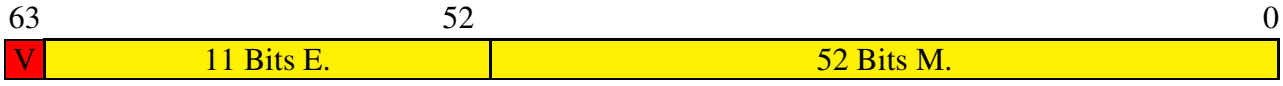
Die binäre Darstellung von Integer Zahlen ergibt sich aus der Mathematik, und die Aufteilung in 1,2,4,8-Byte Versionen ergibt sich aus dem Binärsystem, welches auch die Länge eines Bytes als 8 Bits nahelegt.

Für Gleitkommazahlen gibt es keine solche natürliche Strukturierung. Daher wurde deren Binärdarstellung durch den internationalen Standard IEEE-754 normiert. Sie sieht folgendermaßen aus:

32 Bit Floating Point Darstellung (single precision)



64 Bit Floating Point Darstellung (double precision)



V ist das Vorzeichen der Mantisse.

IEEE 745 definiert u.A. auch noch Gleitkommastandards mit 16 Bits, 128 Bits, 256 Bits. Diese werden aber nicht in allen Programmiersprachen unterstützt.

Auf dieser Standardisierung von Gleitkommazahlen basiert auch die Gleitkommaarithmetik in den Prozessoren. In den Programmiersprachen, sowohl in C als auch in Java, werden sie durch die Typen float (32 Bits) und double (64 Bits) unterstützt. In Java gibt es noch die zusätzlichen Wrapperklassen Float und Double.

Hinweis (für Experten): In C kann man folgendem Befehl den Exponenten einer float-Zahl extrahieren:

```
float a = ...
unsigned int i = *((unsigned int*)(&a));
int exponent = ((i << 1) >> 24) - 127;
```

`&a` ist die Adresse der float-Zahl `a`.
`(unsigned int*)(&a)` zwingt den Compiler, diese Adresse ab sofort als Adresse eines unsigned int-Wertes zu interpretieren.
`i` ist jetzt die Bitfolge des float-Wertes, als unsigned int-Wert interpretiert.
Mit `i << 1` wird das Vorzeichenbit eliminiert.
Mit `>> 24` wird diese Bitfolge um 24 Bits nach rechts verschoben, um an den Exponenten zu kommen. Der Exponent selbst wird in sog. *biased Darstellung* gespeichert. Um den wahren Exponenten zu bekommen, muss man 127 abziehen.

Für `a = 4.25`, z.B., wird `exponent = 2`
Für `a = -10`, z.B., wird `exponent = 3`.

Gleitkommazahlen sind eine endliche Approximation von rationalen und reellen Zahlen. Als Approximation ist aber nicht garantiert, dass sie exakt sind. Z.B. lässt sich der Bruch 1/3 nicht exakt mit endlich vielen Bits darstellen. Es muss gerundet werden, und Rundungen erzeugen Rechenfehler.

Man probiere folgendes Java Programm:

```
public static void main(String[] args) {
    double a = 17.31f;
    double b = 100f;
    System.out.println(a*b);}
```

Die Ausgabe ist nicht 1731, sondern 1730.9999465942383.

Im Gebiet *Numerik* der Mathematik ist ein wichtiger Schwerpunkt der Umgang mit Rundungsfehlern, die sich bei vielen Algorithmen soweit aufschaukeln können, dass die Ergebnisse unbrauchbar werden.

5 Boolesche Datentypen

Eigentlich gibt es nur einen einzigen Booleschen Typ. Seine Werte sind *wahr* und *falsch* bzw. 1 und 0. Zu deren Speicherung genügt *ein einziges Bit*. Ein Bit alleine ist aber in keiner Rechnerarchitektur adressierbar, sondern nur ein Byte. Daher muss ein Boolescher Wert immer in eine größere Bitfolge eingebettet werden.

5.1 Der Boolsche Datentyp in C

Die kleinste adressierbare Einheit in den meisten Prozessoren und in der Programmiersprache C ist ein Byte. Daher muss ein Boolescher Wert in 8 Bits verpackt werden.

Für die 0 ist es einfach: Die Bits 00000000 stellen den Booleschen Wert 0 dar.

Für die 1 gibt es zwei Möglichkeiten:
- 00000001 stellt den Booleschen Wert 1 dar
- jede Bitfolge ungleich 00000000 stellt den Wert 1 dar.

In der Programmiersprache C gibt es den Datentyp _Bool. Er repräsentiert Boolesche Werte als ein Byte. Das kann man z.B. mit folgendem Programm testen.

```
_Bool i,j;
int main(){
  printf("%p␣%p␣\n", &i, &j);}
```

i und j sind Boolesche Werte, die als Bytes abgespeichert werden. &i sowie &j stellen die Adressen der Speicherzellen dar, in der diese Werte abgespeichert sind. Der Ausdruck könnte so aussehen: 0x601051 0x601050.[3]

Wichtig dabei ist, dass zwischen den beiden Hexadezimalzahlen ein Unterschied von 1 ist. Daran sieht man, dass i und j nacheinander in zwei Byte-Zellen gespeichert sind.

Um zu testen, wie genau die 1 abgespeichert ist, kann man folgendes ausprobieren:

```
int main(){
  int m = 255;
  _Bool i = m;
  printf("%d␣%d\n",i,m);}
```

Das Ergebnis ist 1 255.

Die binäre Darstellung des int-Werts 255 ist 000000000 00000000 00000000 11111111. Bei der Umwandlung in den Typ _Bool wurde offensichtlich getestet, ob der int-Wert ungleich 0 ist. Daraus wurde dann der _Bool-Wert 00000001 erzeugt. Das funktioniert auch wenn man m = 256 wählt. In Binärdarstellung besteht das niedrigstwertige Byte von 256 aus acht Nullen. Trotzdem wird daraus der _Bool-Wert 1 erzeugt.

Allerdings kommen viele C-Programme ohne den Datentyp _Bool aus (den gibt es erst seit einiger Zeit). Statt _Bool wurden einfach int-Werte genommen. Dabei ist die Konvention: jeder int-Wert ungleich 0 repräsentiert die 1. In folgendem Programm:

```
int main(){
   int m = 255;
   if(m) {printf("ja\n");}}
```

[3]Die printf-Funktion erwartet einen Kontrollstring, gefolgt von beliebig vielen Argumenten. Der Kontrollstring spezifiziert, wie diese Argumente ausgedruckt werden. %p bedeutet: Ausdruck als Adresse in Hex-Schreibweise.

wird der int-Wert m in der if-Abfrage als Boolescher Wert interpretiert. Da er ungleich 0 ist, wird tatsächlich `ja` ausgedruckt.

5.2 Die Booleschen Datentypen in Java

In Java gibt es zunächst den primitiven Datentyp `boolean`. Eine boolean Variable hat immer den Wert `true` oder `false`. Diese Werte werden aber schon vom Java Compiler auf die int-Werte 1 bzw. 0 abgebildet. (Das kann man sehen, wenn man einen compilierten .class File mit dem Java Disassembler javap ansieht.) Die int-Werte werden wiederum in 32-Bit Darstellung gespeichert. Anders als in C verhindert der Java-Compiler aber jegliche Umwandlung von anderen Datentypen auf den Typ boolean.

Zusätzlich zu boolean gibt es in Java noch die Wrapperklasse `Boolean`. Boolean kapselt einen boolean Wert als Objekt.

Eine Anweisung `Boolean b = new Boolean(true);` erzeugt damit zwei Speicherzellen: eine mit dem boolean Wert 1, und eine mit der *Adresse* auf diese Speicherzelle.

Die obige Anweisung kann man auch verkürzt so schreiben: `Boolean b = true;` (Autoboxing). In diesem Fall wird jedoch das vordefinierte Boolean-Objekt für true genommen, und nicht ein neues Objekt per Konstruktor erzeugt.

6 Ein einziges Bit als Datenobjekt

Obwohl die kleinste adressierbare Einheit ein Byte ist, haben findige Programmierer eine Möglichkeit gefunden, mit einzelnen Bits zu arbeiten. Als Motivationsbeispiel betrachten wir ein Programm, welches Texte ausdruckt. Der Text kann fett, kursiv, unterstrichen oder gespreizt gedruckt werden. Das kann man natürlich mit vier Booleschen Variablen kontrollieren. Die bräuchten aber 4 Bytes, obwohl eigentlich 4 Bits ausreichen würden. Ein `byte`-Typ in Java hat 8 Bits, also völlig ausreichend.

Der Trick funktioniert folgendermaßen:

Wir führen vier statische byte-Variablen ein:

```
static final byte fett          = 1;  // = 0000 0001
static final byte kursiv        = 2;  // = 0000 0010
static final byte unterstrichen = 4;  // = 0000 0100
static final byte gespreizt     = 8;  // = 0000 1000
```

Um z.B. den Text jetzt fett und unterstrichen zu drucken, erzeugen wir eine Variable:

```
byte kontrolle = fett | unterstrichen; // = 0000 0101
```

`|` ist das bitweise logische *oder*.

Diese Variable wird an das Druckprogramm übergeben.

Dort könnte eine if-Abfrage stehen:

```
if((kontrolle & fett) != 0){...}
```

wobei & das logische *und* ist. Im obigen Fall ergibt es:

kontrolle	0000 0101
& fett	0000 0001
=	0000 0001 ($\neq 0$)

Da die `kontrolle`-Variable an der `fett`-Position eine 1 hat, ist `(kontrolle & fett) != 0` wahr.

Genauso kann man alle anderen Varianten abfragen.

Allgemeine Vorgehensweise:
Für jeden binären Kontrollparameter definiert man eine Position in einem Bitstring. Beim int-Typ hat man da sogar 32 Positionen. Die Definition dieser Position macht man am besten mit einer statischen final Variablen.

Um die Position n festzulegen, geht es am einfachsten mit dem Linksshift:
`static final int xn = 1 << n;.`

Ein konkretes Kontrollmuster erzeugt man mit dem logischen *oder*, z.B.:

```
int kontrolle = x3 | x5 | x9;
```

Die Abfrage, welches Bit gesetzt ist, geschieht mit dem logischen *und*, z.B.
`if((kontrolle & x5) != 0){...}`

Der Vorteil dieser Vorgehensweise ist, dass man mit einer einzigen int-Variablen als Kontrolle bis zu 32 Bits übertragen kann. Hat man noch nicht alle 32 Bits genutzt, kann man sogar problemlos in einer neueren Version des Programms mehr Kontrollbits übergeben, ohne die Argumente von Methoden ändern zu müssen.

7 Buchstaben

In den Anfangszeiten der Computer war die englische Sprache als Kommunikationsmittel noch ausreichend. Englisch kommt sehr gut mit den 26 Buchstaben des Alphabets aus. Mit Groß- und Kleinschreibung macht das 52 Buchstaben. Mit noch ein paar Sonderzeichen wurde der 7-Bit ASCII-Code geboren. 7 Bits passen in ein Byte, wobei das linkeste Bit immer 0 ist. Für viele andere Sprachen reichen die Buchstaben des Alphabets aber nicht aus. Indem man das 8. Bit eines Bytes auf 1 setzt, hat man jedoch noch weitere 128 Buchstaben zur Verfügung. Das reicht für die meisten Sprachen aus.

Jetzt passierte der Sündenfall: viele Kulturkreise, die zusätzliche Buchstaben brauchen, definierten ihre eigene Erweiterung von ASCII, z.B. ISO-8859-1 für westeuropäische Sprachen, ISO-8859-7 für Griechisch, ISO-8859-8 für Keltisch und noch viele weitere. Erhält man einen Text von einem fremden Computer, dann ist nicht so ohne weiteres zu sehen, welche Kodierung verwendet wurde.

Mit *Unicode* wurde dann versucht, eine Vereinheitlichung der Kodierungen zu erreichen. Aber auch bei Unicode gab es eine Entwicklung. Zunächst wurde die Basic Multilingual Plane (BMP) entwickelt, die in 16 Bits passt. Damit kann man 65535 Zeichen kodieren. Leider reicht das immer noch nicht für alle Zeichen in allen Sprachen der Erde. Bisher wurden noch 16 weitere 16-Bit Blöcke (Ebenen) hinzugefügt, so dass man derzeit auf 1.114.112 Zeichen kommt. Um wirklich alle Zeichen nutzen zu können, braucht man dann 21 Bits. Ebene 17 ist allerdings nicht standardisiert, so dass man bei 16 Ebenen mit 20 Bits auskommt. Jeder Buchstabe in der Unicode Kodierung hat eine eindeutige Nummer (Codepoint), die man mit der Schreibweise \uxxxx oder \uxxxx xxxx eingeben kann, wobei xxxx für die Hexadezimalkodierung der Nummer steht. Z.B. steht \u03C3 für das griechische σ.

Analog zu dieser Entwicklung hat sich auch die Zeichenkodierung in Programmiersprachen entwickelt. Der Typ `char` in C mit seinen 8 Bits reicht aus, um ASCII und auch die unterschiedlichen ISO-8859 Erweiterungen zu kodieren. Unicode wird in C nicht direkt unterstützt. Es gibt aber die Bibliothek `uchar.h`, mit der man auch in C mit Unicode Buchstaben arbeiten kann.

In Java wurde gleich zu Beginn der Typ `char` als 16-Bit Block eingeführt. Das genügt für die BMP von Unicode. Buchstaben aus höheren Ebenen von Unicode müssen als 32-Bit int-Werte gespeichert werden.

Einzelne Buchstaben zu speichern ist wahrscheinlich nicht so interessant. Wichtiger ist die Speicherung und Verarbeitung von Zeichenketten (Strings). Was es dazu sagen gibt, steht im Miniskript zu Arrays, Stacks und Strings.

Stichwortverzeichnis