4. Zahlendarstellungen

Wertebereich der Typen `int`, `float` und `double` Gemischte Ausdrücke und Konversionen; Löcher im Wertebereich; Fliesskommazahlensysteme; IEEE Standard; Grenzen der Fliesskommaarithmetik; Fliesskomma-Richtlinien;

Binäre Zahlendarstellungen

Binäre Darstellung ("Bits" aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \cdots + b_1 \cdot 2 + b_0$

Beispiel: `101011` entspricht 43.

Niedrigstes Bit, Least Significant Bit (LSB)

Höchstes Bit, Most Significant Bit (MSB)

Binäre Zahlen: Zahlen der Computer?

Wahrheit: Computer rechnen mit Binärzahlen.

NEUE ZÜRCHER ZEITUNG

TECHNIK

Das programmgesteuerte Rechengerät an der Eidgenössischen Technischen Hochschule in Zürich

Binäre Zahlen: Zahlen der Computer?

Klischee: Computer reden 0/1-Kauderwelsch.

Rechentricks

- Abschätzung der Grössenordnung von Zweierpotenzen[3]:

$2^{10} = 1024 = 1\text{Ki} \approx 10^3$.
$2^{20} = 1\text{Mi} \approx 10^6$,
$2^{30} = 1\text{Gi} \approx 10^9$,
$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi} \approx 4 \cdot 10^9$.
$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}$.

[3] Dezimal vs. Binäre Einheiten: MB - Megabyte vs. MiB - Megabibyte (etc.)
kilo (K, Ki) – mega (M, Mi) – giga (G, Gi) – tera(T, Ti) – peta(P, Pi) – exa (E, Ei)

Wertebereich des Typs `int`

```
public class Main {
    public static void main(String[] args) {
        Out.print("Minimum int value is ");
        Out.println(Integer.MIN_VALUE);
        Out.print("Maximum int value is ");
        Out.println(Integer.MAX_VALUE);
    }
}
```

```
Minimum int value is -2147483648.
Maximum int value is 2147483647.
```

Woher kommen diese Zahlen?

Wertebereich des Typs `int`

Repräsentation mit 32 Bits. Wertebereich umfasst die 2^{32} ganzen Zahlen:

$$\{-2^{31}, -2^{31}+1, \ldots, -1, 0, 1, \ldots, 2^{31}-2, 2^{31}-1\}$$

Woher kommt gerade diese Aufteilung?

Rechnen mit Binärzahlen (4 Stellen)

Einfache Addition

2	0010
$+3$	$+0011$
5	0101

Einfache Subtraktion

5	0101
-3	-0011
2	0010

Rechnen mit Binärzahlen (4 Stellen)

Addition mit Überlauf

7	0111
$+9$	$+1001$
16	$(1)0000$

Negative Zahlen?

5	0101
$+(-5)$	$????$
0	$(1)0000$

Rechnen mit Binärzahlen (4 Stellen)

Einfacher: -1

1	0001
$+(-1)$	1111
0	$(1)0000$

Nutzen das aus:

3	0011
$+?$	$+????$
-1	1111

Rechnen mit Binärzahlen (4 Stellen)

Invertieren!

3	0011
$+(-4)$	$+1100$
-1	$1111 \widehat{=} 2^B - 1$

a	a
$+(-a-1)$	$+\bar{a}$
-1	$1111 \widehat{=} 2^B - 1$

Rechnen mit Binärzahlen (4 Stellen)

- Negation: Inversion und Addition von 1

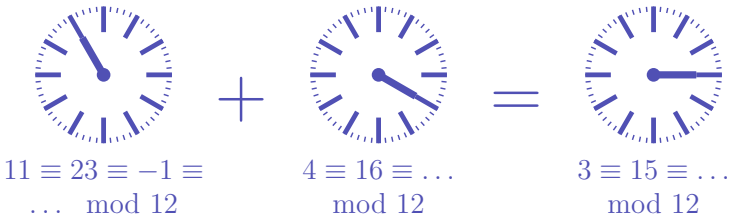
$$-a \quad \widehat{=} \quad \bar{a} + 1$$

- Wrap-around Semantik (Rechnen modulo 2^B)

$$-a \quad \widehat{=} \quad 2^B - a$$

Warum das funktioniert

Modulo-Arithmetik: Rechnen im Kreis[4]



$11 \equiv 23 \equiv -1 \equiv \ldots \mod 12$ $\quad$ $4 \equiv 16 \equiv \ldots \mod 12$ $\quad$ $3 \equiv 15 \equiv \ldots \mod 12$

[4] Die Arithmetik funktioniert auch mit Dezimalzahlen (und auch für die Multiplikation)

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001	**111**	-1
2	010	**110**	-2
3	011	**101**	-3
4	**100**	**100**	-4
5	**101**		
6	**110**		
7	**111**		

Das höchste Bit entscheidet über das Vorzeichen.

Zweierkomplement

- Negation durch bitweise Negation und Addition von 1.

$$-2 = -[0010] = [1101] + [0001] = [1110]$$

  - Arithmetik der Addition und Subtraktion *identisch* zur vorzeichenlosen Arithmetik.

$$3 - 2 = 3 + (-2) = [0011] + [1110] = [0001]$$

  - Intuitive „Wrap-Around" Konversion negativer Zahlen.

$$-n \to 2^B - n$$

  - Wertebereich: $-2^{B-1} \ldots 2^{B-1} - 1$

Überlauf und Unterlauf

- Arithmetische Operationen (`+`,`-`,`*`) können aus dem Wertebereich herausführen.
- Ergebnisse können inkorrekt sein.

`power8`: $15^8 = -1732076671$

`power20`: $3^{20} = -808182895$

- Es gibt *keine Fehlermeldung!*

„Richtig Rechnen”

```
public class Main {

    public static void main(String[] args) {
        Out.print("Celsius: ");
        int celsius = In.readInt();
        int fahrenheit = 9 * celsius / 5 + 32;
        Out.print(celsius + " degrees Celsius are ");
        Out.println(fahrenheit + " degrees Fahrenheit");
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        Out.print("Celsius: ");
        float celsius = In.readInt();
```


Typen `float` und `double`

- sind die fundamentalen Typen für Fliesskommazahlen
- approximieren den Körper der reellen Zahlen $(\mathbb{R}, +, \times)$ in der Mathematik
- haben grossen Wertebereich, ausreichend für viele Anwendungen (`double` hat mehr Stellen als `float`)
- sind auf vielen Rechnern sehr schnell

Fixkommazahlen

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

`0.0824 = 0000000.082` ← dritte Stelle abgeschnitten

Nachteile

- Wertebereich wird *noch* kleiner als bei ganzen Zahlen.
- Repräsentierbarkeit hängt von der Stelle des Kommas ab.

Fliesskommazahlen

- feste Anzahl signifikante Stellen (z.B. 10)
- plus Position des Kommas

`82.4` = $824 \cdot 10^{-1}$

`0.0824` = $824 \cdot 10^{-4}$

- Zahl ist $\mathit{Signifikand} \times 10^{\mathit{Exponent}}$

Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber ...
- Wertebereich ist zusammenhängend (keine „Löcher”): $\mathbb{Z}$ ist „diskret”.

Fliesskommatypen:

- Über- und Unterlauf selten, aber ...
- es gibt Löcher: $\mathbb{R}$ ist „kontinuierlich”.

Löcher im Wertebereich

```
public class Main {
    public static void main(String[] args) {
    Out.print("First number =? ");
    float n1 = In.readFloat();

    Out.print("Second number =? ");
    float n2 = In.readFloat();

    Out.print("Their difference =? ");
    float d = In.readFloat();

    Out.print("computed difference − input difference = ");
    Out.println(n1−n2−d);
    }
}
```

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.2351742E-8

Ja was ist denn hier los?

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),
- $e_{\min}$, der kleinste Exponent,
- $e_{\max}$, der grösste Exponent.

Bezeichnung:

$$F(\beta, p, e_{\min}, e_{\max})$$

Fliesskommazahlensysteme

$F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$d_i \in \{0, \ldots, \beta - 1\}, \quad e \in \{e_{\min}, \ldots, e_{\max}\}.$

In Basis-β-Darstellung:

$$\pm\, d_0 \bullet d_1 \ldots d_{p-1} \times \beta^e,$$

Fliesskommazahlensysteme

Beispiel

- $\beta = 10$

Darstellungen der Dezimalzahl 0.1

$$1.0 \cdot 10^{-1}, \quad 0.1 \cdot 10^{0}, \quad 0.01 \cdot 10^{1}, \quad \ldots$$

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm\, d_0 \bullet d_1 \ldots d_{p-1} \times \beta^e, \qquad d_0 \neq 0$$

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Bemerkung 2

Die Zahl 0 (und alle Zahlen kleiner als $\beta^{e_{\min}}$) haben keine normalisierte Darstellung (werden wir später beheben)!

Menge der normalisierten Zahlen

$$F^*(\beta, p, e_{\min}, e_{\max})$$

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Binäre und dezimale Systeme

- Intern rechnet der Computer mit $\beta = 2$ (binäres System)
- Literale und Eingaben haben $\beta = 10$ (dezimales System)
- Eingaben müssen umgerechnet werden!

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

- Also: $x' = b_{-1}\bullet b_{-2}b_{-3}b_{-4}\ldots = 2 \cdot (x - b_0)$
- Schritt 1 (für x): Berechnen von b_0:

$$b_0 = \begin{cases} 1, & \text{falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$$

- Schritt 2 (für x): Berechnen von $b_{-1}, b_{-2}, \ldots$:

  Gehe zu Schritt 1 (für $x' = 2 \cdot (x - b_0)$)

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2
1.2	$b_{-5} = \mathbf{1}$	0.2	0.4

$\Rightarrow 1.0\overline{0011}$, periodisch, *nicht* endlich

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich, also gibt es Fehler bei der Konversion in ein (endliches) binäres Fliesskommazahlensystem.
- `1.1f` und `0.1f` sind nicht 1.1 und 0.1, sondern geringfügig fehlerhafte Approximationen dieser Zahlen.

$$\begin{aligned} \texttt{1.1} &= \underline{1.1000000000000000}888178\ldots \\ \texttt{1.1f} &= \underline{1.1000000}238418\ldots \end{aligned}$$

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 1.011 \cdot 2^{-1} \\ \hline & = 1.001 \cdot 2^{0} \end{aligned}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl 2. Binäre Addition der Signifikanden 3. Renormalisierung 4. Runden auf p signifikante Stellen, falls nötig

Der IEEE Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird fast überall benutzt
- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ plus $0, \infty, \ldots$
- Double precision (`double`) Zahlen:

  $F^*(2, 53, -1022, 1023)$ plus $0, \infty, \ldots$
- Alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl

32-bit Darstellung einer Fliesskommazahl

$\pm$ Exponent Mantisse

$\pm$ $2^{-126}, \ldots, 2^{127}$ $1.00000000000000000000000$

$0, \infty, \ldots$ $\cdots$ $1.11111111111111111111111$

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 8 Bit für den Exponenten (256 mögliche Werte)(254 mögliche Exponenten, 2 Spezialwerte: 0, ∞,...)

$\Rightarrow$ insgesamt 32 Bit.

Der IEEE Standard 754

Warum

$$F^*(2, 53, -1022, 1023)?$$

- 1 Bit für das Vorzeichen
- 52 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 11 Bit für den Exponenten (2046 mögliche Exponenten, 2 Spezialwerte: 0, ∞,...)

$\Rightarrow$ insgesamt 64 Bit.

Fliesskomma-Richtlinien Regel 1

Regel 1

Teste keine gerundeten Fliesskommazahlen auf Gleichheit!

```
for (float i = 0.1; i != 1.0; i += 0.1){
    Out.println(i);
}
```

Endlosschleife, weil i niemals exakt 1 ist!

Fliesskomma-Richtlinien Regel 2

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ +& 1.000 \cdot 2^0 \\ =& \ 1.00001 \cdot 2^5 \\ \text{“=”}& \ 1.000 \cdot 2^5 \quad \text{(Rundung auf 4 Stellen)} \end{aligned}$$

Addition von 1 hat keinen Effekt!

Fliesskomma-Richtlinien Regel 3

Regel 3

Subtrahiere keine zwei Zahlen sehr ähnlicher Grösse!

Auslöschungsproblematik (ohne weitere Erklärung).

5. Wahrheitswerte

Boolesche Funktionen; der Typ `boolean`; logische und relationale Operatoren; Kurzschlussauswertung

Wo wollen wir hin?

```
int a = In.readInt();
if (a % 2 == 0){
    Out.println("even");
} else {
    Out.println("odd");
}
```

Verhalten hängt ab vom Wert eines Booleschen Ausdrucks

Boolesche Werte in der Mathematik

Boolesche Ausdrücke können zwei mögliche Werte annehmen:

F oder T

- F entspricht *„falsch”*
- T entspricht *„wahr”*

Der Typ `boolean` in Java

- Repräsentiert *Wahrheitswerte*
- Literale `false` und `true`
- Wertebereich $\{$*false*, *true*$\}$

```
boolean b = true; // Variable mit Wert true (wahr)
```

Relationale Operatoren

`a < b` (kleiner als)
`a >= b` (grösser gleich)
`a == b` (gleich)
`a != b` (ungleich)

Zahlentyp $\times$ Zahlentyp $\rightarrow$ `boolean`

Boolesche Funktionen in der Mathematik

- Boolesche Funktion

$$f : \{F, T\}^2 \rightarrow \{F, T\}$$

- F entspricht „falsch”.
- T entspricht „wahr”.

$\mathrm{AND}(x, y)$ $x \wedge y$

- “Logisches Und”

$$f : \{F, T\}^2 \rightarrow \{F, T\}$$

- F entspricht „falsch”.
- T entspricht „wahr”.

x	y	$\mathrm{AND}(x, y)$
F	F	F
F	T	F
T	F	F
T	T	T

Logischer Operator &&

`a && b` (logisches Und)

`boolean` $\times$ `boolean` $\rightarrow$ `boolean`

```
int n = -1;
int p = 3;
boolean b = (n < 0) && (0 < p); // b = true (wahr)
```

$\mathrm{OR}(x, y)$ $x \vee y$

- “Logisches Oder”

$$f : \{F, T\}^2 \to \{F, T\}$$

- F entspricht „falsch”.
- T entspricht „wahr”.

x	y	$\mathrm{OR}(x, y)$
F	F	F
F	T	T
T	F	T
T	T	T

Logischer Operator ||

`a || b` (logisches Oder)

`boolean` $\times$ `boolean` $\to$ `boolean`

```
int n = 1;
int p = 0;
boolean b = (n < 0) || (0 < p); // b = false (falsch)
```


$\mathrm{NOT}(x)$ $\neg x$

- “Logisches Nicht”

$$f : \{F, T\} \to \{F, T\}$$

- F entspricht „falsch”.
- T entspricht „wahr”.

x	$\mathrm{NOT}(x)$
F	T
T	F

Logischer Operator !

`!b` (logisches Nicht)

`boolean` $\to$ `boolean`

```
int n = 1;
boolean b = !(n < 0); // b = true (wahr)
```

Präzedenzen

```
!b && a
   ⇕
(!b) && a

a && b || c && d
      ⇕
(a && b) || (c && d)

a || b   &&   c || d
      ⇕
 a || (b && c) || d
```


Präzedenzen

Der unäre logische Operator `!`

bindet stärker als

binäre arithmetische Operatoren. Diese

binden stärker als

relationale Operatoren,

und diese binden stärker als

binäre logische Operatoren.

```
7 + x < y && y != 3 * z || ! b
7 + x < y && y != 3 * z || (!b)
```


DeMorgansche Regeln

- `!(a && b) == (!a || !b)`
- `!(a || b) == (!a && !b)`

`!` (reich *und* schön) == (arm *oder* hässlich)

Anwendung: Entweder ... Oder (XOR)

`(x || y)`	`&& !(x && y)`	x oder y, und nicht beide
`(x || y)`	`&& (!x || !y)`	x oder y, und eines nicht
`!(!x && !y)`	`&& !(x && y)`	nicht keines, und nicht beide
`!(!x && !y || x && y)`		nicht: keines oder beide

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden zuerst* aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

```
x != 0 && z / x > y
```

$\Rightarrow$ Keine Division durch 0