4.1.1 Der Datentyp boolean

- B... ...rte können in Variablen ges...

```
boolean b = true;
boolean test = false;
```

- Sie entstehen bei Vergleichen und können dann zugewiesen w...

```
b = 3 < 7;
test = x != y;
```

- Sie werden vor allem dort benutzt, wo Bedingungen ausgewertet werden müssen, z.B. in if-Anweisungen, while-Schleifen etc.:

```
if (var1 < var2) ...
while (x != y) ...
```

```
if (b) ...
while (test) ...
```

Boolesche Operationen in Java

- Für die Verknüpfung Boolescher Werte gibt es in Java verschiedene Operatoren:

```
logisches und (and):   & und &&
Logisches oder (or):   | und ||
Exklusives oder (xor):       ^
Logische Negation (not): !
```

verkürzete Auswertung

keine verkürzte Auswertung

- Beispiele:

linker Operand

```
boolean b = true;
boolean test =
b =&  ! ( x ! =
 fa$
test = (x != y) || (b ^ test);
```

Operation

rechter Operand

- & und | tragen die normale Bedeutung von and bzw. or. Sie werden „unbedingt“ ausgewertet, d.h.
  - Zuerst wird der linke und dann der rechte Operand ausgewertet.
  - Danach wird das Ergebnis laut Operationstabelle bestimmt.
- Dagegen werden && und || zur sog. verkürzten Auswertung verwendet.

Verkürzte Auswertung (1)

- Bei der Auswertung der binären Operatoren && und || gilt ebenfalls, dass zunächst der linke Operand ausgewertet wird.
  - Was können wir bereits aussagen, wenn das Ergebnis des ersten Operanden true bzw. false ist?

- Für Boolesche Werte gilt immer (d.h. für alle x ∈ {true,false}):

```
true  or x ergibt true
false and x ergibt
false
```

- Dies eröffnet die Möglichkeit, Boolesche Ausdrücke mit den Operatoren `&&` und `||` besonders effizient auszuwerten.

Verkürzte Auswertung (2)

- Auswertungsregel für **b1 || b2:**
  - Falls **b1** den Wert **true** liefert, muss **b2** nicht mehr ausgewertet werden, denn das Gesamtergebnis **true** steht bereits fest. Es gilt also:
  - `b = (b1 || b2)` ⇔ `if (b1) b = true; else b = b2;`
- Analog für **b1 && b2:**
  - Falls **b1** den Wert **false** liefert, ist das Ergebnis
  - `b = (b1 && b2)` ⇔ `if (!b1) b = false; else b = b2;`
- Damit ist auch

```
if ((x != 0) && ( y/x > 4))...
```

„sicher", da eine Division durch 0 vermieden wird.

Prioritäten

- Operatoren besitzen Prioritäten, welche die Reihenfolge der Auswertung bestimmen.
  - Aus der Schule bereits bekannt: Punkt- vor Strichrechnung.
  - Boolesche Operationen besitzen auch unterschiedliche Prioritäten

niedrige Priorität	Zuweisungsoperator:	=
	logisches Oder (verkürzt)	||
	logisches Und (verkürzt)	&&
	logisches Oder	|
	logisches Und	&
hohe Priorität	Vergleichsoperatoren	<, <=, ==, !=, =>, >
	logische Negation	!

- Die Reihenfolge der Auswertung kann durch Klammerung verändert werden.

```
boolean x = a < b && (c < d || e < f);
```

4.1.2 Der ganzzahlige Datentyp (int)

SORT int
OPS +, −, *, /, %: int × int → int
=: int → int

Weiter Zuweisungsoperatoren: += addiert den Ergebniswert zur Variablen (z.B. a += 1). Genauso z.B.: -=, *=, /=.

- Sorte beseht aus einer Teilmenge von { ... -3, -2, -1, 0 , 1, 2, 3, ... },
- Für die aufgeführten Operatoren gelten die üblichen Rechengesetze.
  - Der Operator / bezeichnet die ganzzahlige Division (Rest vernachlässigt).
  - Der Operator % liefert den Rest der ganzzahligen Division. Somit bleibt die folgende Gleichung erhalten:

$$x == y * (x/y) + (x\%y)$$

  - Priorität: Punkt-vor-Strich-Regel
- Weitere Operatoren auf den ganzen Zahlen sind die üblichen Vergleichsoperatoren

Ganze Zahlen in Java

- Der Bereich von int ist **endlich**!
  - Die ganzen Zahlen liegen zwischen **-2^{31} und $2^{31}-1$**.

- Neben int können ganze Zahlen in Java durch die Datentypen byte, shortund long repräsentiert werden.
  - Wertebereich von byte: **-2^{7} und $2^{7}-1$**
  - Wertebereich von short: **-2^{15} und $2^{15}-1$**
  - Wertebereich von long: **-2^{63} und $2^{63}-1$**

  Warum benötigt man diese verschiedenen Typen?

- *Was passiert bei einem Unterlauf bzw. Überlauf, d.h., wenn das Ergebnis außerhalb des zulässigen Bereichs fällt?*

Darstellung von ganzen positiven Zahlen

Binärdarstellung:

- Wert der Binärzahl:

$b_{m-1} b_{m-2} \ldots b_3 b_2 b_1 b_0$

$\sum_{i=0}^{m-1} b_i 2^i = 2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \ldots + 2^{m-1} b_{m-1}$

Basis der Zahlendarstellung

m entspricht wie viele Stellen eine binäre zahl hat
wir fangen mit der Nummerierung bei 0 von rechts nach links ,also ganz rechts hat die Stelle die niedrigste Potenz und ganz links hat die Stelle die höchste Potenz

- Beispiel (m=16):

$0110\ 1001\ 0011\ 1101_{(2)}$

$1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 + 1 \cdot 32 + 0 \cdot 64 + 0 \cdot 128 + 1 \cdot 256 + 0 \cdot 512 + 0 \cdot 1024 + 1 \cdot 2048 + 0 \cdot 4096 + 1 \cdot 8192 + 1 \cdot 16384 + 0 \cdot 32768 = 26941_{(10)}$

hier hat diese Binäre Zahl 16 stellen damit wir diese Zahl umwandeln von binäre zu Dezimal nutzen wir die obige formel

Umrechnen: Dezimal → Binär

- Hornerschema:

```
26972 / 2 = 13486 Rest 0
13486 / 2 =  6743 Rest 0
 6743 / 2 =  3371 Rest 1
 3371 / 2 =  1685 Rest 1
 1685 / 2 =   842 Rest 1
  842 / 2 =   421 Rest 0
  421 / 2 =   210 Rest 1
  210 / 2 =   105 Rest 0
  105 / 2 =    52 Rest 1
   52 / 2 =    26 Rest 0
   26 / 2 =    13 Rest 0
   13 / 2 =     6 Rest 1
              0   Rest 0
    6 / 2 =     3
                 est 1
                 est 1
```

mit Nullen aufgefüllt bis 16 Bit, wegen 16 Bit Worten

0110 1001 0101 1100

Least significant Bit

Most signficant Bit

Analog geht von Dezimal zu Oktal
und von Dezimal zu Hexa
wir teilen immer durch die Basis, in der wir die Zahl umwandeln wollen
wollen wir 150(dezimal also zur Basis 10) umwandeln zu einer oktalzahl dann teilen wir durch 8, die zahl die wir mit dem Schema erahlten ist dann die Oktalzahl zu dieser Zahl 150
und wollen wir 150 in ein Hexadezimalzahl umwandeln , dann teilen wir immer durch 16 die Zahl, die wir am Ende erhalten ist eine Hexa-dezimalzahl zu der Zahl 150

Bisher haben wir gelernt wie wir eine Dezimalzahl zu einer Zahl in anderen Systemen umwandeln

Schreibweisen für Bitfolgen

- Binärdarstellung: `0110 1001 0011 1101`
  - Zahl wird als Zahl zur Basis 2 geschrieben
- Octaldarstellung (Basis 8):

$$\underbrace{0}\underbrace{110}\underbrace{100}\underbrace{100}\underbrace{111}\underbrace{101}_{(2)} = 064475_{(8)}$$

von Binäre zu Oktal wir nehmen imer 3 Bits nach belibiger Reihenfolge und wandeln diese 3 Bits zu einer Dezimalzahl

$000_{(2)}$ =	$0_{(8)}$	$100_{(2)}$ =	$4_{(8)}$
$001_{(2)}$ =	$1_{(8)}$	$101_{(2)}$ =	$5_{(8)}$
$010_{(2)}$ =	$2_{(8)}$	$110_{(2)}$ =	$6_{(8)}$
$011_{(2)}$ =	$3_{(8)}$	$111_{(2)}$ =	$7_{(8)}$

- Hexadezimaldarstellung (Basis 16):

$$\underbrace{0110}\underbrace{1001}\underbrace{0011}\underbrace{1101}_{(2)} = 693D_{(16)}$$

	$0_{(16)}$	$1000_{(2)}$ =	$8_{(16)}$
	1	1001 =	$9_{(16)}$
	$2_{(16)}$	$1010_{(2)}$ =	$A_{(16)}$
$0011_{(2)}$ =	$3_{(16)}$	$1011_{(2)}$ =	$B_{(16)}$
$0100_{(2)}$ =	$4_{(16)}$	$1100_{(2)}$ =	$C_{(16)}$
$0101_{(2)}$ =	$5_{(16)}$	$1101_{(2)}$ =	$D_{(16)}$
$0111_{(2)}$ =	6	$1111_{(2)}$ =	E
=	$7_{(16)}$	0 =	$F_{(16)}$

Von Dezimal zu Hexadezimalzahl wir nehmen immer 4 Bits, auch nach belibiger Reihenfolge von rechts nach links oder umgekehrt und stellen diese 4 bits als eine dezimal-zahl dar.

Beispiele

- Das RGB-Farbformat beschreibt die Mischung von drei Farben, deren Intensität zwischen 0 und 255 liegen.
  - 3 byte werden zur Darstellung einer Farbmischung verwendet.
- Die Anzahl der Kursteilnehmer in "Objektorientierter Programmierung" kann nicht durch 1 Byte dargestellt werden.
  - Wir benötigen hierfür einen Wert vom Typ short.
- 71137 ist die Anzahl der Plätze in der Allianz Arena.
  - Wir benötigen einen Wert vom Typ int.
- Weltbevölkerung: 7 750 008 172
  - Der Wert muss durch den Typ long repräsentiert werden.

Addition von binären Zahlen:

1. Schreibe die beiden binären Zahlen untereinander, bitweise ausgerichtet.
2. Starte die Addition vom rechten Ende (Least Significant Bit, LSB). (addiere in diesem Schritt die Bits und berücksichtige den Übertrag (carry)
3. Falls der Übertrag (carry) = 1 ist, dann addiere den Carry zum nächsten Paar von Bits.
4. Wiederhole den Prozess für jedes Bitpaar
5. Am Ende der Addition: Falls ein Übertrag übrig bleibt, schreibe ihn als das höchstwertige Bit (Most Significant Bit, MSB) auf.

Rechnen mit Binärzahlen

Rechnen mit Binärzahlen funktioniert genauso wie mit Dezimalzahlen:

```
  0110 1001 0101 1100(2)   =  26972(10)
+ 0011 1010 1110 0101(2)   =  15077(10)
   1_1_1_1   1_1_1 1_1_1_1   1
= 1010 0100 0100 0001(2)   =  42049(10)
```

```
  0110 1001 0101 1100(2)   =  26972(10)
- 0011 1010 1110 0101(2)   =  15077(10)
   1_1_1     1    1_1  1_1_1
   1_1
= 0010 1110 0111 0111(2)   =  11895(10)
```

Problem: mit einer beschränkten Anzahl von Bits kann man nicht beliebig große Zahlen darstellen:

```
  0110 1001 0101 1100(2)   =  26972(10)
+ 1011 1010 1110 0101(2)   =  47845(10)
   1_1_1_1   1_1_1 1_1_1_1   1
=10010 0100 0100 0001(2)   =  74817(10)
```

Überlauf!

Begrenzung der darstellbaren Zahlen

- Mit m Bits lassen sich die Zahlen von 0 bis $2^m - 1$ darstellen.

Most Signifikant Bit:
Es ist das Bit, das den höchsten Platzwert hat
und sich ganz links in der binären Darstellung befindet.

Zum Beispiel, in der 8-Bit-binären Zahl 10101010 ist
das MSB das linke 1, das den Wert von 2^7 representiert

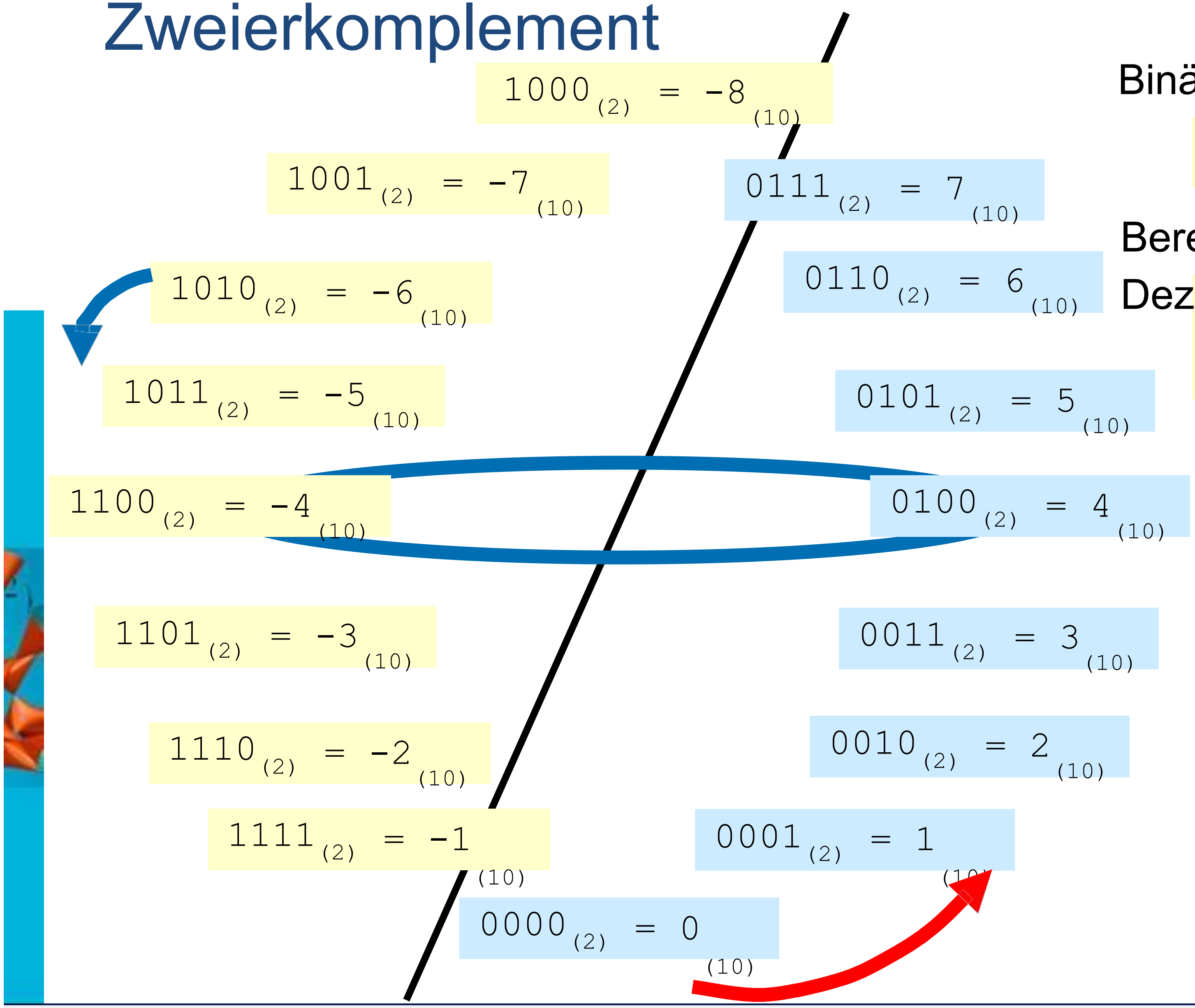
m	$2^m - 1$
4	*15*
8	*255*
16	*65 535*
32	*4 294 967 296*
64	*18 446 744 073 709 551 616*

Das Least Significant Bit (LSB) ist das niedrigstwertige Bit in einer binären Zahl. Es befindet sich ganz rechts in der Bitfolge und hat den geringsten Einfluss auf den Wert der gesamten Zahl

(2^m)-1 weil die Null mit drin ist

- **Was ist eigentlich mit negativen Zahlen?**

Darstellung ganzer Zahlen als Zweierkomplement



Binärdarstellung

$$b_{m-1}\, b_{m-2} \ldots b_2\, b_1\, b_0$$

Berechnung der Dezimalzahl

$$-2^{m-1} \cdot b_{m-1} + \sum_{i=0}^{m-2} 2^i \cdot b_i$$

Hier wird zuerst gekippt dann eine 1 addiert
wenn man das probiert bei allen zahlen von 0 bis 7 dann erhalten die negativen Zahlen --> Das ist das Zweierkoplement

Das Eins-Komplement war ein früher Schritt zur Darstellung von negativen Zahlen in binären Systemen, aber seine Nachteile in Bezug auf die doppelte Null, komplexe Arithmetik und asymmetrischen Zahlenbereich führten zur Entwicklung des Zweierkomplements.

Darstellung ganzer Zahlen als Zweierkomplement

- $1000_{(2)} = -8_{(10)}$
- $1001_{(2)} = -7_{(10)}$
- $1010_{(2)} = -6_{(10)}$
- $1011_{(2)} = -5_{(10)}$
- $1100_{(2)} = -4_{(10)}$
- $1101_{(2)} = -3_{(10)}$
- $1110_{(2)} = -2_{(10)}$
- $1111_{(2)} = -1_{(10)}$
- $0000_{(2)} = 0_{(10)}$
- $0001_{(2)} = 1_{(10)}$
- $0010_{(2)} = 2_{(10)}$
- $0011_{(2)} = 3_{(10)}$
- $0100_{(2)} = 4_{(10)}$
- $0101_{(2)} = 5_{(10)}$
- $0110_{(2)} = 6_{(10)}$
- $0111_{(2)} = 7_{(10)}$

Über-/Unterlauf

Binärdarstellung $b_{m-1} b_{m-2} \ldots b_3 b_2 b_1 b_0$

Berechnung der Dezimalzahl

$$-2^{m-1} \cdot b_{m-1} + \sum_{i=0}^{m-2} 2^i \cdot b_i$$

Vorzeichenwechsel & Rechnen

- Vorzeichenwechsel durch:
  1. Komplementieren (0 und 1 austauschen)
  2. Addieren von 1

Beispiel: (4-Bit)

von 5 zu -5

```
5 = 0101B
---------------------------
Zum Vorzeichenwechsel:
   Komplementieren   1010B
   Addieren von 1    0001B
                     -----
-5 =                 1011B

Zum Vorzeichenwechsel:
   Komplementieren   0100B
   Addieren von 1    0001B
5 = 0101B
---------------------------
```

Hier hast du eine Beispiel wo er die 5 zu -5 und umgekehrt umwandelt

von -5 zu 5

- Addition und Subtraktion arbeiten „normal“:

```
------------------------
     5        0101B
  + -7        1001B
              -----
    -2        1110B
```

wie hat er gewusst, dass diese Zahl -2 ist

weil wenn ich diese 4 Bits invertiere und 1 dazu addiere bekomme ich die 2, also diese zahl 1110 = -2

```
     3    0011B
   - 4    0100B
          -----
    -1    1111B
------------------------
```

Analog hier ist noch das gleiche Addition von einer positiiven mit einer negativen Zahl

Hier haben wir eine Demonstration dass die Addition bzw. die Subtraktion funktioniert im Zweierkomplement sehr gut

Vorzeichenwechsel & Rechnen

- Vorzeichenwechsel durch:

1. Komp... (0 und ...
2. r...

Addie... 4-Bit)

"B" steht für Binärzahl. Hat dieselbe Bedeutung wir Index $_{(2)}$.

Beispiel:

```
( 5 = 0 1 0 1 B    1010B
  Komplementieren
- Addieren von 1   0001B
Zum Vorzeichenwec
hsel:   -5 =  1011B

Zum Vorzeichenwechsel:
   Komplementieren    0100B
   Addieren von 1     0001B

         5 =
         0101B
```

- Addition und Subtraktion arbeiten „normal“:

```
     5     0101B
  + -7     1001B
           -----
    -2
    1110B
  3        0011B
- 4        0100B
           -----
 -1        1111B
```


Philipps Universität Marburg

Vorzeichenwechsel & Rechnen

- **Problem**: das Ergebnis kann eine nicht darstellbare Zahl sein!
  - Über- oder Unterlauf
  - Gilt aber für alle Zahlendarstellungen mit beschränktem Platz

```
     5        0101B
  + 11        1011B
             ------
    0        10000B
```

Warum wird der letzte Bit abgeschnitten

Wird abgeschnitten.

Weitere Operationen auf Integer-Datentypen

i++ heißt postinkriment
beschreibung: Der Ausdruck i++ erhöht den Wert von i um 1, nachdem der aktuelle Wert von i verwendet wurde.
Ablauf:
Der aktuelle Wert von i wird zurückgegeben.
Dann wird i um 1 erhöht.

- Inkrement/Dekrement-**Operationen**

  ++, --

- **Häufig werden Variablen ganzer Zahlen um 1 erhöht bzw. erniedrigt.**

```
int i = 5;
int j = i++; // j bekommt den aktuellen Wert von i, also 5
             // Danach wird i um 1 erhöht, also ist i nun 6
```

  - **Durch**

    ++i; bzw. i++;

    wird der Wert von i um 1 erhöht. Dies hat die gleiche Wirkung wie i += 1 oder i = i + 1;

  - **Durch**

    --i; bzw. i--;

    wird der Wert von i um 1 erniedrigt. Dies hat die gleiche Wirkung wie i -= 1 oder i = i -1;

- **Wir werden später noch auf Besonderheiten dieser Operatoren eingehen.**

int i = 5;
int j = ++i; // i wird um 1 erhöht (auf 6)
// j bekommt den neuen Wert von i, also 6

prä-inkrement:
Beschreibung: Der Ausdruck ++i erhöht den Wert von i um 1, bevor der Wert von i verwendet wird.
Ablauf:
i wird um 1 erhöht.
Dann wird der neue Wert von i zurückgegeben.

Bit-Operatoren

Zusammenfassung : (i++)Der ursprüngliche Wert von i wird zuerst verwendet und dann erhöht.
(++i) i wird zuerst erhöht und dann der neue Wert verwendet.

- **Diese Operatoren interpretieren eine ganze Zahl als Folge von Bits.**
- **Operatoren: &, |, ^, ~, <<, >>, >>>**
  - Die Bit-Operationen **&**, **|**, **^** und **~** (entspricht **and, or**, **xor** und **Komplement**) verknüpfen ganzzahlige Werte bitweise - ohne Rücksicht auf Zahlenwerte.
  - Die Shift-Operationen >>, << und >>> schieben ganzzahlige Werte bitweise.
    - >> und <<: Schieben nach rechts und links wird das Vorzeichenbit erhalten.
    - >>>: Schieben nach rechts und links wird mit Nullen aufgefüllt.

Post-Inkrement (i++) wird häufig verwendet, wenn man den ursprünglichen Wert zuerst benötigt und ihn dann erhöhen möchte. Ein typisches Beispiel wäre das Durchlaufen einer Schleife, bei der der aktuelle Index erst genutzt wird und dann erhöht wird

```
for (int i = 0; i < 10; i++) {
    // Der aktuelle Wert von i wird genutzt, bevor er erhöht wird
    System.out.println(i);
}
```

Prä-Inkrement (++i) wird verwendet, wenn der Wert sofort erhöht werden soll, bevor er verwendet wird. Es kann in Situationen nützlich sein, in denen die Erhöhung vor der weiteren Verwendung wichtig ist:

Multiplikation und Division mit 2

- Eine Multiplikation einer Zahl mit 2^i kann durch eine Shift-Operation erfolgen.

```
int x = 42;                      // Operand der Multiplikation
int i = 3;                       // Exponent
System.out.println(x << i);      // Liefert x* 2^i
```

- Entsprechend kann eine ganzzahlige Division durchgeführt werden.

```
int x = -42;                     // Divisor
int i = 3;                       // Dividend 2^i
System.out.println(x >> i);      // Shift rechts und Auffüllen mit
                                 // Vorzeichenbit.
```

Bisher sind die Schwächen :
1)Umwandeln zwischen den Zahlensysteme nicht rechnerisch sondern als Alogirthmus 2)Eins-Komplement und Zweierkomplement
3)Shift Operatoren

4) Die Bedeutung von Überlauf und Unterlauf
Du musst auch wissen,warum geht z.B. der Bereich von -2^n bis 2^n -1
wie viele Zahlen kann ich wirklich mit 4 Bits darstellen

Ganzzahlige Literale

- Einfache ganzzahlige Literale:
  - `2  17 -3 32767 -889275714`
- Für ganzzahlige Literale gibt es neben der Standardschreibweise auch noch eine weitere:
  - Die binäre Notation beginnt mit den Zeichen Null: **0b**
    - `0b101` entspricht $5_{(10)}$ `(=1*4+0*2+1)`
  - Die oktale Notation beginnt mit einer Null: **0**
    - `0747` entspricht $487_{(10)}$ `(=7*64+4*8+7)`
  - Die hexadezimale Notation einer Zahl beginnt mit den Zeichen **0x**. Danach können normale Ziffern oder hexadezimale Ziffern (A ... F) folgen. Diese können groß oder klein geschrieben werden.
    - $889275714_{(10)}$ entspricht `0xCafeBabe` und `0xCAFEBABE`
- Ganzzahlige Literale bezeichnen zunächst Werte des Datentyps int.
- Will man sie als long kennzeichnen, muss man **L** (oder **l**) anhängen.
  - `4242424242L` oder `0xC0B0L`

Live Vote

PIN: P5BK

×

https://ilias.uni-marburg.de/vote/P5BK

4.1.3 Der Datentyp float

- Repräsentation von Fließpunktzahlen auf dem Rechner.
  - Oft werden diese Zahlen zur Repräsentation von den reellen Zahlen aus der Mathematik benutzt.

Divisionsrest auch auf float anwendbar!

SORT float
OPS

$+, -, *, /, \%\colon\ \text{float} \times \text{float} \rightarrow \text{float}$

$=, +=, -=, *=, /=\colon\ \text{float} \rightarrow \text{float}$

- Für die Operatoren gelten die üblichen Rechengesetze aus der Mathematik für reelle Zahlen.

Besonderheiten auf Rechner

- Wie sieht die Überlaufbehandlung aus?
- Zusätzlich muss mit Rundungsfehlern gerechnet werden! Auf dem Rechner (auch in Java) gilt z.B. nicht mehr das Assoziativgesetz. Somit i.A.

$$(x + y) + z \neq x + (y + z)$$

- Neben float (32 Bits) können reelle Zahlen auch durch double (64 Bits) repräsentiert werden. Operationen sind für double entsprechend definiert.
- Wie kann ich einen Wert vom Typ int in einen Wert vom Typ float konvertieren?

Kontrollrechnung

- Ungenauigkeit durch Längenbeschränkung
  - z. B. kann 1/10 nicht auf einem Computer genau dargestellt werden.
- Weiteres Problem: Fehler kumulieren in Berechnungen!

- Kleinste darstellbare positive Zahl: $\approx 1{,}4 * 10^{-45}$
- Größte darstellbare positive Zahl: $\approx 3{,}4 * 10^{38}$
- Der Datentyp double mit 64 Bits arbeitet zwar genauer, besitzt aber die gleichen Probleme wie der Datentyp float.

Kontrollrechnung

- Ungenauigkeit durch Längenbeschränkung
  - z. B. kann 1/10 nicht auf einem Computer genau dargestellt werden.
- Weiteres Problem: Fehler kumulieren in Berechnungen!

Achtung: bei der Ausgabe von Floats rundet Java automatisch. So kann das Runden umgangen werden:
System.out.println(String.format("%.20f", 1.0f / 10.0f));

- Kl e
  10^{-45}
- Größte darstellbare positive Zahl: ≈ 3,4 *

Float in formatierten Text konvertieren.
von 20 Nachkommastellen

- Der Datentyp double mit 64 Bits arb eite t aber die gleichen Probleme wie der Date ntyp besitzt float.

Beispiel (Berechnung der Zahl e):

Formel zur Berechnung der Zahl e:

$$e = \sum_{i=0} \frac{1}{i!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

```
/** Diese Methode berechnet die Zahl e.
 * @result Liefert einen approximativen Wert von e
 */
float eulerLeftToRight () {
        float sum = 1.0f;
        int i = 1;
        while (i <= 15) {
                sum = sum + 1.0f / fakultaet(i);
                i = i+1;
        }
        return sum;
}
…
```

Beispiel (Berechnung der Zahl e):

Formel zur Berechnung der Zahl e:

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

```
/** Diese Methode berechnet die Zahl e.
 * @result Liefert einen approximativen Wert von e
 */
float eulerRightToLeft () {
        float sum = 0.0f;
        int i= 15; while
        (i >= 1) {
                sum = sum + 1.0f / fakultaet(i);
                i = i-1;
        }
        return sum + 1.0f;
}
…
```

```
…
/** Diese Methode berechnet die Fakultät.
  *@param Eingabeparameter für die Berechnung
  *@return Liefert als Ergebnis n!
 */
long fakultaet(int n) {
        long res = 1;
        while (n >= 1) {
                res = res*n;
                n = n - 1;
        }
        return res;
}
```

- Als Ergebnisse der Methoden eulerLeftToRight und eulerRightToLeft bekommen wir folgende Ausgabe: `2.718282`

```
2.7182817
Differenz: 2.3841858E-7
```

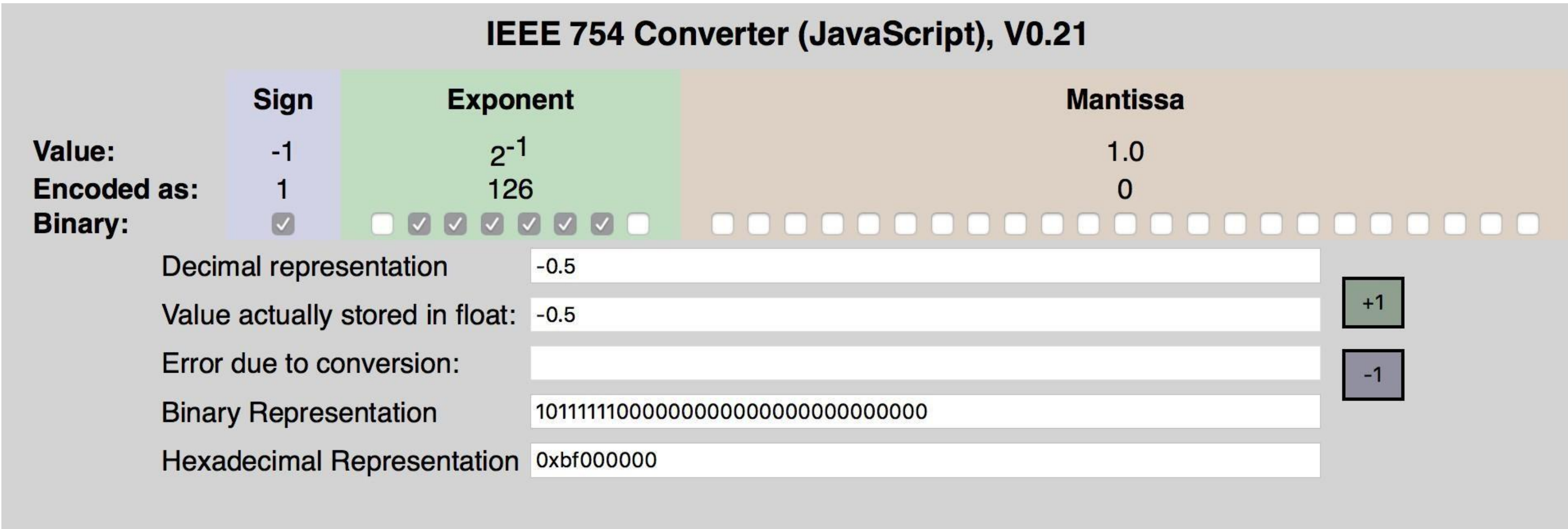
Da stimmt doch irgendetwas nicht?

Binärdarstellung von Fließpunktzahlen

- Darstellung von Fließpunktzahlen durch

$$\pm \mathit{Mantisse} \quad 2^{\mathit{Exponent}}$$

- Es lassen sich nur Zahlen exakt darstellen, die Brüche mit einer Zweierpotenz im Nenner sind



https://www.h-schmidt.net/FloatConverter/IEEE754de.html

Binärdarstellung von Fließpunktzahlen

- Darstellung von Fließpunktzahlen durch

$$\pm Mantisse \cdot 2^{Exponent}$$

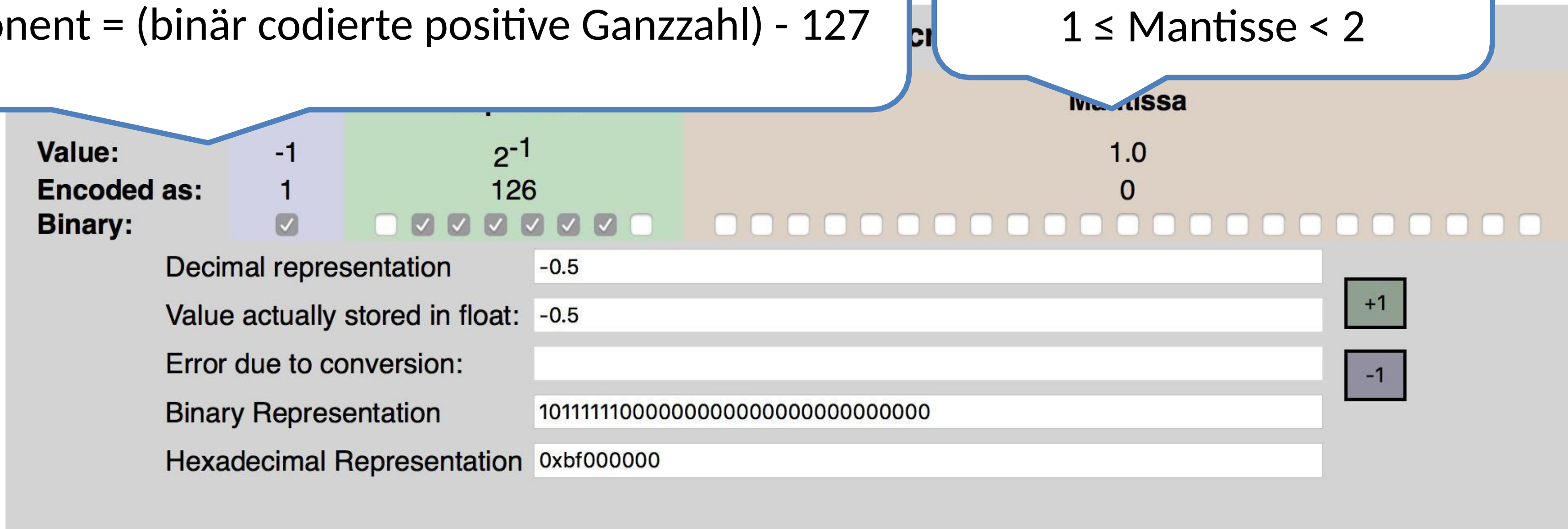
- Es lassen sich nur Zahlen exakt darstellen, die Brüche mit

Kein 2er-Komplement. Stattdessen:

Exponent = (binär codierte positive Ganzzahl) - 127

Mantissen-Bitmuster (M) steht für: 1,M

1 ≤ Mantisse < 2



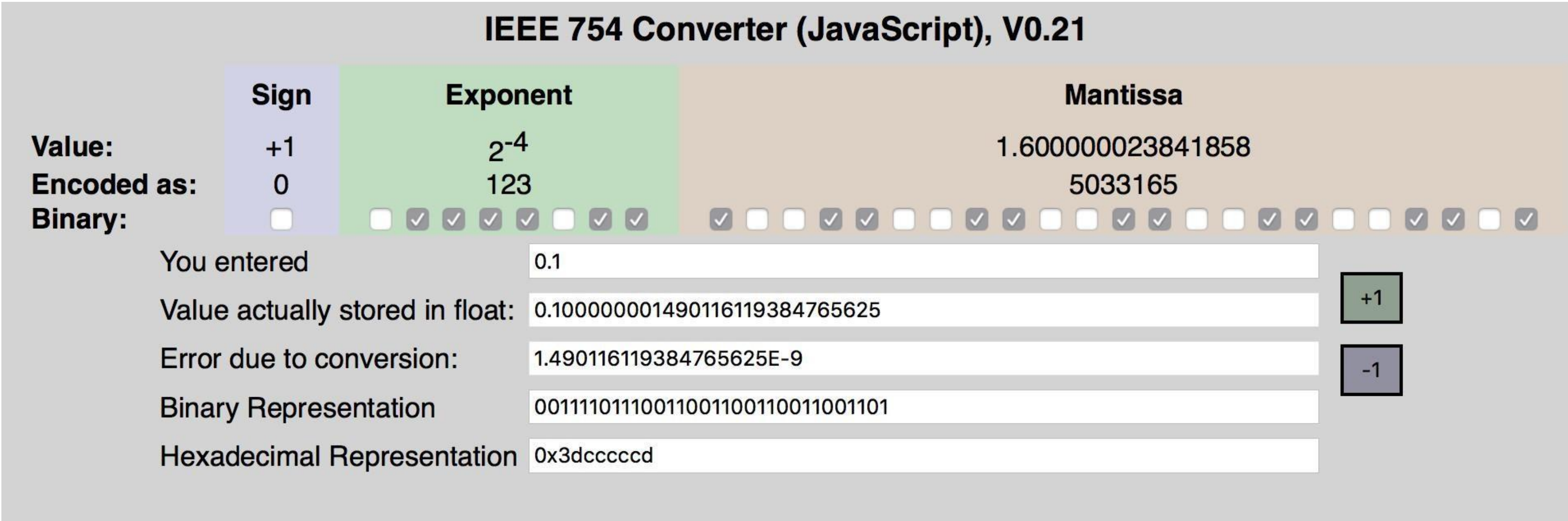
https://www.h-schmidt.net/FloatConverter/IEEE754de.html

Binärdarstellung von Fließpunktzahlen

- Darstellung von Fließpunktzahlen durch

$$\pm \mathit{Mantisse} \cdot 2^{\mathit{Exponent}}$$

- Es lassen sich nur Zahlen exakt darstellen, die Brüche mit einer Zweierpotenz im Nenner sind



https://www.h-schmidt.net/FloatConverter/IEEE754de.html

Weitere Operationen für float

- Auf den Datentypen für Fließpunktzahlen sind die üblichen arithmetischen Operationen definiert:
  - *Inkrement/Dekrement*-**Operationen**

    `++, --`

    rechnen einfach +1 und -1
  - *Vergleichs*-**Operationen**

    `==, !=, <, <=, >, >=`

    problematisch wegen Rechengenauigkeit!

Literale für Fließpunktzahlen

- Zur Erinnerung: in Java sind float (floating point number, 32 Bit) und double (double precision number, 64 Bit).
  - Fließpunkt-Literale bezeichnen normalerweise Werte des Datentyps double.
- Durch Anhängen eines der Suffixe F oder D (bzw. f oder d) spezifiziert man sie explizit als Werte des Datentyps float bzw. double.
- Beispiele für Fließpunkt-Literale des Datentyps float sind :

```
1e1f 2.f .3f 0f 1.0f 2.0f  3.14f 6.022137e+23f
```

Exponentialdarstellung

- Beispiele für Fließpunkt-Literale des Datentyps double:

```
1e1 2. .3 0.0 2.0 42.42 3.14 1e-9d 1e137
```

Live Vote

PIN: 6DJ7

×

https://ilias.uni-marburg.de/vote/6DJ7

Meine Notizen:

Erklärung zu links-Shift Operator(<<) und Rechts-Shift-Operator(>>) und Unsigned Rechts-Shift-Operator (>>>):

1) Beschreibung: Der Links-Shift-Operator verschiebt alle Bits einer Zahl um eine bestimmte Anzahl von Positionen nach links. Die leeren Bits auf der rechten Seite werden mit 0 aufgefüllt Wirkung: Dies entspricht einer Multiplikation der Zahl mit 2^n, wobei n Anzahl der Verschiebungen ist.

Syntax (x<<n):
x: Die Zahl, deren Bits verschoben werden.
n: Die Anzahl der Positionen, um die die Bits verschoben werden.
Beispiel:

```
int x = 3;// Binär: 00000011
int result = x << 2; // Verschiebt die Bits um 2 nach links
// Ergebnis: 00001100 (im Dezimalsystem 12)
```

Nach der Ausführung ist result = 12.

2) Rechts-Shift-Operator (>>)

Beschreibung: Der Rechts-Shift-Operator verschiebt alle Bits einer Zahl um eine bestimmte Anzahl von Positionen nach rechts. Die leeren Bits auf der linken Seite werden bei positiven Zahlen mit dem Vorzeichenbit (0 bei positiven und 1 bei negativen Zahlen) aufgefüllt. Dies wird als arithmetischer Shift bezeichnet.

Wirkung: Dies entspricht einer ganzzahligen Division der Zahl durch 2^n, wobei n die Anzahl der Verschiebungen ist

Syntax: x >> n
x: Die Zahl, deren Bits verschoben werden.
n: Die Anzahl der Positionen, um die die Bits verschoben werden

```
int x = 12;        // Binär: 00001100
int result = x >> 2; // Verschiebt die Bits um 2 nach rechts
// Ergebnis: 00000011 (im Dezimalsystem 3)
```

Nach der Ausführung ist result = 3.

Für negative Zahlen:

```
int x = -12;  // Binär: 11111100 (als Zweierkomplement)
int result = x >> 2; // Verschiebt die Bits um 2 nach rechts
// Ergebnis: 11111111 (im Dezimalsystem -3)
//Hier wird das Vorzeichenbit (1 für negativ) verwendet, um die leeren Stellen aufzufüllen.
```

3) Unsigned Rechts-Shift-Operator (>>>)

Beschreibung: Der unsigned Rechts-Shift-Operator verschiebt alle Bits einer Zahl um eine bestimmte Anzahl von Positionen nach rechts. Im Gegensatz zu >> werden die leeren Bits auf der linken Seite immer mit 0 aufgefüllt, unabhängig vom Vorzeichen der Zahl. Dies wird als logischer Shift bezeichnet.

Wirkung: Dieser Operator wird häufig für Unsigned-Bit-Operationen verwendet, wo das Vorzeichen keine Rolle spielt.

Syntax: x >>> n
x: Die Zahl, deren Bits verschoben werden.
n: Die Anzahl der Positionen, um die die Bits verschoben werden.

```
int x = -12;  // Binär: 11111100 (als Zweierkomplement)
int result = x >>> 2; // Verschiebt die Bits um 2 nach rechts
// Ergebnis: 00111111 (im Dezimalsystem 1073741821)
```

Nach der Ausführung ist result = 1073741821. Dies ist ein großes positives Ergebnis, da die linke Auffüllung mit 0 erfolgt, wodurch das Vorzeichen ignoriert wird

Zusammenfassung der Unterschiede

<< (Links-Shift): Verschiebt Bits nach links 2) Füllt mit 0 auf der rechten Seite auf. 3)Entspricht einer Multiplikation mit 2^n

>> (Rechts-Shift):

Signed VS Unsigned

Signed (Vorzeichenbehaftet)

Beschreibung: Eine "signed" Zahl ist eine vorzeichenbehaftete Zahl, was bedeutet, dass sie sowohl positive als auch negative Werte darstellen kann. Ein Bit, normalerweise das höchstwertige Bit (Most Significant Bit, MSB), wird als Vorzeichenbit verwendet:

1 im Vorzeichenbit zeigt an, dass die Zahl positiv ist.
2 im Vorzeichenbit zeigt an, dass die Zahl negativ ist.
Darstellung: In einem 8-Bit-System bedeutet dies, dass der Bereich der darstellbaren Werte von -128 bis 127 reicht.

Beispiel: In einem 8-Bit-System wird die Zahl -12 als 11110100 im Zweierkomplement dargestellt.
Bereich:

In einem 8-Bit-System: -128 bis 127
In einem 16-Bit-System: -32.768 bis 32.767
In einem 32-Bit-System: -2.147.483.648 bis 2.147.483.647
Unsigned (Ohne Vorzeichen)

Beschreibung: Eine "unsigned" Zahl ist eine Zahl ohne Vorzeichen, was bedeutet, dass sie nur nicht-negative Werte (also nur positive Zahlen und Null) darstellen kann. Alle Bits der Zahl werden zur Darstellung des Wertes verwendet, und es gibt kein Vorzeichenbit.
Darstellung: Da es kein Vorzeichenbit gibt, kann eine unsigned Zahl einen größeren positiven Wertebereich darstellen als eine gleich große signed Zahl.
Beispielsweise reicht der Bereich der darstellbaren Werte in einem 8-Bit-System von 0 bis 255.
Beispiel: In einem 8-Bit-System wird die Zahl 12 als 00001100 dargestellt.
Bereich:
In einem 8-Bit-System: 0 bis 255
In einem 16-Bit-System: 0 bis 65.535
In einem 32-Bit-System: 0 bis 4.294.967.295
Zusammenfassung der Unterschiede
Signed Zahlen:

Können sowohl positive als auch negative Werte darstellen.
Nutzen das höchstwertige Bit als Vorzeichenbit.
Haben einen kleineren positiven Bereich, da ein Teil der Bits für das Vorzeichen verwendet wird.
Unsigned Zahlen:

Können nur nicht-negative Werte darstellen (0 oder positive Zahlen).
Nutzen alle Bits zur Darstellung des Wertes, ohne ein Vorzeichenbit.
Haben einen größeren positiven Bereich im Vergleich zu signed Zahlen desselben Bitformats.
Beispiel für 8-Bit Zahlen
Signed 8-Bit Bereich: -128 (binär 10000000) bis 127 (binär 01111111)
Unsigned 8-Bit Bereich: 0 (binär 00000000) bis 255 (binär 11111111)

Rundungsfehler

Warum entstehen Rundungsfehler?
Begrenzte Präzision:

Computer können nur eine begrenzte Anzahl von Bits verwenden, um Zahlen darzustellen. Viele Zahlen, insbesondere irrationale oder sehr kleine/ große Dezimalzahlen, können nicht exakt in binärer Form dargestellt werden. Daher müssen sie auf den nächsten darstellbaren Wert gerundet werden.
Darstellung von Dezimalzahlen in Binärform:

Manche Dezimalzahlen können nicht exakt in binärer Form dargestellt werden. Zum Beispiel kann die Zahl 0,1 im Dezimalsystem nicht exakt als binäre Zahl dargestellt werden. Sie wird als eine endlose binäre Folge gespeichert, die abgeschnitten werden muss, was zu einem Rundungsfehler führt.
Beispiele für Rundungsfehler
Darstellung von 0,1 in Binär:

Die Dezimalzahl 0,1 entspricht in binärer Form 0,0001100110011001100110011... (eine periodische Zahl). Da Computer eine begrenzte Anzahl von Bits haben, wird diese Zahl abgeschnitten, was zu einem kleinen Fehler führt.
Addition von Gleitkommazahlen:

Wenn man viele kleine Zahlen addiert, kann sich der Rundungsfehler summieren. Zum Beispiel kann der Ausdruck (0.1 + 0.2) im Computer zu einem Wert wie 0.30000000000000004 führen, anstatt zu 0.3.
Auswirkungen von Rundungsfehlern
Kumulierte Fehler: Bei vielen Berechnungen können kleine Rundungsfehler sich summieren und zu signifikanten Abweichungen führen.
Vergleichsprobleme: Zwei scheinbar gleiche Zahlen können aufgrund von Rundungsfehlern unterschiedlich dargestellt werden, was bei Vergleichen (==) zu unerwarteten Ergebnissen führen kann.
Numerische Instabilität: In bestimmten Algorithmen, besonders in der numerischen Mathematik, können Rundungsfehler zu instabilen Ergebnissen führen
Maßnahmen zur Minimierung von Rundungsfehlern
Erhöhte Präzision:

Durch die Verwendung von Datentypen mit höherer Präzision (z. B. double statt float in Java) kann der Rundungsfehler verringert werden.
Korrekte Rundung:

Verwende Rundungsfunktionen, um sicherzustellen, dass Zwischenergebnisse korrekt gerundet werden.
Vermeidung von Subtraktionen:

Wenn möglich, sollten Subtraktionen vermieden werden, bei denen ähnliche Werte abgezogen werden, da dies die Auswirkungen von Rundungsfehlern verstärken kann.
Verwendung spezieller Algorithmen:

In der numerischen Mathematik gibt es spezielle Algorithmen, die entwickelt wurden, um die Auswirkungen von Rundungsfehlern zu minimieren.

Du musst auch wissen, wie man die Zahlen mit komma (Gleitkommazahlen) in binär umwandelt und warum bestimmte zahlen wie z.B. 0,5 exakt darstellen kann und andere wie 0,1 nicht exakt darstellen kann

Die Dezimalzahl 0,1 kann nicht exakt im Computer dargestellt werden, weil sie in der Binärdarstellung (die im Computer verwendet wird) eine unendliche, nicht- periodische Folge von Bits hat. Dies liegt daran, dass das Binärsystem (zur Basis 2) nicht in der Lage ist, bestimmte Dezimalzahlen exakt zu repräsentieren, ähnlich wie das Dezimalsystem (zur Basis 10) nicht in der Lage ist, bestimmte Brüche exakt darzustellen (z. B. 1/3 wird im Dezimalsystem als 0.3333... dargestellt).

Binäre Darstellung von Dezimalzahlen

In der Computerwelt werden Zahlen in Binärform (mit 0 und 1) dargestellt. Während einige Dezimalzahlen wie 0,5 einfach in Binärform umgewandelt werden können (0,1 in Binär), ist dies bei anderen, wie 0,1, nicht möglich.

Umwandlung von 0,1 in Binär

Um die Binärdarstellung von 0,1 zu berechnen, müssen wir wiederholt 0,1 mit 2 multiplizieren und die Ganzzahlanteile notieren: 0,1*2=0,2
(Ganzzahlanteil 0)
0,2*2=0,4(Ganzzahlanteil 0)
0,4*2=0,8(Ganzahlanteil 0)
0,8*2=1,6(Ganzzahlanteil 1)
0,6*2= 1,2 (Ganzahlanteil 1)
0,2*2 = 0,4 (Ganzahanteil 0)

Dieser Prozess führt zu einer periodischen Binärsequenz: 0,00011001100110011..., die unendlich weitergeht. Begrenzte Präzision im Computer

Computer haben eine begrenzte Anzahl von Bits zur Verfügung, um eine Zahl zu speichern. Da die Binärdarstellung von 0,1 unendlich ist, kann der Computer diese Zahl nur auf eine bestimmte Anzahl von Bits runden. Diese Rundung führt dazu, dass die Darstellung von 0,1 im Computer eine kleine Ungenauigkeit aufweist. im gängigen IEEE 754-Standard für Gleitkommazahlen wird die Zahl 0,1 als eine Näherung gespeichert. Für float (32-Bit) und double (64-Bit) Gleitkommazahlen werden die Bits wie folgt gespeichert:

float (32-Bit): 0.10000000149011612 (ungefähr) double (64-Bit): 0.10000000000000000555 (ungefähr)
Diese kleine Abweichung zeigt sich, wenn man z. B. mehrere Rechenoperationen durchführt und vergleicht.

Kurzer Einblick in Gleitkommazahlen bzw. Fließkommazahlen
wir wissen in der Mathematik wir haben natürlich, ganze, rationale und reelle zahlen .wenn wir uns nur die Zahlen mit Komma angucken, wir haben einmal die zahlen mit Komma die exakt dargestellt werden können als eine dezimal zahl mit endlichen nicht periodischen Nachkommastellen
und rationale zahlen mit komm die eine unendliche periodische Nachkommastelle haben
(die zahlen, die eine unendliche nicht periodische Nachkommastellen haben sind die reelen Zahlen)
Inder Informatik (Zahlen mit Komma heißen Gleitkommazahlen bzw. Fließkommazahlen)
Der Begriff bezieht sich darauf, dass das Komma innerhalb der Zahl "gleiten" kann, um unterschiedliche Größenordnungen darzustellen.
(Was heißt gleiten ?) bedeutet, dass das Komma in einer Zahl verschoben werden kann, um unterschiedliche Größenordnungen (also Zehnerpotenzen) darzustellen. Dies ist ein grundlegendes Konzept in der wissenschaftlichen Notation oder bei Gleitkommazahlen.
Beispiel: die Zahle 12345
komma nach der 1. Stelle also 1,2345= 1,2345 * 10 ^4
Komma nach der 12: 12,345 = 12,345 * 10^3
komma nach 123 123,45 = 123,45*10^2
Komma nach der 1234: 1234,5 → 1234,5 * 10^1
Das Verschieben des Kommas in einer Zahl verändert die Darstellung der Zahl, aber durch das Multiplizieren mit einer entsprechenden Zehnerpotenz bleibt der numerische Wert gleich. In der wissenschaftlichen Notation wird das Komma immer so platziert, dass nur eine Ziffer vor dem Komma steht, und der Rest der Zahl als Zehnerpotenz ausgedrückt wird.

Wissenschaftliche Notation

Die wissenschaftliche Notation ist eine Methode zur Darstellung sehr großer oder sehr kleiner Zahlen, indem man sie als Produkt einer Zahl zwischen 1 und 10 und einer Potenz von 10 schreibt. Diese Methode erleichtert das Arbeiten mit extrem großen oder kleinen Werten und ist in den Naturwissenschaften und Ingenieurwissenschaften weit verbreitet.

Struktur der wissenschaftlichen Notation

Eine Zahl wird in der wissenschaftlichen Notation allgemein so geschrieben: $(a * 10^{b})$

wobei a ist eine zahl,die mindestens 1 ist und kleiner als 10 ist und b ist eine ganze zahl die, die Potenz von 10 angibt.

Beispiel :Die Zahl 0,00043 kann als $4{,}3 \times 10^{-4}$ geschrieben werden 4,3 ist a also die zahl zwischen 1 und 10 und -4 ist die Potenz von 10

Achtung :In der wissenschaftlichen Notation darf nur eine Ziffer vor dem Komma stehen, und zwar eine Zahl zwischen 1 und 10. Der Rest der Zahl wird durch die Potenz von 10 angegeben wir wollen die Dezimalzahl ist `0,75` umwandeln in eine binärZahl.

Schritt 1: Multiplizieren Sie die Nachkommastelle mit 2: 0,75×2=1,5 (Das Ergebnis ist `1,5`. Der Ganzzahlanteil ist `1`, und die Nachkommastelle ist `0,5`.)

Schritt 2: Vermerken Sie den Ganzzahlanteil: (Der Ganzzahlanteil `1` wird das erste Bit der Binärdarstellung)

Schritt 3: Wiederholen Sie den Vorgang mit der neuen Nachkommastelle (`0,5`):0,5×2=1,0

Das Ergebnis ist `1,0`. Der Ganzzahlanteil ist wieder `1`, und die Nachkommastelle ist `0`.

Schritt 4: Vermerken Sie den Ganzzahlanteil:

Der Ganzzahlanteil `1` wird das nächste Bit der Binärdarstellung.

Schritt 5: Beenden:

Da die Nachkommastelle jetzt `0` ist, können wir aufhören.

Also 0,75 wird in binär als 0,11 dargestellt . Aber warum 0,75=0,11 in binär ?

Das erste Bit nach dem Komma (`0,1`) repräsentiert `0,5` (also 2^-1) und Das zweite Bit (`0,01`) repräsentiert `0,25` (also 2^-2). Zusammen ergibt 0,1+0,11(also 0,5+0,25) =0,75 und damit **0,75** (Dezimal) = **0,11** (Binär)

Nehmen wir die Dezimalzahl `0,3`.

0,3×2=0,6 (Ganzzahlanteil: `0`)

0,6×2=1,2 (Ganzzahlanteil: `1`)

0,2×2=0,4 (Ganzzahlanteil: `0`)

0,4×2=0,8 (Ganzzahlanteil: `0`)

0,8×2=1,6 (Ganzzahlanteil: `1`)

0,6×2=1,2 (Ganzzahlanteil: `1`)

- 0,2×2=0,4 (Ganzzahlanteil: `0`)

Die Binärdarstellung von `0,3` ist eine unendliche periodische Folge: `0,01001100110011...`

Nehmen wir die Dezimalzahl `0,625`.

Schritt 1:

- 0,625×2=1,25 (Ganzzahlanteil: `1`)

Schritt 2:

- 0,25×2=0,5 (Ganzzahlanteil: `0`)

Schritt 3:

- 0,5×2=1,0 (Ganzzahlanteil: `1`)

ErDie Binärdarstellung von `0,625` ist `0,101`.