TECHNISCHE UNIVERSITÄT DARMSTADT

# Kapitel 11:
# Interne Zahldarstellung

Karsten Weihe

## Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

**Ganze Zahlen:**
- byte
- short
- int
- long

**Gebrochene Zahlen:**
- float
- double

**Logik:**
- boolean

**Zeichen:**
- char

***Erinnerung*: Im Abschnitt „Allgemein: Primitive Datentypen“ in Kapitel 01b hatten wir schon diese Auflistung aller primitiven Datentypen gesehen, ...**

## Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

**Ganze Zahlen:**
- byte
- short
- int
- long

**Gebrochene Zahlen:**
- float
- double

**Logik:**
- boolean

**Zeichen:**
- char

**... die vier ganzzahligen Typen ...**

## Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

**Ganze Zahlen:**
- byte
- short
- int
- long

**Gebrochene Zahlen:**
- float
- double

**Logik:**
- boolean

**Zeichen:**
- char

... und die beiden gebrochenzahligen Typen.

## Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

**Ganze Zahlen:**
- byte
- short
- int
- long

**Gebrochene Zahlen:**
- float
- double

**Logik:**
- boolean

**Zeichen:**
- char

**Diese beiden primitiven Datentypen sind nicht im Fokus dieses Kapitels, die sprechen wir im Folgenden nicht mehr an.**

TECHNISCHE UNIVERSITÄT DARMSTADT

# Vorarbeit:
# Zahlensysteme zu unterschiedlichen Basen

**Wir rekapitulieren hier nur ganz, ganz kurz Stoff aus der gymnasialen Mittelstufe beziehungsweise vergleichbaren Bildungswegen. Wie Sie wissen, ist unser dekadisches Zahlensystem nur *ein* mögliches System. Jede andere ganze Zahl größer 1 käme genauso als Basis in Frage.**

**Auf Standardcomputern wird im binären System gerechnet, also zur Basis 2.**

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

$$107_{10} = 1 \cdot 10^2 + 7 \cdot 10^0$$

$$107_{10} = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1101011_2$$

$$107_{10} = 1 \cdot 3^4 + 2 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 10222_3$$

$$107_{10} = 1 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 1223_4$$

$$107_{10} = 6 \cdot 16^1 + B \cdot 16^0 = 6B_{16}$$

**Hier sehen Sie ein einfaches Beispiel: die Dezimalzahl 107 dargestellt in verschiedenen Zahlensystemen.**

**Die Gleichheitszeichen auf dieser Folie bedeuten wie in der Mathematik Wertgleichheit. Wenn man mit verschiedenen Basen zugleich umgeht, schreibt man die Basis zur Vermeidung von Missverständnissen als Subskript an die Zahl. Bei den Rechnungen zwischen den Gleichheitszeichen sind alle Zahlen dezimal, die Basis 10 schreiben wir aber der Übersichtlichkeit halber dort nicht dazu.**

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

$107_{10} = 1 \cdot 10^2 + 7 \cdot 10^0$

$107_{10} = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1101011_2$

$107_{10} = 1 \cdot 3^4 + 2 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 10222_3$

$107_{10} = 1 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 1223_4$

$107_{10} = 6 \cdot 16^1 + B \cdot 16^0 = 6B_{16}$

**Die 107 wird bekanntlich im Dezimalsystem deshalb so und nicht anders geschrieben, weil die 107 kleiner als 10 hoch 3 ist, die 10 hoch 2 genau einmal hineinpasst, die 10 hoch 1 kein Mal in den Rest 7 passt und die 10 hoch 0 siebenmal in den Rest 7 passt.**

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

$$107_{10} = 1 \cdot 10^2 + 7 \cdot 10^0$$

$$107_{10} = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1101011_2$$

$$107_{10} = 1 \cdot 3^4 + 2 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 10222_3$$

$$107_{10} = 1 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 1223_4$$

$$107_{10} = 6 \cdot 16^1 + B \cdot 16^0 = 6B_{16}$$

**Nach exakt derselben Logik können wir diese Zahl auch im Binärsystem darstellen, wir müssen nur die Basis 10 durch die Basis 2 ersetzen.**

**Bei Basis 10 gibt es die Ziffern 0 bis 9, bei Basis 2 entsprechend nur die Ziffern 0 und 1.**

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

$$107_{10} = 1 \cdot 10^2 + 7 \cdot 10^0$$

$$107_{10} = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1101011_2$$

$$107_{10} = 1 \cdot 3^4 + 2 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 10222_3$$

$$107_{10} = 1 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 1223_4$$

$$107_{10} = 6 \cdot 16^1 + B \cdot 16^0 = 6B_{16}$$

**Dasselbe kann man dann auch für die Basis 3 machen, die möglichen Ziffern sind dann 0, 1 und 2: Die 3 hoch 4 passt einmal in die 107, das ergibt Rest dezimal 26. Die 3 hoch 3 passt nicht in die 26 hinein, also ist die nächste Ziffer eine 0. Aber die 3 hoch 2 passt zweimal hinein, der Rest ist dezimal 8. Da passt 2 hoch 1 zweimal und in den Rest davon wiederum 2 hoch 0 zweimal hinein.**

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

$$107_{10} = 1 \cdot 10^2 + 7 \cdot 10^0$$

$$107_{10} = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1101011_2$$

$$107_{10} = 1 \cdot 3^4 + 2 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 10222_3$$

$$107_{10} = 1 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 1223_4$$

$$107_{10} = 6 \cdot 16^1 + B \cdot 16^0 = 6B_{16}$$

**Genauso geht es mit der Basis 4 und den Ziffern 0 bis 3.**

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

$$107_{10} = 1 \cdot 10^2 + 7 \cdot 10^0$$

$$107_{10} = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1101011_2$$

$$107_{10} = 1 \cdot 3^4 + 2 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 10222_3$$

$$107_{10} = 1 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0 = 1223_4$$

$$107_{10} = 6 \cdot 16^1 + B \cdot 16^0 = 6B_{16}$$

**Die Basis muss nicht kleiner als dezimal 10 sein. Sehr häufig kommt in der Informatik die Basis 16 vor, dieses Zahlensystem heißt Hexadezimalzahlsystem (hexa = 6). Dafür werden aber neben 0 bis 9 noch weitere sechs Ziffern benötigt, nämlich für die Zahlenwerte dezimal 10 bis 15. Traditionell nimmt man dafür die ersten sechs Großbuchstaben des Alphabets, A für dezimal 10, B für dezimal 11 bis F für dezimal 15. Ansonsten ist alles exakt so wie in den anderen Zeilen dieser Folie.**

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

| | |
|---|---|
| 233 | 11101001 |
| + 107 | + 1101011 |
| -------- | ---------- |
| 340 | 101010100 |
| | |
| 233 | 11101001 |
| – 107 | – 1101011 |
| -------- | ---------- |
| 126 | 1111110 |

**Die arithmetischen Operationen sind in allen diesen Zahlensystemen völlig analog. Wir stellen hier nur beispielhaft Addition und Subtraktion im Dezimalsystem und im Binärsystem einander gegenüber. Jede Dezimalzahl links hat denselben Wert wie die Binärzahl in derselben Zeile rechts.**

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  233          11101001
+ 107        +   1101011
--------     ------------
  340        101010100

  233          11101001
– 107        – 1101011
--------     ------------
  126          1111110
```

**Wenn kein Übertrag aus der vorherigen Spalte zu berücksichtigen ist, werden einfach die Ziffern addiert beziehungsweise subtrahiert, wobei potentiell ein positiver Übertrag entstehen kann. Bei der Addition oder Subtraktion von nur zwei Zahlen ist in jedem Zahlensystem der Übertrag immer 0 oder 1.**

## Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  233          11101001
+ 107        +   1101011
--------     ------------
  340        101010100

  233          11101001
– 107        – 1101011
--------     ------------
  126          1111110
```

**Hier sehen Sie in jedem der vier Fälle exemplarisch die Situation, dass ein Übertrag von der vorherigen Spalte einzuberechnen ist.**

**Damit schließen wir unsere kleine Erinnerung an die Mittelstufe ab und kommen zum eigentlichen Thema dieses Kapitels.**

TECHNISCHE UNIVERSITÄT DARMSTADT

# Ganze Zahlen

**Zuerst die vier ganzzahligen Datentypen.**

## Ganze Zahlen: Bereiche

TECHNISCHE UNIVERSITÄT DARMSTADT

| Typ | Bits | Bereich |
|---|---|---|
| byte | 8 | -128 ... +127 |
| short | 16 | -32.768 ... +32.767 |
| int | 32 | -2,147,483,648 ... +2,147,483,647 |
| long | 64 | -9,223,372,036,854,775,808 ... +9,223,372,036,854,775,807 |

**Auch diese Folie haben Sie schon in Kapitel 01b im Abschnitt zu primitiven Datentypen gesehen, Die einzelnen ganzzahligen Datentypen unterscheiden sich in der Anzahl Bits. Ein Datentyp mit N Bits kann 2 hoch N viele Zahlen darstellen.**

# Ganze Zahlen: Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

**Maximaler positiver/negativer int-Wert:**

- **Integer.MAX_VALUE**
- **Integer.MIN_VALUE**

**➔Analog byte (Byte), short (Short), long (Long)**

**Diese Folie mit den Klassenkonstanten, die die Grenzen der vier Wertebereiche auf der letzten Folie als Werte enthalten, hatten Sie ebenfalls in Kapitel 01b schon gesehen.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Nichtnegative Zahlen mit N Bits:

| | |
|---|---|
| 000...000 | 0 |
| 000...001 | 1 |
| 000...010 | 2 |
| ... | |
| 011...110 | $2^{N-1} - 2$ |
| 011...111 | $2^{N-1} - 1$ |

**Zum tieferen Verständnis der Informatik allgemein und zum Verständnis einiger Aspekte von Java befassen wir uns in diesem Kapitel damit, wie Zahlen in diesen ganzzahligen Datentypen kodiert sind. Wir beginnen erst einmal mit dem einfacheren Fall, Kodierung nichtnegativer ganzer Zahlen, also die Null und positive ganze Zahlen.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Nichtnegative Zahlen mit N Bits:

| | |
|---|---|
| 000...000 | 0 |
| 000...001 | 1 |
| 000...010 | 2 |
| ... | |
| 011...110 | $2^{N-1} - 2$ |
| 011...111 | $2^{N-1} - 1$ |

**Nichtnegative Zahlen erkennt man in jedem der ganzzahligen Datentypen daran, dass das führende Bit Null ist. Wir werden gleich sehen, dass dieses Bit bei negativen Zahlen Eins ist. Daher nennt man dieses Bit häufig das *Vorzeichenbit.* Man denkt sich also Null als Vorzeichen Plus und Eins als Vorzeichen Minus. Dieser Begriff ist allerdings etwas ungenau, denn der Zahlenwert 0 hat ja eigentlich kein Vorzeichen.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Nichtnegative Zahlen mit N Bits:

| | |
|---|---|
| 000...000 | 0 |
| 000...001 | 1 |
| 000...010 | 2 |
| ... | |
| 011...110 | $2^{N-1} - 2$ |
| 011...111 | $2^{N-1} - 1$ |

**Der Zahlenwert 0 ist einfach kodiert durch Nullen als Bitmuster.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Nichtnegative Zahlen mit N Bits:

| | |
|---|---|
| 000...000 | 0 |
| 000...001 | 1 |
| 000...010 | 2 |
| ... | |
| 011...110 | $2^{N-1} - 2$ |
| 011...111 | $2^{N-1} - 1$ |

**Das rechteste Bit, auch das am wenigsten signifikante Bit genannt, ist die Einerstelle im Binärsystem. Dieses Bit ist 1 bei allen nichtnegativen ganzen Zahlen, die ungerade, also nicht durch 2 teilbar sind. Bei der Zahl 1 selbst ist natürlich nur dieses eine Bit ungleich 0.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Nichtnegative Zahlen mit N Bits:

| | |
|---|---|
| 000...000 | 0 |
| 000...001 | 1 |
| 000...010 | 2 |
| ... | |
| 011...110 | $2^{N-1} - 2$ |
| 011...111 | $2^{N-1} - 1$ |

**Bei der Zahl 2 ist im binären System hingegen eine 1 an der Zweierstelle und ansonsten alles 0.**

**Dieses Bit ist gleich 1 bei allen Zahlen, die durch 2, aber nicht durch 4 teilbar sind.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Nichtnegative Zahlen mit N Bits:

| | |
|---|---|
| 000...000 | 0 |
| 000...001 | 1 |
| 000...010 | 2 |
| ... | |
| 011...110 | $2^{N-1} - 2$ |
| 011...111 | $2^{N-1} - 1$ |

**Wir überspringen die weiteren Zahlen bis zu den allergrößten darstellbaren.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Nichtnegative Zahlen mit N Bits:

| | |
|---|---|
| 000...000 | 0 |
| 000...001 | 1 |
| 000...010 | 2 |
| ... | |
| 011...110 | $2^{N-1} - 2$ |
| 011...111 | $2^{N-1} - 1$ |

**Das ist die größte darstellbare Zahl. Jedes Bit außer dem Vorzeichenbit ist 1, also die Einerstelle, die Zweierstelle, die Viererstelle, die Achterstelle und so weiter. Aus der Schulmathematik ist bekannt, dass die Summe der Zweierpotenzen – also 2 hoch 0 plus 2 hoch 1 plus 2 hoch 2 und so weiter – gleich der nächsthöheren Zweierpotenz minus 1 ist.**

**Zum Beispiel bei Datentyp byte mit acht Bit, also N gleich 8, ergibt sich hier 2 hoch 0 bis 2 hoch 6, das ist 127, und das ist 2 hoch 7 minus 1.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Nichtnegative Zahlen mit N Bits:

| | |
|---|---|
| 000...000 | 0 |
| 000...001 | 1 |
| 000...010 | 2 |
| ... | |
| 011...110 | $2^{N-1} - 2$ |
| 011...111 | $2^{N-1} - 1$ |

**Die zweitgrößte Zahl ist genau 1 weniger, also die Einerstelle auf Null statt auf 1.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$
$00001010_2 = 10_{10}$
$01100100_2 = 100_{10}$

$01101101_2 \;/\; 01100100_2 = 00000001_2$ → "1??"
$01101101_2 \;\%\; 01100100_2 = 00001001_2$
$00001001_2 \;/\; 00001010_2 = 00000000_2$ → "10?"
$00001001_2 \;\%\; 00001010_2 = 00001001_2$
$00001001_2 \;/\; 00000001_2 = 00001001_2$ → "109"

Wir schauen uns jetzt an, wie aus einer nichtnegativen Binärstellung eine Zeichendarstellung im Dezimalsystem berechnet wird, denn eine Zahl wird ja üblicherweise auf dem Bildschirm oder auf dem Drucker als Dezimalzahl ausgegeben. Um das Ganze übersichtlich zu halten, schauen wir uns nur den kleinsten ganzzahligen Datentyp an, Datentyp byte mit acht Bit. Für größere Datentypen funktioniert alles völlig analog, nur mit mehr Zehnerpotenzen und mehr einzelnen Verfahrensschritten.

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$
$00001010_2 = 10_{10}$
$01100100_2 = 100_{10}$

$01101101_2$ / $01100100_2 = 00000001_2$ → "1??"
$01101101_2$ % $01100100_2 = 00001001_2$
$00001001_2$ / $00001010_2 = 00000000_2$ → "10?"
$00001001_2$ % $00001010_2 = 00001001_2$
$00001001_2$ / $00000001_2 = 00001001_2$ → "109"

**Die Bitmuster für die Zehnerpotenzen sind an einer bestimmten Stelle im Speicher, die uns hier nicht interessiert, hinterlegt. Der Datentyp byte kann maximal die Zahl 127 darstellen, daher brauchen wir für Byte nur die Zehnerpotenzen bis 100.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$

$00001010_2 = 10_{10}$

$01100100_2 = 100_{10}$

$01101101_2$ / $01100100_2 = 00000001_2$ → "1??"

$01101101_2$ % $01100100_2 = 00001001_2$

$00001001_2$ / $00001010_2 = 00000000_2$ → "10?"

$00001001_2$ % $00001010_2 = 00001001_2$

$00001001_2$ / $00000001_2 = 00001001_2$ → "109"

**Diese Binärzahl soll in Dezimalzeichen umgewandelt werden. Die Einer-, Vierer-, Achter-, Zweiunddreißiger- und Vierundsechzigerstelle sind 1, diese Binärahl ist also 1 + 4 + 8 + 32 + 64 gleich 109.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$

$00001010_2 = 10_{10}$

$01100100_2 = 100_{10}$

$01101101_2$ / $01100100_2 = 00000001_2$ → "1??"

$01101101_2$ % $01100100_2 = 00001001_2$

$00001001_2$ / $00001010_2 = 00000000_2$ → "10?"

$00001001_2$ % $00001010_2 = 00001001_2$

$00001001_2$ / $00000001_2 = 00001001_2$ → "109"

**Die erste Stelle der Dezimaldarstellung bekommen wir, indem wir die umzurechnende Zahl ganzzahlig durch 100 dividieren. Im Falle der 109 ist das Ergebnis 1, denn 109 geteilt durch 100 ist 1 Rest 9.**

**Wäre das Ergebnis eine 0 gewesen, also wäre die umzurechnende Zahl kleiner als 100 gewesen, dann wäre die 0 in der Regel nicht in die Dezimaldarstellung geschrieben worden, denn führende Nullen werden typischerweise bei der Ausgabe unterdrückt.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$
$00001010_2 = 10_{10}$
$01100100_2 = 100_{10}$

$01101101_2 \;/\; 01100100_2 = 00000001_2$ → "1??"
$01101101_2 \;\%\; 01100100_2 = 00001001_2$
$00001001_2 \;/\; 00001010_2 = 00000000_2$ → "10?"
$00001001_2 \;\%\; 00001010_2 = 00001001_2$
$00001001_2 \;/\; 00000001_2 = 00001001_2$ → "109"

**Jetzt müssen noch die Zehner- und die Einerstelle berechnet werden. Diese setzen sich aus dem zusammen, was von der ursprünglichen Zahl 109 übrig bleibt, wenn man die ganzen Hunderter abzieht.**

**Dieses Ergebnis bekommen wir durch Modulo-Rechnung: Die ganzzahlige Division von 109 durch 100 ergibt eben 109 minus 100, also 9.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$
$00001010_2 = 10_{10}$
$01100100_2 = 100_{10}$

$01101101_2 \;/\; 01100100_2 = 00000001_2$ → "1??"
$01101101_2 \;\%\; 01100100_2 = 00001001_2$
$00001001_2 \;/\; 00001010_2 = 00000000_2$ → "10?"
$00001001_2 \;\%\; 00001010_2 = 00001001_2$
$00001001_2 \;/\; 00000001_2 = 00001001_2$ → "109"

So wie wir als erstes den ursprünglichen Zahlenwert durch die höchste Zehnerpotenz ganzzahlig geteilt haben, um die erste Dezimalstelle zu erhalten, so dividieren wir für die zweite Dezimalstelle jetzt den Rest, nachdem die Hunderter durch Modulo abgezogen sind, durch die nächstkleinere Zehnerpotenz, also 10. Das Ergebnis von 9 durch 10 ist 0, und das ist unsere nächste Dezimalstelle.

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$
$00001010_2 = 10_{10}$
$01100100_2 = 100_{10}$

$01101101_2 \;/\; 01100100_2 = 00000001_2$ → "1??"
$01101101_2 \;\%\; 01100100_2 = 00001001_2$
$00001001_2 \;/\; 00001010_2 = 00000000_2$ → "10?"
$00001001_2 \;\%\; 00001010_2 = 00001001_2$
$00001001_2 \;/\; 00000001_2 = 00001001_2$ → "109"

**Und so wie wir in der ersten Runde die ganzen Hunderter von 109 abgezogen haben, müssen wir jetzt die ganzen Zehner von 9 abziehen, also modulo 10 rechnen.**

***Wir* wissen schon vorher, dass sich dadurch nichts ändern wird, 9 modulo 10 ist nun einmal 9. Aber das Verfahren weiß das natürlich nicht, sondern rechnet stur weiter.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$
$00001010_2 = 10_{10}$
$01100100_2 = 100_{10}$

$01101101_2 \;/\; 01100100_2 = 00000001_2$ → "1??"
$01101101_2 \;\%\; 01100100_2 = 00001001_2$
$00001001_2 \;/\; 00001010_2 = 00000000_2$ → "10?"
$00001001_2 \;\%\; 00001010_2 = 00001001_2$
$00001001_2 \;/\; 00000001_2 = 00001001_2$ → "109"

**Die letzte Stelle bekommen wir wieder völlig analog mittels Division durch die letzte Zehnerpotenz, also die 1. Ganzzahlige Division durch 1 ändert natürlich nie etwas am Ergebnis, daher ist es nicht unwahrscheinlich, dass dieser letzte Schritt bei realen Umrechnern ausgespart wird, um nicht unnötig die Laufzeit zu erhöhen.**

**Hier auf den Folien interessiert uns aber eher die Systematik als die Laufzeit, daher bleiben wir ganz schematisch und machen auch diesen letzten, eigentlich überflüssigen Schritt noch. Das Ergebnis ist also die 9.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Nichtnegativ: Umwandlung in Dezimalzeichen

$00000001_2 = 1_{10}$
$00001010_2 = 10_{10}$
$01100100_2 = 100_{10}$

$01101101_2 \;/\; 01100100_2 = 00000001_2$ → "1??"
$01101101_2 \;\%\; 01100100_2 = 00001001_2$
$00001001_2 \;/\; 00001010_2 = 00000000_2$ → "10?"
$00001001_2 \;\%\; 00001010_2 = 00001001_2$
$00001001_2 \;/\; 00000001_2 = 00001001_2$ → "109"

**Insgesamt sehen wir eine Schleife, die in absteigender Reihenfolge durch alle Zehnerpotenzen geht, die in den Datentyp hineinpassen. In jedem Durchlauf der Schleife wird von vorne nach hinten jeweils die nächste Stelle der Dezimaldarstellung berechnet, indem der momentane Rest durch die zugehörige Zehnerpotenz dividiert wird. Für die weiteren Stellen der Dezimaldarstellung rechnen wir jeweils mit dem Rest dieser Division weiter, was im letzten Durchlauf natürlich unterbleiben kann.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

$00000000_2$ → $00110000_2$ → ´0´
$00000001_2$ → $00110001_2$ → ´1´
$00000010_2$ → $00110010_2$ → ´2´
$00000011_2$ → $00110011_2$ → ´3´
$00000100_2$ → $00110100_2$ → ´4´
$00000101_2$ → $00110101_2$ → ´5´
$00000110_2$ → $00110110_2$ → ´6´
$00000111_2$ → $00110111_2$ → ´7´
$00001000_2$ → $00111000_2$ → ´8´
$00001001_2$ → $00111001_2$ → ´9´

**Durch das Verfahren auf der letzten Folie wurden ja eigentlich erst einmal nur die binären Bitmuster der einzelnen Dezimalstellen 0 bis 9 berechnet, also die Bitmuster der Zahlenwerte 0 bis 9. Diese Zahlenwerte müssen dann noch in Zeichen für die einzelnen Ziffern 0 bis 9 kodiert werden. Das Prinzip sehen Sie jetzt hier.**

**Aus Platzgründen zeigen wir aber nur die acht unteren Bits an. obwohl das Ergebnis im Datentyp char und damit in 16 Bits gespeichert wird. Die oberen 8 Bits sind bei den Dezimalziffern alle gleich 0, so dass wir durch die Verkürzung auf die unteren 8 Bits nichts auslassen.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

$00000000_2$ → $00110000_2$ → ´0´
$00000001_2$ → $00110001_2$ → ´1´
$00000010_2$ → $00110010_2$ → ´2´
$00000011_2$ → $00110011_2$ → ´3´
$00000100_2$ → $00110100_2$ → ´4´
$00000101_2$ → $00110101_2$ → ´5´
$00000110_2$ → $00110110_2$ → ´6´
$00000111_2$ → $00110111_2$ → ´7´
$00001000_2$ → $00111000_2$ → ´8´
$00001001_2$ → $00111001_2$ → ´9´

**Die zehn Dezimalziffern sind in Unicode in aufsteigender Reihenfolge kodiert mit den Zahlenwerten 48 bis 57. Auf den Wert 0 bis 9 wird also der Wert 48 draufaddiert, um die Kodierung des entsprechenden Zeichens zu erhalten. Im Binärsystem heißt das einfach nur, dass die Sechzehner- und die Zweiunddreißigerstelle von 0 auf 1 gesetzt werden.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

| | | | |
|---|---|---|---|
| $00000000_2$ | → | $00110000_2$ | → ´0´ |
| $00000001_2$ | → | $00110001_2$ | → ´1´ |
| $00000010_2$ | → | $00110010_2$ | → ´2´ |
| $00000011_2$ | → | $00110011_2$ | → ´3´ |
| $00000100_2$ | → | $00110100_2$ | → ´4´ |
| $00000101_2$ | → | $00110101_2$ | → ´5´ |
| $00000110_2$ | → | $00110110_2$ | → ´6´ |
| $00000111_2$ | → | $00110111_2$ | → ´7´ |
| $00001000_2$ | → | $00111000_2$ | → ´8´ |
| $00001001_2$ | → | $00111001_2$ | → ´9´ |

**Es fehlt noch der Schritt zur Ausgabe auf dem Bildschirm oder dem Drucker. Für jede Zahl, die ein Zeichen repräsentiert, ist eine Graphik hinterlegt, die dann auf dem Bildschirm oder dem Drucker für diese Zahl ausgegeben wird.**

**Genauer gesagt, gibt es eine solche Tabelle von Graphiken für jede Schriftart, denn das Bild eines Zeichens ist natürlich bei jeder Schriftart ein anderes.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Von Dezimaldarstellung in Datenytyp:

"3856" →

$0...0011_2 * 0...1111101000_2 = 0...101110111000_2$

$0...1000_2 * 0...0001100100_2 = 0...001100100000_2$

$0...0101_2 * 0...0000001010_2 = 0...000000110010_2$

$0...0110_2 * 0...0000000001_2 = 0...000000000110_2$

→ $0...111100010000_2$

**Der umgekehrte Schritt, von der Darstellung einer Zahl durch Dezimalzeichen in einen ganzzahligen primitiven Datentyp, ist nun relativ einfach zu verstehen.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Von Dezimaldarstellung in Datenytyp:

**“3856“ →**

$0...0011_2 * 0...1111101000_2 = 0...101110111000_2$

$0...1000_2 * 0...0001100100_2 = 0...001100100000_2$

$0...0101_2 * 0...0000001010_2 = 0...000000110010_2$

$0...0110_2 * 0...0000000001_2 = 0...000000000110_2$

$\rightarrow 0...111100010000_2$

**Wir nehmen uns als Beispiel eine etwas größere Zahl her, die nicht mehr in den Datentyp byte, aber in jeden der anderen ganzzahligen primitiven Datentypen passt. Wir nehmen also an, dass der Zieltyp 16, 32 oder 64 Bit hat, also short, int oder long.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Von Dezimaldarstellung in Datenytyp:

"3856" →

$0...0011_2 * 0...1111101000_2 = 0...101110111000_2$

$0...1000_2 * 0...0001100100_2 = 0...001100100000_2$

$0...0101_2 * 0...0000001010_2 = 0...000000110010_2$

$0...0110_2 * 0...0000000001_2 = 0...000000000110_2$

→ $0...111100010000_2$

**Jedes dieser vier Zeichen ist im Datentyp char ja wieder eine Zahl, beginnend mit 48 für das Zeichen ´0´. So wir wir eben 48 draufaddiert haben, um die Zahlenwerte für die Zeichen zu bekommen, so ziehen wir von den Zahlenwerten für die Zeichen jeweils 48 ab, um die Ziffern als binäre Zahlen zu erhalten. Dafür werden die Zweiunddreißiger- nud die Sechzehnerstelle einfach von 1 auf 0 gesetzt.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Von Dezimaldarstellung in Datenytyp:

"3856" →

$0...0011_2 * 0...1111101000_2 = 0...101110111000_2$

$0...1000_2 * 0...0001100100_2 = 0...001100100000_2$

$0...0101_2 * 0...0000001010_2 = 0...000000110010_2$

$0...0110_2 * 0...0000000001_2 = 0...000000000110_2$

→ $0...111100010000_2$

**Jede Ziffer wird mit der zugehörigen Zehnerpotenz multipliziert, also die 3 mit der 1000, die 8 mit der 100, die 5 mit der 10 und die 6 mit der 1.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

**Von Dezimaldarstellung in Datenytyp:**

**“3856“ →**

$0...0011_2 * 0...1111101000_2 = 0...101110111000_2$

$0...1000_2 * 0...0001100100_2 = 0...001100100000_2$

$0...0101_2 * 0...0000001010_2 = 0...000000110010_2$

$0...0110_2 * 0...0000000001_2 = 0...000000000110_2$

**→** $0...111100010000_2$

**Das sind dann die vier Produkte: 3000, 800, 50 und 6.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Von Dezimaldarstellung in Datenytyp:

"3856" →

$0...0011_2 * 0...1111101000_2 = 0...101110111000_2$

$0...1000_2 * 0...0001100100_2 = 0...001100100000_2$

$0...0101_2 * 0...0000001010_2 = 0...000000110010_2$

$0...0110_2 * 0...0000000001_2 = 0...000000000110_2$

→ $0...111100010000_2$

**Und diese vier Produkte müssen dann nur noch aufsummiert werden, um die Binärdarstellung von 3856 zu erhalten.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Von Dezimaldarstellung in Datenytyp:

"3856" →

$0...0011_2 * 0...1111101000_2 = 0...101110111000_2$

$0...1000_2 * 0...0001100100_2 = 0...001100100000_2$

$0...0101_2 * 0...0000001010_2 = 0...000000110010_2$

$0...0110_2 * 0...0000000001_2 = 0...000000000110_2$

→ $0...111100010000_2$

**Insgesamt sehen wir wieder eine Schleife über Zehnerpotenzen, nämlich über so viele, wie die Zahl an Ziffern enthält.**

**Die Ergebnisse der einzelnen Schleifendurchläufe werden addiert, das kann wie hier angedeutet am Ende geschehen. Natürlich können die Zwischenergebnisse statt dessen in jedem Schleifendurchlauf auch gleich schrittweise auf das mit 0 initialisierte Ergebnis addiert werden.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Negative Zahl im 2-Komplement:

**+3856**

**→ $000...111100010000_2$**

**→ $111...000011101111_2$ + 1**

**→ $111...000011110000_2$**

**→ −3856**

**Was noch fehlt, ist die Darstellung von *negativen* Zahlen in ganzzahligen Datentypen. Java schreibt eine bestimmte Darstellungsweise vor, die auch auf eigentlich allen heute gängigen Hardwareplattformen die Darstellungsweise für negative ganze Zahlen ist. Der Fachbegriff lautet 2-Komplement.**

***Achtung*: Nicht in allen Programmiersprachen ist eine bestimmte Darstellung negativer ganzer Zahlen verbindlich vorgeschrieben.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Negative Zahl im 2-Komplement:

**+3856**

→ $000...111100010000_2$

→ $111...000011101111_2$ + 1

→ $111...000011110000_2$

→ −3856

**Wir nehmen wieder dieselbe Zahl wie eben her, 3856, und berechnen aus ihrer Darstellung beispielhaft die Darstellung ihres negativen Gegenstücks, der -3856.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

+3856

→ $000...111100010000_2$

→ $111...000011101111_2$ + 1

→ $111...000011110000_2$

→ −3856

**Das ist die soeben von uns berechnete Binärdarstellung von 3856. Wir machen jetzt zwei Schritte.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

+3856

→ $000...111100010000_2$

→ $111...000011101111_2$ + 1

→ $111...000011110000_2$

→ −3856

Im ersten Schritt drehen wir einfach jedes Bit um: Aus einer 1 wird eine 0, und aus einer 0 wird eine 1. Das Ergebnis heißt *1-Komplement*.

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**+3856**

**→ $000...111100010000_2$**

**→ $111...000011101111_2$** **+ 1**

**→ $111...000011110000_2$**

**→ −3856**

**Aus dem 1-Komplement erhält man das *2-Komplement*, indem man noch eine 1 draufaddiert.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**+3856**

**→ $000...111100010000_2$**

**→ $111...000011101111_2$ + 1**

**→ $111...000011110000_2$**

**→ −3856**

**Dadurch werden alle Einsen rechts von der rechtesten Null zu Nullen, und die erste Null wird zu einer Eins. In dem Grenzfall, dass das am wenigsten signifikante Bit des 1-Komplements eine Null ist, heißt das natürlich, dass dieses Bit zu einer Eins wird und sonst nichts passiert.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**+3856**

**→ $000...111100010000_2$**

**→ $111...000011101111_2$ + 1**

**→ $111...000011110000_2$**

**→ −3856**

**Das ist dann die Darstellung von -3856 im 2-Komplement.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**-3856**

**→ $111...0000111100000_2$**

**→ $000...1111000011111_2$ + 1**

**→ $000...1111000010000_2$**

**→ +3856**

**Wenden wir das Verfahren auf die so erhaltene negative Zahl nochmals an – berechnen wir also das 2-Komplement des 2-Komplements –, dann erhalten wir wieder das Bitmuster des positiven Gegenstücks. So soll es ja auch sein, zweimal Minus soll Plus ergeben.**

**Das ist auch logisch, dass das immer so ist und nicht nur vielleicht zufällig in diesem Beispiel, denn die Addition von +1 bei der Umwandlung von positiv nach negativ vermindert das Bitmuster einer negativen Zahl betragsmäßig um +1, bei der Umkehrung wird das Bitmuster einer positiven Zahl betragsmäßig um +1 erhöht.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**-3856**

**→ $111...0000111100000_2$**

**→ $000...1111000011111_2$ + 1**

**→ $000...1111000100000_2$**

**→ +3856**

**Das 2-Komplement hat gegenüber anderen möglichen Darstellungsweisen für negative Zahlen wie etwa das 1-Komplement zwei Vorteile:**

**Erstens gibt es nur eine Darstellung der 0, das 2-Komplement des Bitmusters identisch 0 ist wieder das Bitmuster identisch 0. Beim 1-Komplement gibt es hingegen *zwei* Darstellungen der 0: alle Bits auf 0 und alle Bits auf 1. Für den Test eines Wertes auf gleich 0 müsste man diesen Wert im 1-Komplement also immer mit *zwei* Bitmustern vergleichen, beim 2-Komplement nur mit *einem*.**

**Zweitens kann man eine Zahl von einer anderen abziehen, indem man den Subtrahenden ins 2-Komplement setzt und jetzt beide Zahlen einfach miteinander addiert. Mehr dazu in Lehrveranstaltungen zu Digitaltechnik und Rechnerorganisation.**

# Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Negative Zahl im 2-Komplement:

**Was ist mit dem Bitmuster 100...00?**

**100...00**

**011...11**

**100...00**

**Es gibt ein einziges Bitmuster mit negativem Vorzeichen, das sich nicht durch 2-Komplementbildung aus einer nichtnegativen Zahl berechnen lässt. Wenn wir im ersten Schritt das 1-Komplement von 100...00 bilden und dann durch Addition von 1 das 2-Komplement, dann erhalten wir dasselbe Bitmuster wieder. Das sehen Sie in den unteren drei Zeilen auf dieser Folie.**

**Bei einer negativen Zahl, die sich aus einer nichtnegativen Zahl durch 2-Komplement berechnen lässt, würden wir hingegen durch 2-Komplement auf dieser negativen Zahl wieder die ursprüngliche nichtnegative Zahl erhalten, wie wir eben gesehen haben.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**Was ist mit dem Bitmuster 100...00?**

| | |
|---|---|
| 011...11 | 000...01 |
| 100...00 | 111...10 |
| 100...01 | 111...11 |

**100...01 + 111...11 = 100...00**

**Um zu sehen, welche Zahl diesem Bitmuster entspricht, machen wir mehrere Gedankenschritte.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**Was ist mit dem Bitmuster 100...00?**

| | |
|---|---|
| 011...11 | 000...01 |
| 100...00 | 111...10 |
| 100...01 | 111...11 |

**100...01 + 111...11 = 100...00**

**Als erstes berechnen wir das negative Gegenstück zu der größten darstellbaren Zahl via 1-Komplement im 2-Komplement und stellen fest, dass das Bitmuster sich nur in der Einerstelle von unserer gesuchten Zahl unterscheidet.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**Was ist mit dem Bitmuster 100...00?**

| | |
|---|---|
| 011...11 | 000...01 |
| 100...00 | 111...10 |
| 100...01 | 111...11 |

**100...01 + 111...11 = 100...00**

**Als zweites berechnen wir das Bitmuster von -1 aus dem Bitmuster von +1, wieder via 1-Komplement im 2-Komplement. Die –1 wird also durch das Bitmuster identisch 1 dargestellt.**

## Ganze Zahlen: Zahldarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

### Negative Zahl im 2-Komplement:

**Was ist mit dem Bitmuster 100...00?**

| | |
|---|---|
| 011...11 | 000...01 |
| 100...00 | 111...10 |
| 100...01 | 111...11 |

**100...01 + 111...11 = 100...00**

**Wenn wir nun beide Werte miteinander addieren, dann ergibt sich an der letzten Stelle 0, Übertrag 1. Daraus ergibt sich an der vorletzten Stelle ebenfalls 0, Übertrag 1 und so weiter. Erst beim führenden Bit ergibt sich nicht 0, sondern 1, und wir erhalten unser Bitmuster.**

**Unser Bitmuster ist also 1 weniger als das negative Gegenstück der größten darstellbaren Zahl. Das erklärt die Asymmetrie bei den Bereichen der einzelnen primitiven Datentypen: Der negative Bereich ist jeweils um 1 größer als der positive Bereich, und der Grund dafür ist genau dieses eine Bitmuster.**

TECHNISCHE UNIVERSITÄT DARMSTADT

# Bitlogik

**Java bietet einige weitere Operatoren, die in gewissem Sinne arithmetisch genannt werden können. Diese sind eng verbunden mit dem Thema interne Zahldarstellung, so dass wir sie erst in diesem Kapitel betrachten.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean bitIsSet
                    ( int bitArray, int position ) {
  return bitArray & ( 1 << position ) != 0;
}
```

**In manchen, vor allem systemnahen Kontexten finden sich einzelne ganzzahlige Variable, die nicht die Aufgabe haben, einen Zahlenwert zu speichern, sondern die einzelnen Bits haben unterschiedliche Bedeutungen. Jedes Bit speichert dann eine binäre Information so wie eine boolesche Variable, nur kompakter. Mit dieser einfachen Methode bekommen wir heraus, ob ein bestimmtes Bit gesetzt ist oder nicht.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean bitIsSet
                    ( int bitArray, int position ) {
  return bitArray & ( 1 << position ) != 0;
}
```

**Der erste Parameter ist der ganzzahlige Wert, in dem jedes Bit eine spezifische binäre Information trägt. Solche Werte nennt man auch Bitarrays, daher der Name des Parameters. Wegen Datentyp int sind das in diesem Fall also 32 einzelne binäre Informationen.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean bitIsSet
                    ( int bitArray, int position ) {
  return bitArray & ( 1 << position ) != 0;
}
```

**Das ist die Nummer des Bits, dessen Wert wir auslesen wollen. Bei der Implementation der Methode gehen wir davon aus, dass der Parameter position ein Wert im Bereich 0 bis 31 ist und dass position gleich 0 die Einerstelle ist, so dass position gleich 31 das Vorzeichenbit ist.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean bitIsSet
                    ( int bitArray, int position ) {
  return bitArray & ( 1 << position ) != 0;
}
```

**Dieser Operator heißt Linksshift-Operator, und genau das tut er. Er verschiebt das Bitmuster des ersten Operanden um so viele Stellen nach links, wie der zweite Operand angibt. Von rechts wird mit Nullen aufgefüllt.**

**Der erste Operand hat eine 1 an Position 0 und sonst alles Nullen, der Linksshift ergibt also eine 1 an der angegebenen Position, und alle Bits links davon sind gleich 0. Wegen der Auffüllung von rechts mit Nullen sind auch alle anderen Bits gleich 0.**

**Wir sagen, dass wir damit eine *Maske* für die angebebene Position gebildet haben.**

**Natürlich gibt es analog einen Rechtsshift-Operator, der spiegelsymmetrisch exakt dasselbe tut und durch zwei Größer-Zeichen dargestellt wird.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean bitIsSet
                    ( int bitArray, int position ) {
  return bitArray & ( 1 << position ) != 0;
}
```

**Mit diesem Operator werden zwei ganzzahlige Operanden zu einem Ergebnis desselben Datentyps verknüpft, und zwar durch bitweise Verundung: An jeder einzelnen Position kommt genau dann 1 heraus, wenn an dieser Position bei beiden Operanden jeweils eine 1 steht.**

# Ganze Zahlen: Bitlogik

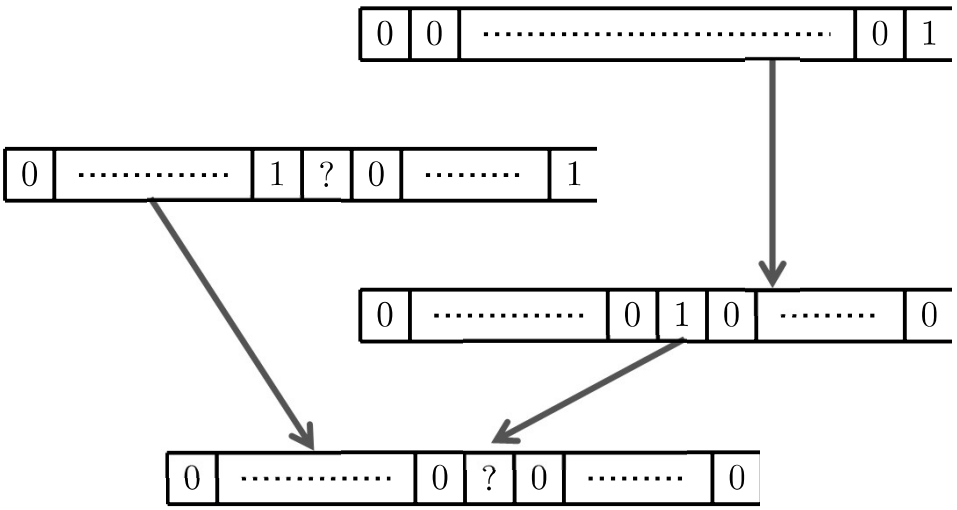
TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static boolean bitIsSet
                    ( int bitArray, int position ) {
  return bitArray & ( 1 << position ) != 0;
}
```

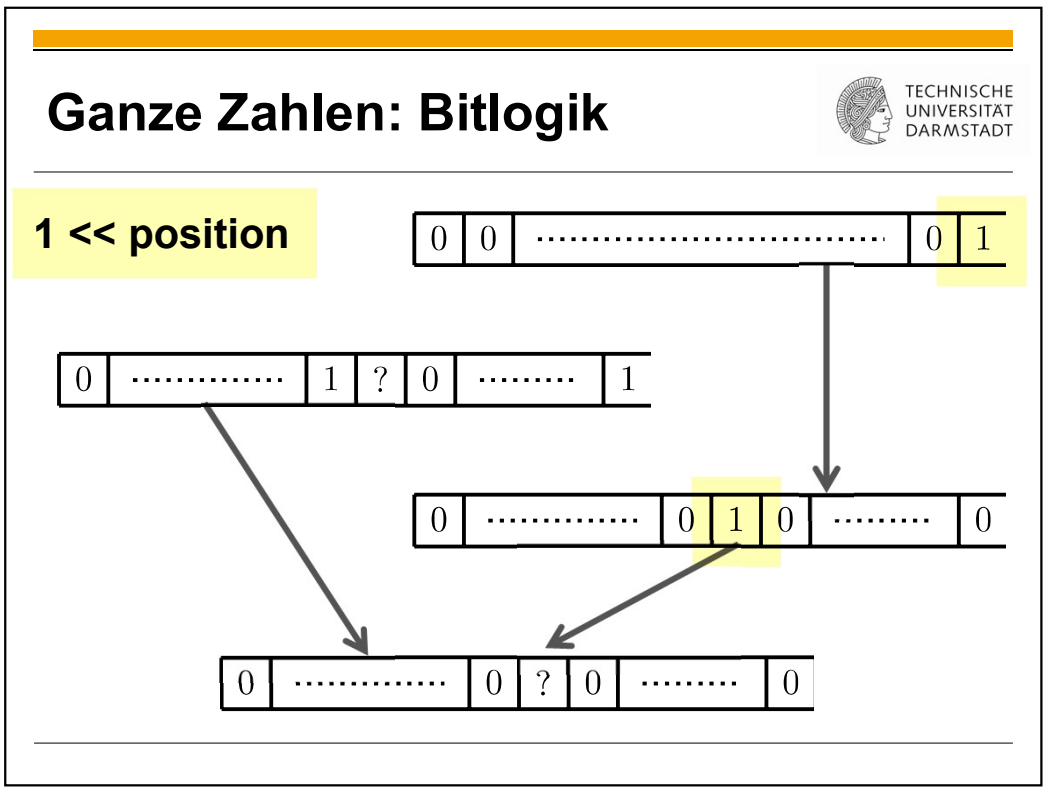
**An der angegebenen Position ist das Ergebnis genau dann 1, wenn dort im Bitarray eine 1 steht. An allen anderen Positionen ist das Ergebnis 0, da ja die Maske an allen anderen Positionen eine 0 hat. Das Ergebnis ist als Zahlenwert also genau dann ungleich 0, wenn das Bit an der angegebenen Position im Ergebnis und somit im Bitarray ungleich 0 ist.**

# Ganze Zahlen: Bitlogik

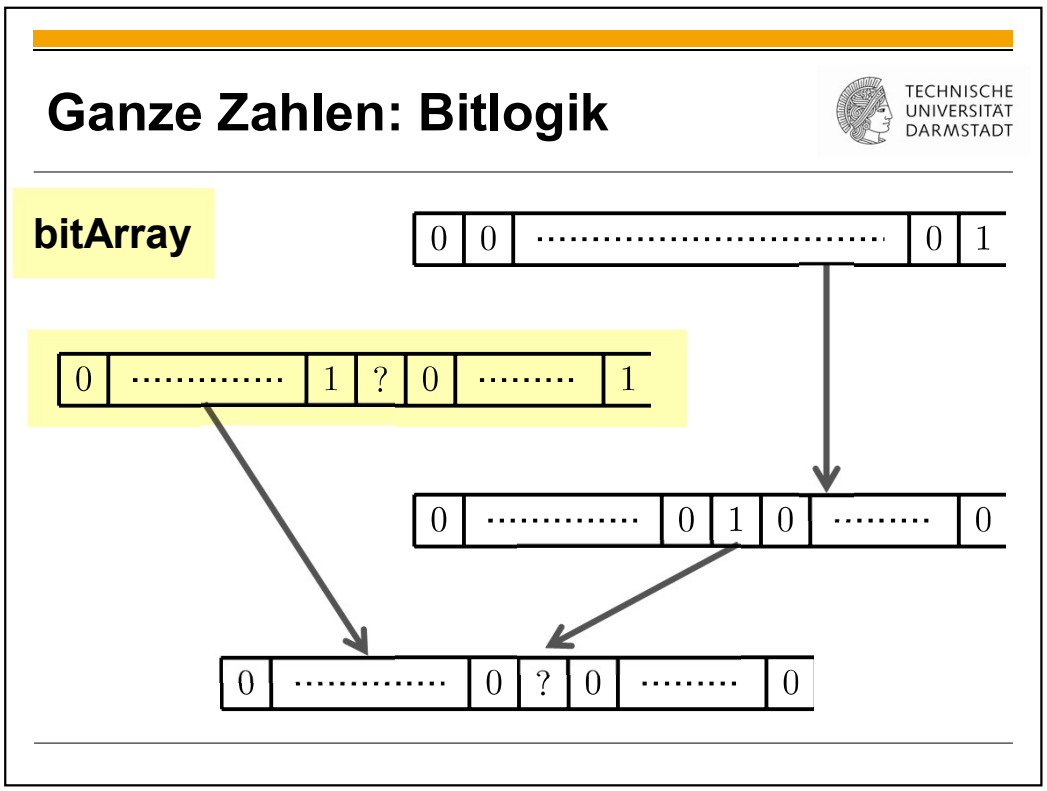
TECHNISCHE UNIVERSITÄT DARMSTADT



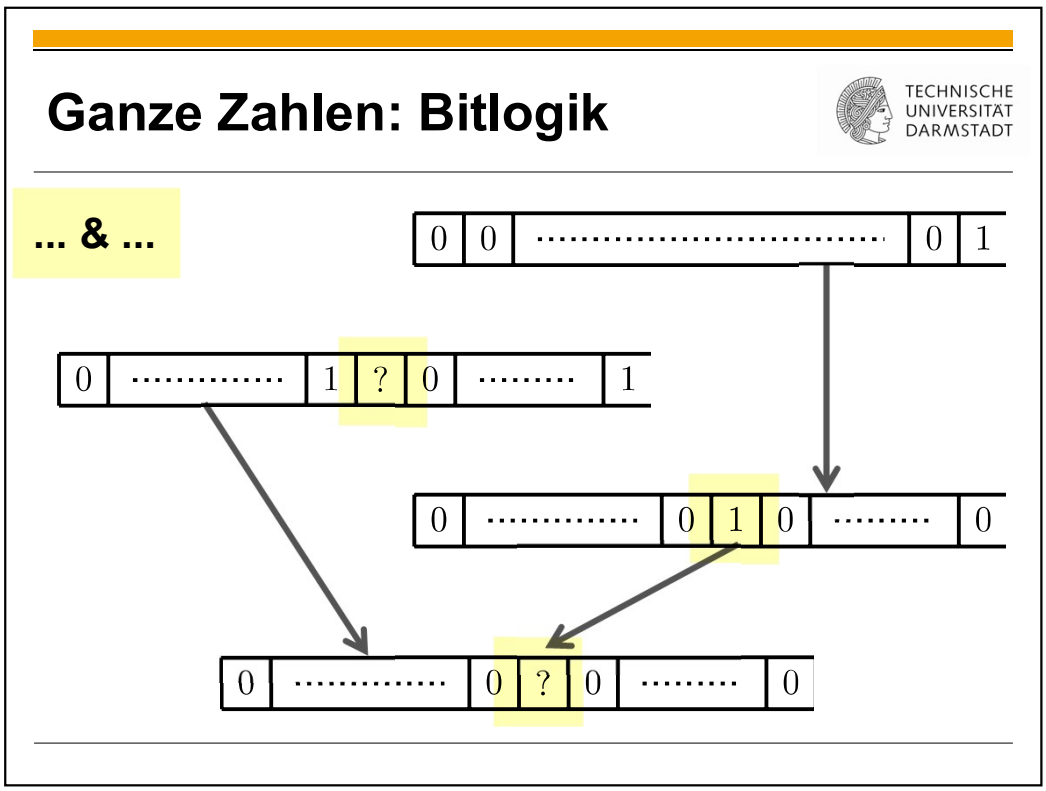

**So kann man sich das Ganze bildlich vorstellen.**




**Hier wird des Linksshift-Operator angewandt, um die 1 vom der Einerstelle an die angegebene Position zu verschieben.**




**Das Bitarray hat an den einzelnen Positionen Nullen und Einsen. An der angegebenen Position steht natürlich ebenfalls eine 0 oder 1, das ist durch das Fragezeichen offengelassen.**




**Durch die bitweise Verundung mit 0..010..0 hat das Bit an dieser Position denselben Wert wie im Bitarray, und die Ergebnisbits an allen anderen Positionen sind durch die Verundung mit 0 gleich 0. Das Gesamtergebnis ist somit genau dann gleich 0, wenn das Bit an der fraglichen Position im eingegebenen Bitarray gleich 0 ist.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int setBit
                    ( int bitArray, int position ) {
  return bitArray | ( 1 << position );
}
```

**Wir haben gesehen, wie man ein einzelnes Bit an einer bestimmten Position in einem ganzzahligen Wert *lesen* kann. Jetzt schauen wir uns noch an, wie man ein einzelnes Bit auf 1 beziehungsweise 0 *setzen* kann, zuerst auf 1. Die hier gezeigte Methode liefert das Bitarray wieder zurück mit dem einzigen Unterschied, dass an der angegebenen Position auf jeden Fall eine 1 steht, egal ob in bitArray dort eine 1 oder eine 0 steht.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int setBit
                    ( int bitArray, int position ) {
  return bitArray | ( 1 << position );
}
```

**Wieder die Maske für die angegebene Position.**

# Ganze Zahlen: Bitlogik

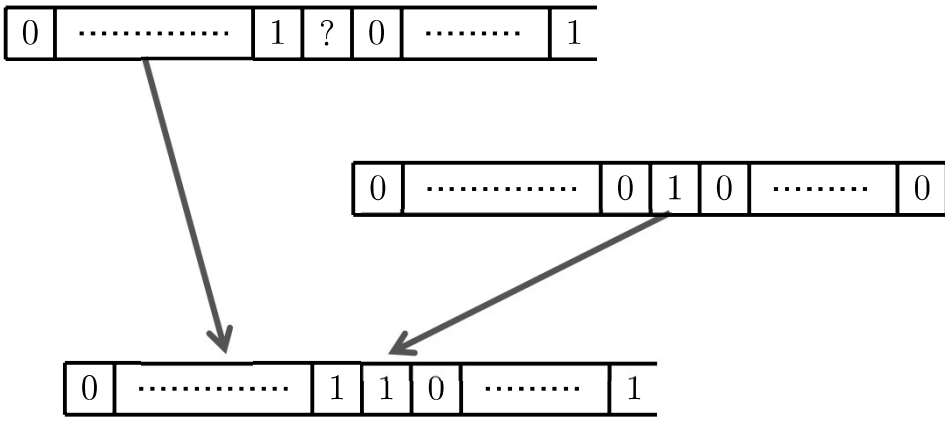
TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int setBit
                    ( int bitArray, int position ) {
  return bitArray | ( 1 << position );
}
```

**Auch der einzelne senkrechte Strich ist ein Bitoperator, nämlich bitweise Veroderung mit Inklusiv-Oder. Das heißt, ein Bit im Ergebnis ist gleich 1, wenn mindestens einer der beiden Operanden an dieser Position eine 1 hat. Der Rückgabewert hat also alle Einsen vom Bitarray, und zusätzlich werden die Bits an allen Positionen auf 1 gesetzt, an denen zweite Operand eine 1 hat. Letzteres ist aber nur an einer einzigen Position der Fall, nämlich an der angegebenen. Zusammengefasst ist also wie gewünscht der einzige Unterschied zum Bitarray, dass an der angegebenen Position jetzt auf jeden Fall eine 1 steht, egal ob im eingegebenen Bitarray hier eine 1 oder 0 steht.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT




**So sieht das Ganze wieder an einem schematischen Beispiel aus: Wo im Bitarray noch ein Fragezeichen steht, um anzudeuten, dass hier eine 0 oder 1 stehen kann, steht im Ergebnis eine 1. Alle anderen Bits werden durch Veroderung mit 0 eins-zu-eins übernommen.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int unsetBit
                    ( int bitArray, int position ) {
  return bitArray & ~ ( 1 << position );
}
```

**Diese Methode liefert ebenfalls das Bitarray wieder zurück, nur dass jetzt an der angegebenen Position auf jeden Fall eine 0 steht, egal ob im Bitarray dort eine 0 oder eine 1 steht.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int unsetBit
                    ( int bitArray, int position ) {
  return bitArray & ~ ( 1 << position );
}
```

**Auch hier wieder die Maske für die angegebene Position.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int unsetBit
                    ( int bitArray, int position ) {
  return bitArray & ~ ( 1 << position );
}
```

**Allerdings mit einer Modifikation: Dieser Operator, englisch Tilde genannt, ist unär und gibt das 1-Komplement seines Operanden zurück. Das 1-Komplement haben wir schon bei negativen ganzen Zahlen gesehen: Jedes Bit wird einfach umgekehrt. Konkret hier heißt das, an der angegebenen Position steht eine 0 und überall sonst steht eine 1.**

# Ganze Zahlen: Bitlogik

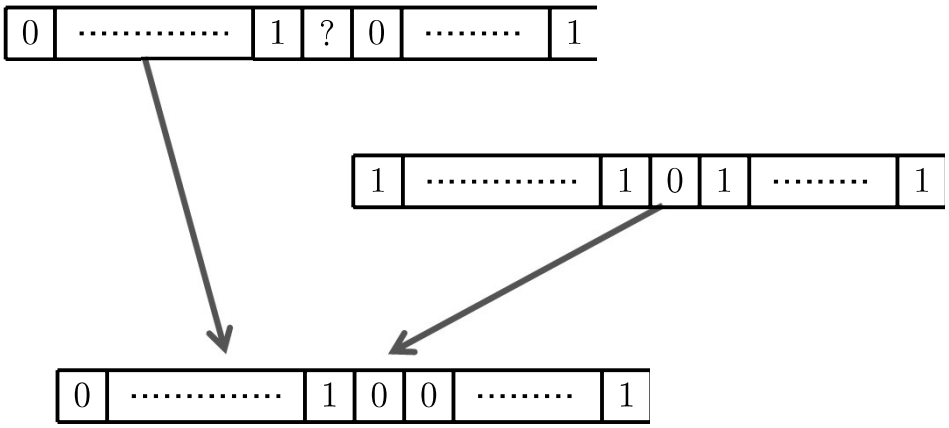
TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int unsetBit
                    ( int bitArray, int position ) {
  return bitArray & ~ ( 1 << position );
}
```

**Die bitweise Verundung haben wir eben schon gesehen. Hier hat sie zur Konsequenz, dass das Bit an der angegebenen Position mit 0 verundet wird, also Ergebnis 0, während die Bits an den anderen Positionen mit 1 verundet werden, also so bleiben, wie sie im Bitarray sind.**

# Ganze Zahlen: Bitlogik

TECHNISCHE UNIVERSITÄT DARMSTADT




**So sieht das Ganze wieder an einem schematischen Beispiel aus: Wo im Bitarray noch ein Fragezeichen steht, um anzudeuten, dass hier eine 0 oder 1 stehen kann, steht im Ergebnis eine 0.**

**Es gibt natürlich noch viel mehr zu Bitlogik in Java und darüber hinaus zu sagen, aber Sie sollten durch diesen Abschnitt ausreichendes Verständnis erworben haben, um sich weitere Inhalte selbst anzueignen.**

TECHNISCHE UNIVERSITÄT DARMSTADT

# Gebrochene Zahlen

**Nun kommen wir zur zweiten Art von zahlenwertigen primitiven Datentypen in Java und vielen anderen Sprachen.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

| | float | double |
|---|---|---|
| Bits | 32 | 64 |
| Max. Wert | $\pm$ 3.402... $\cdot$ $10^{38}$ | $\pm$ 1.797... $\cdot$ $10^{308}$ |
| Genauigkeit | 1 zu 8.388.608 | ca. 1 zu 4,5 Billiarden |

***Erinnerung*: Diese Aufstellung haben wir in Kapitel 01b, Abschnitt zu primitiven Datentypen, schon einmal gesehen und diskutiert.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**Probleme mit Ungenauigkeiten (Beispiele):**

- **Umkehrrechnungen liefern nicht genau den Ausgangswert**
- **Bei Addition extrem unterschiedlich großer Zahlen geht die kleinere unter**
- **Bei Subtraktion fast gleich großer Zahlen bleiben mglw. nur inkorrekte Bits**
- **Test auf Gleichheit muss ersetzt werden durch Test auf „ausreichend nahe beieinander“**

**Auch diese Problematiken waren wir dort schon einmal durchgegangen, so dass wir das hier nicht noch einmal tun müssen.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

+3.14159E17
–3.14159E17

**Bestandteile des Literals:**

- Vorzeichen
- Basis: 10
- Mantisse: 3.14159
- Exponent: 17

**Wir schauen uns jetzt genauer an, wie gebrochene Zahlen in den beiden Datentypen float und double intern dargestellt werden. Das ist ziemlich analog zu Literalen in wissenschaftlicher Notation, daher beginnen wir damit. Die interne Darstellnug ist natürlich binär, nicht wie bei Literalen dezimal.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

+3.14159E17
–3.14159E17

Bestandteile des Literals:

- Vorzeichen
- Basis: 10
- Mantisse: 3.14159
- Exponent: 17

**Das Vorzeichen ist eine binäre Information, die intern in einem einzelnen Bit abgespeichert wird.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

+3.14159E17
–3.14159E17

**Bestandteile des Literals:**

- Vorzeichen
- Basis: 10
- Mantisse: 3.14159
- Exponent: 17

**Bei Zahldarstellungen für menschliche Leser wie auf der linken Seite ist die Basis in der Regel 10, intern ist sie natürlich 2.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

+3.14159E17
–3.14159E17

Bestandteile des Literals:

- Vorzeichen
- Basis: 10
- Mantisse: 3.14159
- Exponent: 17

**Die Mantisse, das ist in Gleitkommadarstellung die Zahl, die mit einer Potenz der Basis multipliziert wird, um die eigentlich dargestellte Zahl zu erhalten.**

**Man sollte allerdings dazusagen, dass der Begriff „Mantisse“ nicht einheitlich definiert ist. Es finden sich auch Verwendungen dieses Begriffs, die von der Verwendung auf dieser Folie in kleinen Details abweichen.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

+3.14159E17
–3.14159E17

**Bestandteile des Literals:**

- Vorzeichen
- Basis: 10
- Mantisse: 3.14159
- Exponent: 17

**Der Exponent sagt dann aus, mit welcher Potenz der Basis die Mantisse zu multiplizieren ist.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

- **Vorzeichen: 1 Bit**
- **1 bedeutet negativ**
- **Mantisse und Exponent in normalisierter Binärdarstellung**

**Was heißt das jetzt genau für die Datentypen float und double?**

# Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

- **Vorzeichen: 1 Bit**
- **1 bedeutet negativ**
- **Mantisse und Exponent in normalisierter Binärdarstellung**

**Auch im Deutschen wird diese Abkürzung Englisch ausgesprochen, also „ai-trippl-ih“. Das ist die weltweite Vereinigung von Elektrotechnikern und Informationstechnikern. Eine der Kernaufgaben dieser Vereinigung ist Standardisierung.**

# Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

- Vorzeichen: 1 Bit
- 1 bedeutet negativ
- Mantisse und Exponent in normalisierter Binärdarstellung

**Im Standard Nr. 754 wird die binäre Darstellung von Gleitkommazahlen festgelegt. Bei der Entwicklung von Java wurde festgelegt, dass float und double immer nach diesem Standard implementiert sein müssen. Manche Programmiersprachen wie C und C++ geben die Implementation nicht vor und erlauben damit hardwarekonforme Implementationen auf Hardwareplattformen, die diesen Standard nicht einhalten.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

IEEE 754:

- Vorzeichen: 1 Bit
- 1 bedeutet negativ
- Mantisse und Exponent in normalisierter Binärdarstellung

Wie gesagt, wird das Vorzeichen in einem einzelnen Bit gespeichert. Wie bei ganzen Zahlen gibt die 1 an, dass die Zahl negativ ist.

Im Gegensatz zu ganzzahligen Datentypen gibt es bei gebrochenzahligen Datentypen also eine positive und eine negative Null. Bei Tests auf gleich 0 werden beide Werte getestet, was natürllich sehr einfach geht: einfach das Vorzeichen beim Test nicht mittesten.

*Erinnerung*: In Kapitel 04a hatten wir im Abschnitt zu Rekursion in Java anhand des Bisektionsverfahren und dann noch einmal später in 04a bei check-within diskutiert, dass Tests auf exakte Wertgleichheit von gebrochenen Zahlen ohnehin in den meisten Fällen hoch problematisch ist.

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

IEEE 754:

- Vorzeichen: 1 Bit
- 1 bedeutet negativ
- Mantisse und Exponent in normalisierter Binärdarstellung

Was es mit dieser normalisierten Binärdarstellung auf sich hat, sehen wir uns auf den nächsten Folien an.

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

IEEE 754:

| | float | double |
|---|---|---|
| Mantisse | 23 Bits | 52 Bits |
| Exponent | 8 Bits | 11 Bits |

**Das ist zunächst einmal die Aufteilung der Bits in float und double für die Mantisse und den Exponenten. Nimmt man noch das eine Bit für das Vorzeichen hinzu, dann kommt man auf die insgesamt 32 beziehungsweise 64 Bit für float und double.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E7**

**→ 31415900**

**→ 11101111101011110010111100**

**→ $1.11011111010111100101111 00 * (10)^{11000}$**

**Wir schauen uns jetzt für zwei verschiedene dezimale Gleitkommazahlen beispielhaft an, wie sie in float und double umgewandelt werden.**

**Als erstes eine relativ große, aber nicht allzu große Zahl, bei der wir aber schon die Begrenztheit von float sehen werden.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E7**

**→ 31415900**

**→ 11101111101011110010111100**

**→ $1.11011111010111100101111 00 * (10)^{11000}$**

**So wird der Umrechnungsalgorithmus wahrscheinlich nicht vorgehen, aber hier geht es ja um prinzipielles Verständnis der Zahldarstellung, nicht um technische Realisierungen.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E7**

**→ 31415900**

**→ 11101111101011110010111100**

**→ $1.11011111010111100101111 00 * (10)^{11000}$**

**In Binärdarstelllung ist das diese Zahl.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E7**

**→ 31415900**

**→ 11101111101011110010111100**

**→ $1.11011111010111100101111 00 * (10)^{11000}$**

**Und so sieht diese Binärzahl aus, wenn wir zu normalisierter Gleitpunktdarstellung übergehen, das heißt, den Dezimalpunkt hinter die erste Stelle verschieben und den Exponenten entsprechend anpassen.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E7**

**→ 31415900**

**→ 11101111101011110010111100**

**→ 1.11011111010111100101111 00 * (10)$^{11000}$**

**Die Basis ist dann 2, also Eins-Null im Binärsystem.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E7**

**→ 31415900**

**→ 11101111101011110010111100**

**→ 1.11011111010111100101111 00 * (10)$^{11000}$**

**Und der Dezimalpunkt wurde um 24 Stellen verschoben, die 24 hat diese Darstellung im Binärsystem.**

# Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: Mantisse**

**$1.11011111010111100101110 0 * (10)^{11000}$**

**→ 11011111010111100101110**

**→ 11011111010111100101110...0**

**Oben ist die normalisierte binäre Gleitpunktdarstellung von der letzten Folie übernommen. Als erstes leiten wir daraus die Darstellung der Mantisse in float und double ab.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: Mantisse**

$1.11011111010111100101110 0 * (10)^{11000}$

→ 11011111010111100101110

→ 11011111010111100101110...0

**Wir sehen, dass die in Datentyp float gespeicherte Mantisse dasselbe Bitmuster ist, nur die 1 vor dem Dezimalpunkt und das letzte Bit fehlen.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

### IEEE 754: Mantisse

1.11011111010111100101110 0 * (10)$^{11000}$

→ 11011111010111100101110

→ 11011111010111100101110...0

Vor dem Dezimalpunkt steht immer eine 1, daher enthält diese Stelle im Grunde keine Information. Also wird die führende 1 einfach weggelassen und muss dann später bei jedem Verarbeitungsschritt wieder hinzugedacht werden.

Durch diesen kleinen Trick wird ein Bit eingespart. Oder anders gesagt: Mit den 23 beziehungsweise 52 Bit kann man ein Bit mehr darstellen, die Genauigkeit der Mantisse hat sich verdoppelt.

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

### IEEE 754: Mantisse

$1.11011111010111100101110\mathbf{0} * (10)^{11000}$

→ 11011111010111100101110

→ 11011111010111001011100...0

**Die Zahl hat ja 24 Stellen hinter dem Dezimalpunkt, aber für die Mantisse sind im Datentyp float nur 23 Stellen reserviert. Daher muss das letzte Bit leider abgeschnitten werden.**

**In diesem Beispiel ist das abgeschnittene Bit eine 0, und somit wird die Zahl weiterhin exakt dargestellt. Wäre die letzte Stelle eine 1 gewesen, dann wäre die Darstellung nicht mehr ganz exakt. Je größer die Zahl ist, um so mehr Bits müssen natürlich abgeschnitten werden, und um so weniger exakt wird die Darstellung, wenn die abgeschnittenen Bits Einsen sind.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

### IEEE 754: Mantisse

$1.11011111010111100101110 0 * (10)^{11000}$

→ 11011111010111100101110

→ 11011111010111100101110...0

**Im Datentyp double wird ebenfalls die führende 1 abgeschnitten, ebenfalls um ein Bit einzusparen. Aber es ist nicht notwendig, das letzte Bit abzuschneiden, da double nicht wie float 23, sondern 52 Bits für die Mantisse reserviert. Die weiteren Bits der Mantisse werden dann mit Nullen aufgefüllt. Hätte die Mantisse weniger als 23 Bits gehabt, dann wäre auch bei float nichts abgeschnitten, sondern stattdessen mit Nullen aufgefüllt worden.**

# Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

## IEEE 754: Exponent

$1.10111110101111001011100 * (10)^{11000}$

→ 11000 $+127_{10}$ / $+1023_{10}$

→ 10010111 / 10000010111

**Nun müssen wir noch die Darstellung des Exponenten ableiten. Wieder finden Sie oben die normalisierte Gleitpunktdarstellung, mit der wir die ganze Zeit gearbeitet haben.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: Exponent**

$1.10111110101111001011100 * (10)^{11000}$

→ 11000 $+127_{10}$ / $+1023_{10}$

→ 10010111 / 10000010111

**Der Exponent in diesem Beispiel war die 24.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

### IEEE 754: Exponent

$1.10111110101111001011100 * (10)^{11000}$

→ 11000 $+127_{10}$ / $+1023_{10}$

→ 10010111 / 10000010111

**Auf den Exponenten wird dann bei float und bei double jeweils eine feste Zahl addiert. Diese Zahl ergibt sich aus der Anzahl Bits für den Exponenten. Das waren ja 8 Bits bei float und 11 Bits bei double. Davon wird 1 abgezogen, also 7 und 10. Dann wird 2 hoch dieser Zahl genommen und noch 1 abgezogen.**

**Der Sinn dahinter ist, dass der Exponent immer als nichtnegative Zahl abgespeichert wird. Das werden wir gleich beim zweiten Zahlenbeispiel besser verstehen, bei dem der ursprüngliche Exponent negativ ist.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

### IEEE 754: Exponent

$1.10111110101111001011100 * (10)^{11000}$

→ 11000 $+127_{10}$ / $+1023_{10}$

→ 10010111 / 10000010111

**Und das sind die beiden resultierenden Exponenten, wie sie dann tatsächlich in float beziehungsweise double gespeichert werden. Durch die Addition mit 127 beziehungsweise 1023 ergibt sich, dass das erste Bit 1 ist, wenn der ursprüngliche Exponent positiv war.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E–3**

**→ 0.00314159**

**→0.00000001011101101...**

**→1.011101101... * $(10)^{-1001}$**

**Nun zum zweiten Beispiel, diesmal mit negativem Exponenten.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E–3**

**→ 0.00314159**

**→0.0000000010111011 01...**

**→1.011101101... * $(10)^{-1001}$**

**Die allermeisten gebrochenen Zahlen, die im Dezimalsystem exakt darstellbar sind, sind im Binärsystem *nicht* exakt darstellbar, so auch diese.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754:**

**3.14159E–3**

**→ 0.00314159**

**→0.00000001011101101...**

**→1.011101101... * $(10)^{-1001}$**

**Verschiebung des Dezimalpunktes um 9 Stellen nach rechts ergibt Exponent -9.**

# Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: Mantisse**

**$1.011101101... * (10)^{-1001}$**

**→ 011101101...**

**Die Mantisse wird wieder analog gebildet: Die führende 1 wird entfernt, und nach 23 beziehungsweise 52 Bits werden die weiteren Bits abgeschnitten, oder die Bits werden mit Nullen aufgefüllt, je nachdem.**

**In diesem Beispiel werden bei beiden Datentypen Bits abgeschnitten, da die exakte Darstellung unendlich lang ist und dementsprechend nicht in die 23 oder 52 Bits passt.**

# Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: Exponent**

**$1.011101101... * (10)^{-1001}$**

**→ –1001 $+127_{10}$ / $+1023_{10}$**

**→ 01110110 / 01111110110**

**Nun noch der Exponent. Oben ist wieder die ursprüngliche binäre Gleitpunktdarstellung der zweiten betrachteten Zahl.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: Exponent**

**$1.011101101... * (10)^{-1001}$**

**→ –1001** **$+127_{10}$ / $+1023_{10}$**

**→ 01110110 / 01111110110**

**Wieder wird 127 beziehungsweise 1023 auf den Exponent draufaddiert.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: Exponent**

**$1.011101101... * (10)^{-1001}$**

**→ –1001 $+127_{10}$ / $+1023_{10}$**

**→ 01110110 / 01111110110**

**Und das sind die resultierenden Bitmuster für den Exponenten. Bei nichtpositiven ursprünglichen Exponenten ist das Ergebnis siebenstellig beziehungsweise zehnstellig und muss daher mit einer Null vorne aufgefüllt werden.**

**Wir hatten schon festgestellt, dass bei positiven ursprünglichen Exponenten eine 1 ganz vorne steht; jetzt sehen wir, dass bei nichtpositiven Exponenten eine 0 vorne steht.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: „unendlich“ und NaN**

- **Der höchstmögliche Exponent 11...11 ist reserviert für zwei Spezialfälle**
- **Bei Mantisse 0: ∞**
  - **Je nach Vorzeichen +∞ oder –∞**
- **Bei anderer Mantisse: NaN („not a number“)**

**Die Datentypen float und double bieten noch einen Service in Form von zwei speziellen Werten, die hin und wieder ganz nützlich sind.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

IEEE 754: „unendlich“ und NaN

- Der höchstmögliche Exponent 11...11 ist reserviert für zwei Spezialfälle
- Bei Mantisse 0: ∞
  - Je nach Vorzeichen +∞ oder –∞
- Bei anderer Mantisse: NaN („not a number“)

Der Wert unendlich tritt üblicherweise bei Division durch 0 auf. Bei gebrochenen Zahlen ist das also kein Fehlerfall, sondern er führt zu einem speziellen Ergebnis.

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

IEEE 754: „unendlich“ und NaN

- Der höchstmögliche Exponent 11...11 ist reserviert für zwei Spezialfälle
- Bei Mantisse 0: ∞
  - Je nach Vorzeichen +∞ oder –∞
- Bei anderer Mantisse: NaN („not a number“)

**Der andere Wert tritt bei mathematischen Operationen mit unbestimmtem Ergebnis auf, zum Beispiel 0 geteilt durch 0 oder unendlich minus unendlich. Daher der Name NaN für not a number.**

***Erinnerung* an Kapitel 01b, Abschnitt zu primitiven Datentypen: In den Wrapper-Klassen Float und Double finden Sie diese Werte in den Klassenkonstanten POSITIVE_INFINITY, NEGATIVE_INFINITY und NaN.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

IEEE 754: „unendlich“ und NaN

- Der höchstmögliche Exponent 11...11 ist reserviert für zwei Spezialfälle
- Bei Mantisse 0: ∞
  - Je nach Vorzeichen +∞ oder –∞
- Bei anderer Mantisse: NaN („not a number“)

Dass einer dieser beiden Fälle auftritt, zeigt sich im Bitmuster darin, dass der Exponent seinen höchstmöglichen Wert annimmt, also acht beziehungsweise elf Einsen.

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

IEEE 754: „unendlich“ und NaN

- Der höchstmögliche Exponent 11...11 ist reserviert für zwei Spezialfälle
- Bei Mantisse 0: ∞
  - ➢ Je nach Vorzeichen +∞ oder –∞
- Bei anderer Mantisse: NaN („not a number“)

Die Unterscheidung zwischen den beiden Werten findet sich in der Mantisse: Sind alle für die Mantisse reservierten Bits gleich 0, dann ist der Wert als unendlich zu verstehen, sonst als NaN. NaN ist zwar abstrakt gesehen ein einzelner Wert, eigentlich nicht einmal wirklich ein Wert, aber nicht kodiert als ein einzelnes Bitmuster, sondern viele Bitmuster stehen für NaN.

# Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

IEEE 754: „unendlich“ und NaN

- Der höchstmögliche Exponent 11...11 ist reserviert für zwei Spezialfälle
- Bei Mantisse 0: ∞
  - ➢ Je nach Vorzeichen +∞ oder –∞
- Bei anderer Mantisse: NaN („not a number“)

Das Vorzeichenbit ist auch bei unendlich signifikant.

# Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: denormalisierte Zahlen**

- **Der kleinstmögliche Exponent 00...00 ist für Zahlen ≠0 reserviert, die eigentlich zu klein für normalisierte Darstellung sind**
- **In diesem Fall wird keine 1 vor der Mantisse hinzugerechnet**
- **Nicht auf jeder Hardware implementiert**
- **Bei manchen zu-/abschaltbar**

**Ein kleines Detail soll nicht unerwähnt bleiben, eine Abweichung von der bis hierher diskutierten Zahldarstellung für gebrochene Zahlen in bestimmten Fällen.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: denormalisierte Zahlen**

- **Der kleinstmögliche Exponent 00...00 ist für Zahlen ≠0 reserviert, die eigentlich zu klein für normalisierte Darstellung sind**
- **In diesem Fall wird keine 1 vor der Mantisse hinzugerechnet**
- **Nicht auf jeder Hardware implementiert**
- **Bei manchen zu-/abschaltbar**

**Wenn der Exponent identisch 0 ist, dann wird sich keine 1 vor der Mantisse hinzugedacht, so dass noch einmal halb so große Zahlen wie sonst darstellbar sind.**

## Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

**IEEE 754: denormalisierte Zahlen**

- **Der kleinstmögliche Exponent 00...00 ist für Zahlen ≠0 reserviert, die eigentlich zu klein für normalisierte Darstellung sind**
- **In diesem Fall wird keine 1 vor der Mantisse hinzugerechnet**
- **Nicht auf jeder Hardware implementiert**
- **Bei manchen zu-/abschaltbar**

**Dazu muss aber gesagt werden, dass diese Denormalisierung nicht einheitlich überall realisiert ist. Bei hardwarenäheren Sprachen als Java kann man das auf manchen Plattformen auch steuern.**