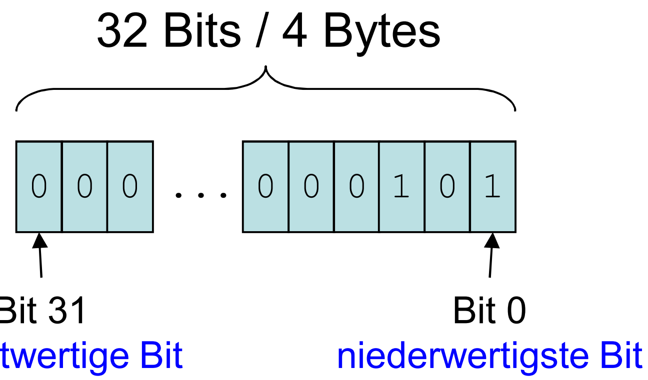
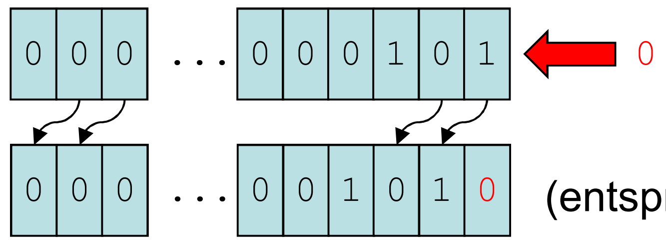
# Bitoperationen

`int x = 5;`




Links-Shift um eine Position

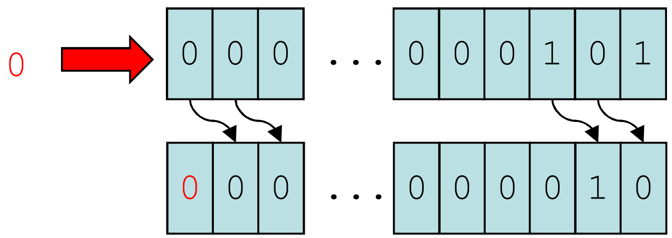
ergibt




(entspricht dem Wert $10_{10}$)

Rechts-Shift um eine Position

ergibt




(entspricht dem Wert $2_{10}$)

# Bitoperationen

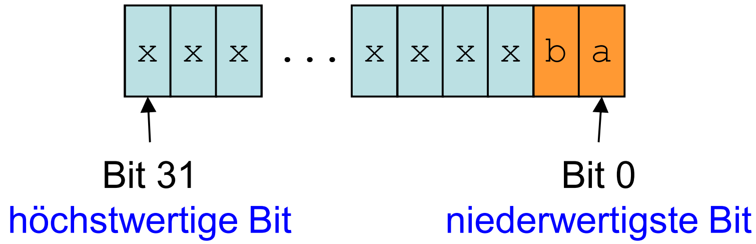
| Operator | Beispiel | Wert Beispiel | Wirkung |
|---|---|---|---|
| `~` | `~0` | `-1` | Bitweise Negation<br>$\sim 0...000_2 = 1...111_2 = -1_{10}$ |
| `<<` | `1 << 2` | `4` | Shiften nach links<br>$0...01_2 << 2 = 0...0100_2 = 4_{10}$ |
| `>>` | `-4 >> 2` | `-1` | Shiften nach rechts mit Vorzeichenbehandlung<br>$11..1100_2 >> 2 = 111...111_2$ |
| `>>>` | `-4 >>> 2` | `1073741823` | Shiften nach rechts (0 wird eingeschoben)<br>$1..1100_2 >>> 2 = 0011...111_2 = 1073741823_{10}$ |
| `&` | `9 & 3` | `1` | Bitweises Und<br>$0...01001_2 \& 0...00011_2 = 0...01_2 = 1_{10}$ |
| `\|` | `9 \| 3` | `11` | Bitweises Oder<br>$0...01001_2 \mid 0...00011_2 = 0...01011_2 = 11_{10}$ |
| `^` | `9 ^ 3` | `10` | Exlusives Oder<br>$0...01001_2 \wedge 0...00011_2 = 0...01010_2 = 10_{10}$ |

- Shiften: von einer Seite 0-Bits reinschieben, auf der anderen Seite fallen entsprechend viele Bits raus
- mit Vorzeichenbehandlung: eingeschobene Bits entsprechend ursprünglich höchsten Bit (0 oder 1)

# Beispiel

- Aufgabenstellung: gebe den Wert der beiden letzten / niederwertigen Bits eines int-Wertes aus
- Idee: isoliere ein Bit und gebe dann den Wert dieses Bits aus (0 oder 1)

```
int zahl;
```




- Programmieransatz:
  – Schritt 1: `zahl & 0x1` liefert den Wert des 0. Bits `a` (0 oder 1)
  – Schritt 2: `zahl >> 1` schiebt alle Bits eine Position nach rechts. Dadurch wird das ursprünglich zweite Bit `b` das neue niederwertigste Bit (und damit Situation wie bei Schritt 1)
  – Schritt 3: `zahl & 0x1` liefert den Wert des Bits `b` (0 oder 1)

- In der Operation `zahl & 0x1` nennt man `0x1` eine Bit-Maske

# Java-Programm zum Beispiel

```
public class Bitwerte {
    public static void main(String[] args) {
        // Beispielwert
        int wert = -1234567;
        // hierher wird jeweils das Bit extrahiert
        int bitWert;

        //---------- 1. Bit behandeln ----------------
        // alle anderen Bits ausser unterstem Bit maskieren
        bitWert = wert & 0x1;
        // Bit-Wert ausgeben
        System.out.println("1.Bit hat den Wert " + bitWert);

        //---------- 2. Bit behandeln ----------------
        // nach rechts shiften
        wert = wert >> 1;
        // alle anderen Bits ausser unterstem Bit maskieren
        bitWert = wert & 0x1;
        // Bit-Wert ausgeben
        System.out.println("2.Bit hat den Wert " + bitWert);
    }
}
```

# Überlauf

- Das Rechnen mit ganzzahligen Werten liefert immer exakte Resultate!
- Ausnahme: Überlauf
- Hier am Beispiel des Datentyps `byte` mit 8 Bits:

| | Bit 7<br>höchstwertige Bit | | | | | | | Bit 0<br>niederwertigste Bit |
|---|---|---|---|---|---|---|---|---|
| 127 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| + 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Überlauf | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| = -128 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- Die Arithmetik mit ganzzahligen Werten ignoriert einen Überlauf
- Es liegt in der Verantwortung eines Programmierers, dies zu vermeiden!

# Zwischenstand

- Ganzzahlige Typen werden auch zur Darstellung von Bitfolgen entsprechender Länge verwendet.
- Es gibt eine Reihe von Operationen, die bitweise auf einer solchen Bitfolge arbeiten.
- Ganzzahlige Operationen liefern immer das exakte Ergebnis
- Ausnahme: Bereichsüberlauf, bei dem stillschweigend weitergerechnet wird

**Reflektion**

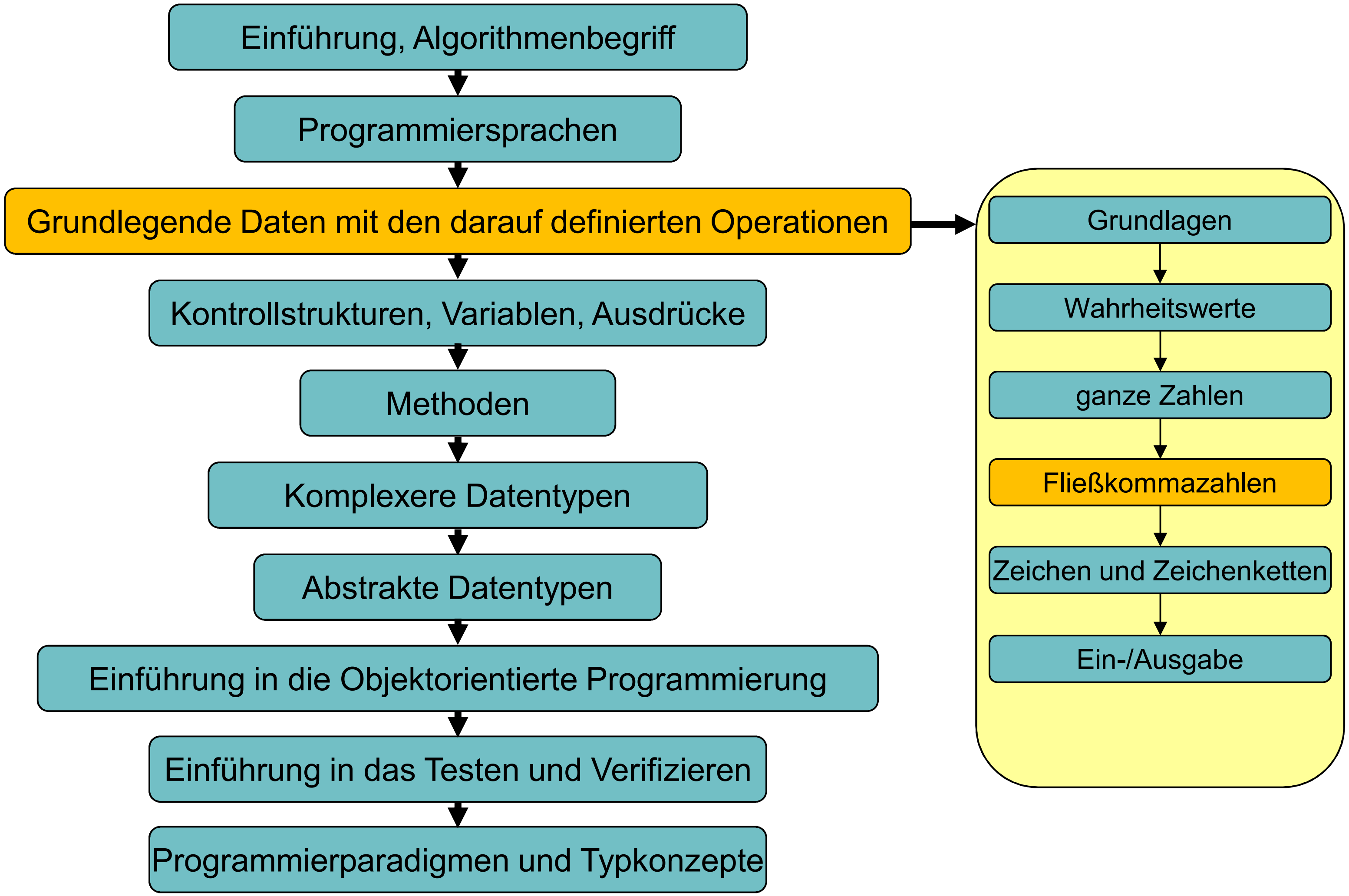
- Kann aus zwei positiven Zahlen und einer Operation auf ihnen durch Bereichsüberlauf eine negative Zahl entstehen?
- Kann aus zwei negativen Zahlen eine positive werden?
- Kann aus einer negativen und einer positiven eine negative/positive Zahl entstehen? Jeweils Beispiele?

# Zusammenfassung

- Ebenso wie in der Mathematik gibt es auch in Programmiersprachen eine Unterscheidung unterschiedlicher Zahlenmengen
- Ganze Zahlen werden intern durch eine Zweierkomplementdarstellung repräsentiert
- Angabe von Konstanten in dezimal, oktal und hexadezimal (und seit Java 7 binär) möglich
- Division und Modulo mit ganzzahliger Bedeutung
- Möglicher Überlauf muss durch Programmierer vermieden werden
- Steckbrief ganze Zahlen in Java

| | |
|---|---|
| Name | `byte, short, int, long` |
| Wertemenge | Teilmenge der ganzen Zahlen |
| Kodierung | 1,2,4,8 Bytes mit Zweierkomplementdarstellung |
| Konstanten | Dezimalzahlen, Oktalzahlen, Hexadezimalzahlen |
| Operationen | `+,-,*,/,%,...` |
| Nutzung | für ganze Zahlen und Bitfolgen |
| Besonderheiten | Berechnungen alle exakt bis auf Überlauf |

# Inhalt dieser Veranstaltung



# Reelle Zahlen

- Wie lassen sich reelle Zahlen darstellen?
- Zur Erinnerung:
  - n Bits ergeben genau $2^n$ verschiedene Werte
  - Im Dezimalsystem steht 12,34 für die Zahl $1 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$
  - Allgemein für Basis B: $$\boxed{Wert(z_{n-1}z_{n-2}...z_1z_0, d_1...d_m) = \sum_{i=0}^{n-1} z_i \cdot B^i + \sum_{j=1}^{m} d_j \cdot B^{-j}}$$
- Erster Ansatz:
  - Teile die verfügbaren n Bits auf in einem Vorkommaanteil und einen Nachkommaanteil
  - Vor- und Nachkommaanteil enthält Koeffizienten für Zweierpotenzen
- Beispiel: $$011{,}110_2 = \sum_{i=0}^{2} z_i \cdot 2^i + \sum_{j=1}^{3} d_j \cdot 2^{-j}$$
$$= (0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) + (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3})$$
$$= (0 + 2 + 1) + (0{,}5 + 0{,}25 + 0)$$
$$= 3{,}75_{10}$$

# Halblogarithmische Schreibweise

- Nachteil des ersten Ansatzes: feste Aufteilung in $n_1$ Bits für Vorkommaanteil und $n_2$ Bits für Nachkommaanteil
- Beispiel: n=16, $n_1$=8, $n_2$=8: nur Zahlen im Bereich (128,-129) darstellbar

- Idee: halblogarithmische Schreibweise wie etwa $+6{,}02 \cdot 10^{23}$

- Allgemeines Format für solch eine Darstellung zu einer Basis B:
  - Vorzeichen V (Beispiel oben: 0 für +, 1 für -)
  - Mantisse M (Beispiel oben: 6,02)
  - Exponent E (Beispiel oben: 23)
  - $\mathrm{wert}(V,M,E)_B = (-1)^V \cdot M \cdot B^E$

- Eine Mantisse $M \neq 0$ zu einer Basis B heißt normalisiert, wenn $1 \leq |M| < B$, d.h. die Darstellung hat z.B. für B=2 eine Form $1{,}d_1 \ldots d_m$.
- Darstellung einer reellen Zahl in halblogarithmischer Schreibweise und normalisierter Mantisse nennt man Fließkommazahl (oder Gleitkommazahl)

# Darstellungsgenauigkeit

- Für m Bits und eine normalisierte Mantisse $d_1 \ldots d_m$ zur Basis 2 kann man als Zahl darstellen $d_1 \cdot 2^{-1} + d_2 \cdot 2^{-2} + \ldots d_m \cdot 2^{-m}$
- Konsequenzen:
  - Durch die Beschränkung auf m Bits in der Mantisse ist die Genauigkeit beschränkt, mit der man eine reelle Zahl darstellen kann (Beispiel s.u.)
  - Der maximale Rundungsfehler beträgt $2^{-(m+1)}$
  - Nur reelle Zahlen, die sich als Summe von Zweierpotenzen zur Basis 2 darstellen lassen, sind auch exakt darstellbar
- Beispiel aus dem Dezimalsystem:
  Die Zahl $0{,}123456_{10}$ lässt sich mit 4 erlaubten (Nachkomma-)Dezimalziffern annäherungsweise darstellen als $0{,}1234_{10}$ oder $0{,}1235_{10}$, aber nicht exakt
- Aufgabe: Versuchen Sie die Zahl $0{,}1_{10}$ als Summe von negativen Zweierpotenzen $d_j \cdot 2^{-j}$ darzustellen!

# IEEE 754 Darstellungsformat

- Die Organisation IEEE hat im Standard 754 mehrere Formate (und Anforderungen) für Fließkommazahlen festgelegt, die in heutigen Prozessoren verwendet werden

- Die für uns wichtigsten sind die Spezifikationen für 32 und 64 Bits

- 32 Bit / 4 Bytes:
  - 1 Bit Vorzeichen, 8 Bit Exponent, 23 Bit Mantisse
  - Mit m=23 ergibt sich ein maximaler Rundungsfehler von $2^{-(23+1)} \approx 5{,}96 \cdot 10^{-8}{}_{10}$ oder 7 signifikante Dezimalnachkommastellen

- 64 Bit / 8 Bytes:
  - 1 Bit Vorzeichen, 11 Bit Exponent, 52 Bit Mantisse
  - Mit m=52 ergibt sich ein maximaler Rundungsfehler von $2^{-(52+1)} \approx 9 \cdot 10^{-15}{}_{10}$ oder 14 signifikante Dezimalnachkommastellen

# Besonderheiten zur Norm IEEE 754

- Mantisse
  - Die Mantisse wird in einer normalisierten Darstellung zur Basis 2 gespeichert, so dass genau eine 1 vor dem Komma steht (außer für die Zahl 0 ist dies immer möglich)
  - Von der normalisierten Mantisse wird die erste Ziffer in der Darstellung weggelassen (weil sie ja immer 1 ist außer bei der Zahl 0)

- Der Exponent wird im Excess-127 Code dargestellt
  - Excess-127 Code: für eine Zahl x ist die Darstellung im Excess-127 Code die Zahl x+127 in Dualdarstellung
    Beispiel x=5: $5+127 = 132_{10} = 10000100_2$
  - Diese Excess-127 Darstellung hat Vorteile beim Vergleich von IEEE 754 Zahlen in Prozessoren

# Zwischenstand

- Halblogarithmische Schreibweise erlaubt die Darstellung auch größerer Zahlenbereiche (kleiner Vorkommaanteil und großer Nachkommaanteil oder großer Vorkommaanteil und kleiner Nachkommaanteil)
- Nicht alle Zahlen sind mehr exakt darstellbar
- In Prozessoren wird die IEEE-754-Darstellung verwandt

**Reflektion**

- Erläutern Sie mit eigenen Worten, was die Vorteile und Nachteile einer halblogarithmischen Darstellung (bei einer festen Stellenzahl) sind.

# Umwandlung einer reellen Zahl

- Schritt 1: Darstellung der reellen Zahl als Dualzahl mit Vorzeichen, getrennt mit Dualzahlvorkommaanteil und einem Dualzahlnachkommaanteil
  Beispiel: $5{,}125_{10} = 101{,}001_2$
  weil $5_{10}=101_2$ und $0{,}125_{10}=0 \cdot 0{,}5_{10}+0 \cdot 0{,}25_{10}+1 \cdot 0{,}125_{10}=0{,}001_2$

- Schritt 2: Normalisierung der Mantisse, so dass genau eine 1 vor dem Komma steht ("Verschieben des Kommas")
  Beispiel: $101{,}001_2 \rightarrow 1{,}01001_2$

- Schritt 3: Anpassen des Exponenten, so dass die Kommaverschiebung wertmäßig wieder ausgeglichen wird
  Beispiel: $101{,}001_2 = 1{,}01001_2 \cdot 2^2$

- Schritt 4: Kodierung des Exponenten im Excess-127 Code
  Beispiel: 2 im Excess-127 Code: $127_{10}+2_{10}=129_{10}=10000001_2$

- Schritt 5: Zusammensetzung der Darstellung in Vorzeichen, Exponent, Mantisse (ohne führende 1) entsprechend der Darstellungsform (32/64 Bit)
  Beispiel (32 Bit): 0 10000001 010010…000

# Darstellungsbedingte Probleme

- Das Beispiel zur Problematik wird im Dezimalsystem gegeben
- Annahme im Beispiel:
  - Mantissenlänge 3 (Dezimal-)Ziffern
  - Addition der beiden Zahlen in normalisierter Mantisse $1{,}234 \cdot 10^1$ und $9{,}999 \cdot 10^3$
- (Interne) Schritte bei der Addition:
  - Anpassen der Werte auf gleichen größeren Exponenten $0{,}01234 \cdot 10^3$ und $9{,}999 \cdot 10^3$. Aufgrund der Mantissenbeschränkung kann dies aber nur als $0{,}012 \cdot 10^3$ und $9{,}999 \cdot 10^3$ intern dargestellt werden!·
  - Addition der Werte: $0{,}012 \cdot 10^3 + 9{,}999 \cdot 10^3 = 10{,}011 \cdot 10^3$
  - Normalisierung der Mantisse: $1{,}001 \cdot 10^4$
  - Korrekter Wert wäre $1{,}001134 \cdot 10^4$
- Solche Probleme können auftreten bei
  - Operationen wie Addition auf Zahlen sehr unterschiedlicher Größe
  - Subtraktion fast gleicher Zahlen
  - Division durch Werte nahe 0

# Fließkommazahlen in Java

| Datentyp | Anzahl Bits | Anzahl Bytes | Wertebereich |
|---|---|---|---|
| `float` | 32 | 4 | ca. $-3{,}4 \cdot 10^{38}$ bis $+3{,}4 \cdot 10^{38}$ |
| `double` | 64 | 8 | ca. $-1{,}8 \cdot 10^{308}$ bis $1{,}8 \cdot 10^{308}$ |

- Übereinkunft: wenn nichts dagegen spricht (zu hoher Speicherbedarf), nimmt man im Programm für reelle Werte den Datentyp `double`
- Konstanten sind vom Typ `double`
- Viele mathematische Funktionen erwarten als Argument und liefern als Resultat Werte vom Typ `double`

# Fließkommakonstanten

- Fließkommakonstanten haben, gegenüber einer dezimalen Ganzzahldarstellung, wenigstens eines (oder eine Kombination) von:
  - einen Dezimalpunkt (kein Komma!) mit nichtleerem Vorkommaanteil oder nichtleerem Nachkommaanteil oder beides
  - einen Exponentteil der Form `e` oder `E` gefolgt von einer int-Konstanten, wobei der Exponent zur Basis 10 angegeben wird (eine Angabe Ex entspricht $\cdot 10^x$)
  - einen Fließkommasuffix `f` / `F` (float) oder `d` / `D` (für double)

- Wird kein Suffix `f`/`F` angegeben, so ist die Konstante vom Typ `double`

- Beispiele:
  - `0.3, .3, 3.`
  - `3E2, .3E2 3.0E2`
  - `3f, 3D, 3.0f, 3.e3f, 3.0e-3f`

# Arithmetische Operationen mit Fließkommawerten

| Operator | Beispiel | Wert Beispiel | Wirkung |
|---|---|---|---|
| `+` | `+3.0` | +3.0 | Vorzeichenplus |
| `-` | `-3.0` | -3.0 | Vorzeichenminus |
| `+` | `2.0 + 4.0` | 6.0 | Addition |
| `-` | `4.0 - 2.0` | 2.0 | Subtraktion |
| `*` | `2.0 * 2.0` | 4.0 | Multiplikation |
| `/` | `8.0 / 2.0` | 4.0 | Division (nicht ganzzahlig) |
| `%` | `9.0 % 2.0` | 1.0 | Rest bei Division (Modulo) |

# Beispiel

```
public class FloatBeispiel {
    public static void main(String[] args) {

        float f1, f2;
        double d1, d2;

        // Fliesskommakonstanten des Typs float haben ein Suffix f
        f1 = 0.1f;
        f2 = 0.1f;
        d1 = 0.1;
        d2 = 0.1;

        // Wir geben das Ergebnis einiger Operationen aus
        System.out.println("f1 * f2: " + (f1 * f2));
        System.out.println("d1 * d2: " + (d1 * d2));
    }
}
```

Ausgabe (beides mathematisch falsch!):

```
f1 * f2: 0.010000001
d1 * d2: 0.010000000000000002
```

# Vergleichsoperationen mit Fließkommawerten

| Operator | Beispiel | Wert Beispiel | Wirkung |
|---|---|---|---|
| `<` | `3.0 < 4.0` | `true` | kleiner |
| `<=` | `3.0 <= 4.0` | `true` | kleiner oder gleich |
| `>` | `3.0 > 4.0` | `false` | größer |
| `>=` | `5.0 >= 3.0` | `true` | größer oder gleich |
| `==` | `3.0 == 4.0` | `false` | gleich (Achtung!) |
| `!=` | `3.0 != 4.0` | `true` | ungleich (Achtung!) |

- Diese Operationen liefern Ergebnis vom Typ `boolean`
- Der Test auf Gleichheit und Ungleichheit sollte bei Fließkommawerten nur mit Vorsicht angewandt werden
- Grund: `0.1f == 0.099999999f` ergibt `true`
- Statt dessen besser einen Test auf verschwindend geringen Abstand:

  ```
  Math.abs(x-y) < toleranz
  ```

  für einen selbst gewählten und anwendungsabhängigen Wert von `toleranz` (zum Beispiel: `1e-10`)

# Weitere mathematische Operationen

| Funktion | Beispiel | Wert Beispiel | Wirkung |
|---|---|---|---|
| `Math.abs(x)` | `Math.abs(-3.0)` | `3.0` | Absolutwert $\lvert x \rvert$ einer Zahl |
| `Math.ceil(x)` | `Math.ceil(3.1)` | `4.0` | Aufrunden auf ganze Zahl |
| `Math.floor(x)` | `Math.floor(3.1)` | `3.0` | Abrunden auf ganze Zahl |
| `Math.pow(x,y)` | `Math.pow(3.0, 2.0)` | `9.0` | Potenz $x^y$ |
| `Math.sqrt(x)` | `Math.sqrt(9.0)` | `3.0` | Quadratwurzel |
| `Math.sin(x)` | `Math.sin(0.0)` | `0.0` | Sinus |
| `Math.cos(x)` | `Math.cos(0.0)` | `1.0` | Cosinus |
| `Math.tan(x)` | `Math.tan(0.0)` | `0.0` | Tangens |
| `Math.exp(x)` | `Math.exp(1.0)` | `2.71...` | $e^x$ |
| `Math.log(x)` | `Math.log(1.0)` | `0.0` | Logarithmus zur Basis e |
| `Math.log10(x)` | `Math.log10(10.0)` | `1.0` | Logarithmus zur Basis 10 |

Zu beachten:

- die trigonometrischen Funktionen arbeiten mit Radiantwerten
- Umwandlung von Grad x nach Radiant y: $y = x \cdot (\pi / 180)$
- Umwandlung von Radiant y nach Grad x: $x = y \cdot (180 / \pi)$
- Es gibt auch entsprechende Umwandlungsfunktionen in `Math`

# Zwischenstand

- Bei arithmetischen Operationen mit Fließkommawerten können erhebliche Probleme bzgl. der Genauigkeit des Resultats auftreten
- `float` und `double` dienen in Java der Darstellung von Fließkommazahlen
- Neben den Grundrechenarten gibt es eine Vielzahl von Operationen auf Fließkommazahlen (Taschenrechnerfunktionalität)
- Der Test auf (Un-)Gleichheit ist problematisch

**Reflektion**

- Welche Toleranzangabe würde bei dem Test `Math.abs(x-y) < toleranz` prinzipiell Sinn machen, welche nicht (wieso)?
  1E10, 1E1, 1E-1, 1E-5, 1E-10, 1E-15, 1E-20, 1E-25, 1E-30, 1E-100, 1E-1000

# Beispiel

- Schiefer Wurf der Physik (ohne Luftreibung)
- Gegeben: Anfangsgeschwindigkeit $v_0$, Abstoßwinkel $\alpha$
- Gesucht: Höhe y bei Entfernung x
- Formel dazu (g=9,81 m/s²): $y = \tan(\alpha) \cdot x - \frac{g \cdot x^2}{2 \cdot (v_0 \cdot \cos(\alpha))^2}$

```
public class SchieferWurf {
    public static void main(String[] args) {
        double v0 = 30.0;                         // Anfangsgeschwindigkeit
        double alpha = 45.0 * (Math.PI / 180.0); // Abschusswinkel
        double g = 9.81;                          // Erdschwerebeschleunigung
        double x = 10.0;                          // gegebener x-Wert
        double y;                                 // gesuchter y-Wert

        // Berechnung der Formel
        y = Math.tan(alpha)*x
            - ((g * x*x) / (2.0 * Math.pow(v0 * Math.cos(alpha), 2.0)));

        // Ausgabe des Ergebnisses
        System.out.println("Hoehe am Punkt " + x + " ist " + y);
    }
}
```

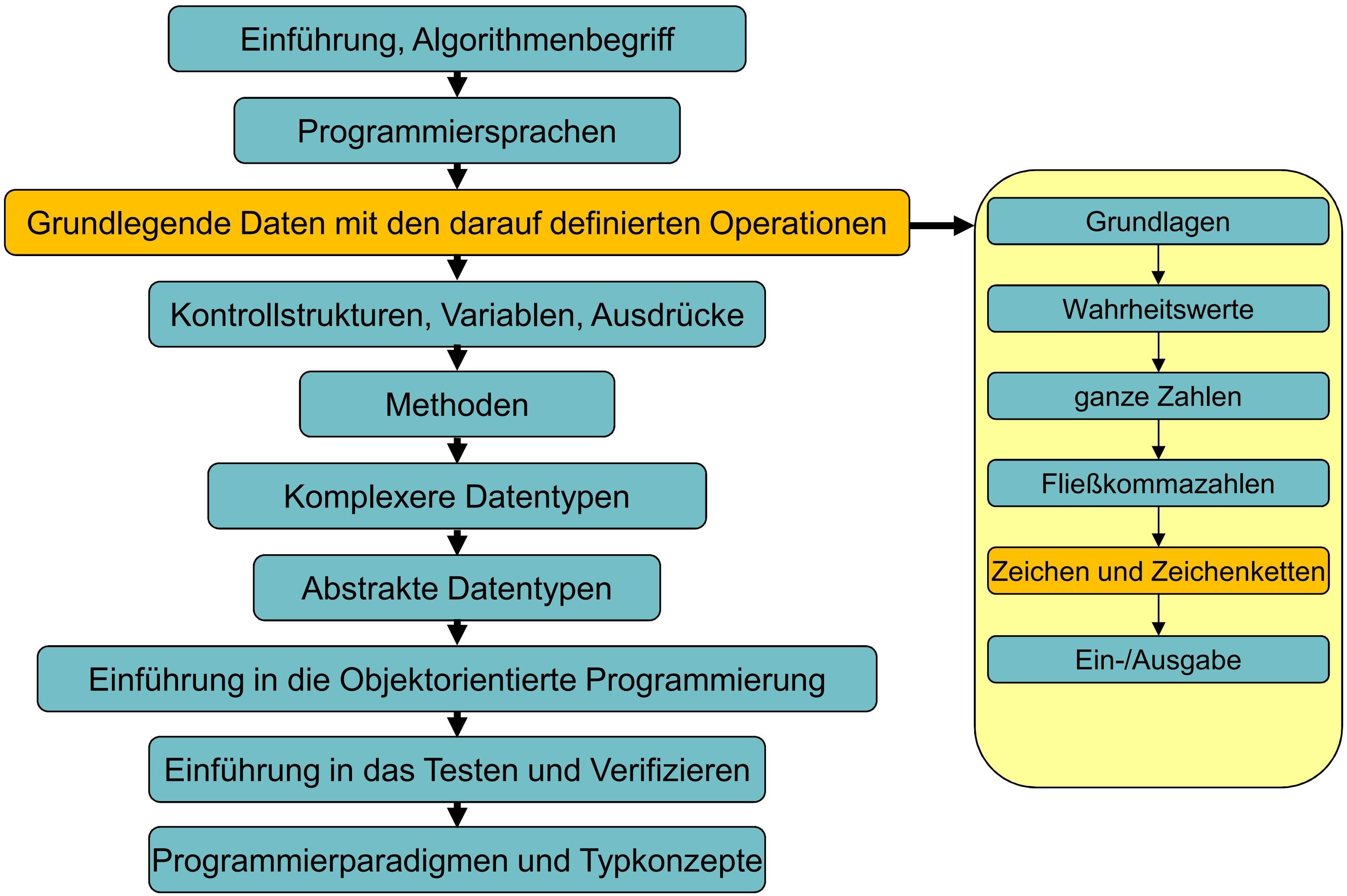
# Zusammenfassung

- Fließkommazahlen decken Teile der reellen Zahlen ab
- Selbst "einfache" reelle Zahlen wie 0,1 lassen sich nicht exakt darstellen
- Arithmetik muss nicht exakt sein, mathematische Gesetze gelten nicht unbedingt (Assoziativgesetz, Distributivgesetz)

- Steckbrief Fließkommazahlen

| | |
|---|---|
| Name | `float, double` |
| Wertemenge | Teilmenge der reellen Zahlen |
| Kodierung | 4 oder 8 Bytes, IEEE 854 |
| Konstanten | Punkt- und/oder Exponentialteil mit optionalem Suffix |
| Operationen | `+,-,*,/,%,...` |
| Nutzung | für reelle Zahlen |
| Besonderheiten | exakte Darstellung von Werten und exakte Arithmetik nicht garantiert |

# Inhalt dieser Veranstaltung



# Buchstaben / Zeichen

- Nächste Frage: wie lassen sich Buchstaben oder allgemein Zeichen darstellen?
- Dazu zu klären: welche Zeichen sollen kodiert werden?
  - Buchstaben (Groß-/Kleinbuchstaben) des lateinischen Alphabets A,...,Z,a,...z
  - Ziffern 0,...,9
  - Interpunktion: ? ! . ; , : ...
  - sonstige Zeichen = + - § $ ...

- Auch Sonderzeichen:
  - ä, ö, ü, ß, à, â,... ?
  - griechische Buchstaben
  - arabische Zeichen
  - chinesische Zeichen
  - ...

# Übersicht über die historische Entwicklung

- Die erste weit verbreitete Kodierung von Zeichen ist ASCII (American Standard Code for Information Interchange) aus dem Jahre 1966
- 7 Bits ($2^7$=128 Möglichkeiten) zur Darstellung u.a. lateinischer Buchstaben sowie einiger Kontrollcodes (Zeilenende, Seitenende, Gong,...)
- Erweiterung ISO-8859-1 mit insgesamt 8 Bit ($2^8$=256 Möglichkeiten insgesamt) für europäische Zeichen ä,ö,ü,...
- 8-32 Bit Unicode als einheitliche Kodierung für alle Regionen
- UCS (Universal Character Set) als internationaler Standard der ISO
- Alle diese Kodierungen erweitern die vorherigen Kodierungen (Obermenge)

# ASCII

Oberste 3 Bits

Unterste 4 Bits

| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| **0000** | NUL | DLE | SPACE | 0 | @ | P | _ | p |
| **0001** | SOH | DC1 | ! | 1 | A | Q | a | q |
| **0010** | STX | DC2 | " | 2 | B | R | b | r |
| **0011** | ETX | DC3 | # | 3 | C | S | c | s |
| **0100** | EOT | DC4 | $ | 4 | D | T | d | t |
| **0101** | ENQ | NAK | % | 5 | E | U | e | u |
| **0110** | ACK | SYN | & | 6 | F | V | f | v |
| **0111** | BEL | ETB | " | 7 | G | W | g | w |
| **1000** | BS | CAN | ( | 8 | H | X | h | x |
| **1001** | HAT | EM | ) | 9 | I | Y | i | y |
| **1010** | LF | SUB | * | : | J | Z | j | z |
| **1011** | VT | ESC | + | ; | K | [ | k | { |
| **1100** | FF | FS | , | < | L | \ | l | \| |
| **1101** | CR | GS | - | = | M | ] | m | } |
| **1110** | SO | RS | . | > | N | ^ | n | ~ |
| **1111** | SI | US | / | ? | O | ___ | o | DEL |

# Einige Eigenschaften der ASCII-Kodierung

- Groß- und Kleinbuchstaben sowie Ziffern sind jeweils aufeinanderfolgend in Blöcken angeordnet
- Durch Vergleich von Codes innerhalb eines Blocks kann man lexikografisch kleinere, größere und gleiche Zeichen ermitteln

- Einen Großbuchstaben kann ich in einen Kleinbuchstaben umwandeln, indem ich auf den Code 0x20 addiere
  Beispiel: das Zeichen A hat den Code $100\ 0001_2$=0x41, das Zeichen a hat den Code $110\ 0001_2$ = 0x61 = 0x41+0x20
- Einen Kleinbuchstaben kann ich einen Großbuchstaben umwandeln, indem ich vom Code 0x20 abziehe
  Beispiel: das Zeichen a hat den Code $110\ 0001_2$=0x61, das Zeichen A hat den Code $100\ 0001_2$ = 0x41 = 0x61-0x20
- Zu einer Ziffer kann ich den Wert dieser Ziffer ermitteln, indem ich vom Code den Code der Ziffer 0 abziehe
  Beispiel: Die Ziffer 0 hat den Code $011\ 0000_2$ = 0x30. Die Ziffer 9 hat den Code $011\ 1001_2$=0x39. 0x39 – 0x30 = 9

# ISO-8859-1Erweiterung

- Problem (für uns): In der ASCII-Tabelle erscheinen zum Beispiel keine deutschen Umlaute oder æåôõ...

- Die ISO (International Organization for Standardization; http://www.iso.org/) hat verschiedene regionsspezifische Erweiterungen definiert.

- ISO-8859-1 (manchmal auch ISO Latin-1 genannt): Erweiterung um 128 Zeichen mit den Codes 128-255 für westeuropäische Zeichen (also von 7 Bit ASCII auf insgesamt 8 Bit)

- Für andere Regionen der Erde gab es ebenfalls 128 Erweiterungscodes, man musste sich also entscheiden, welche Erweiterung man nehmen wollte (Codepages)

- Wichtig: die ersten 128 Zeichen haben in jeder Regionserweiterung die gleiche Bedeutung (=ASCII)

# Unicode

- Bekanntlich sind in der Welt mehr als 256 Schriftzeichen bekannt
- Unicode (http//www.unicode.org/) ist eine fortlaufende Entwicklung, alle Schriftzeichen der Welt zu kodieren

- Mit Unicode 1.0 wurden zu Beginn 16 Bits genutzt und damit $2^{16}$=65536 Möglichkeiten definiert

- Mit neueren Unicode-Definitionen (Version 2 aufwärts) wurde weiterer Raum geschaffen (sogenannte Planes) und weitere (eher exotische) Zeichen definiert
- In der aktuellen Version sind alle Unicode-Zeichen mit 21 Bits darstellbar

- Wichtig: die ersten 256 Zeichen in Unicode haben die gleiche Kodierung wie ISO Latin-1 (inklusive ASCII)

# UCS

- UCS (Universal Character Set) ist ein internationaler Standard der ISO, der nahezu identisch mit Unicode ist (und gehalten wird)

- UCS-2 mit 2 Bytes entspricht Unicode 16 Bit (veraltet)
- UCS-4 mit 4 Bytes entspricht den neueren Erweiterungen von Unicode mit mehr als $2^{16}$ Kodierungen (Erweiterungs-Planes). Nur 21 Bits davon sind (derzeit) relevant.

# UTF-8 (und UTF-16, UTF-32)

- UTF-8 (Unicode Transformation Format) ist ein Format auf Basis von Unicode / UCS zur speichereffizienten Darstellung von Zeichen
- Wird genutzt u.a. zur Speicherung von Dokumenten und im Austausch von Daten im Internet (Mail, Browser,...)

- In UTF-8 haben Zeichenkodierungen keine feste Byte-Länge, sondern diese kann je nach Zeichen zwischen 1 und 4 Bytes variieren

- Idee: häufig genutzte Zeichen mit sehr wenigen Bytes kodieren, selten genutzte Zeichen mit mehr Bytes
- Alle ASCII-Zeichen und damit alle lateinischen Groß-/Kleinbuchstaben und Ziffern lassen sich mit 1 Byte speichern

- UTF-16 kodiert in einem oder zwei 16-Bit Wörtern. Geschieht die Kodierung nur in einem16-Bit Wort, so entspricht die Kodierung der von UCS-2.
- UTF-32 kodiert in einem 32-Bit Wort.

# Kodierungsalgorithmus für UTF-8

- Im Folgenden bezeichnet `xxx` die Nutzinformation
- Jedes mit einem 0-Bit beginnende Zeichen enthält in den restlichen 7 Bit den ASCII-Code eines Zeichens: Muster: `0xxx xxxx`.
- Besteht ein UTF-8 Code aus mehreren (n) Bytes, so beginnt das erste Byte mit n 1-Bits und einer 0 und jedes Folgebyte beginnt mit der Bitfolge 10. Beispielmuster: `110x xxxx 10xx xxxx`.
- 2-Byte Codes haben die Form `110x xxxx 10xx xxxx` und ermöglichen die Abspeicherung UCS-2 Codes, die maximal 11 Bits benötigen.
- 3-Byte Codes haben die Form `1110 xxxx 10xx xxxx 10xx xxxx` und ermöglichen die Abspeicherung aller 16 Bit UCS-2 Codes.
- 4-Byte Codes haben die Form `1111 0xxx 10xx xxxx ...` und ermöglichen die Abspeicherung aller UCS-4 Codes.

# Zwischenstand

- Es existieren historisch gewachsen verschiedene Kodierungen für Zeichen
- Diese sind aufeinander aufbauend
- UTF-8 ist ein Komprimierungsverfahren optimiert auf häufig genutzte Zeichen

**Reflektion**

- Welche Probleme treten auf, wenn man in einem Text verschiedene Kodierungen mischt?
- Was könnte/müsste man tun, um diese Probleme zu lösen?

# Einzelne Zeichen in Java

| Datentyp | Anzahl Bits | Anzahl Bytes | Wertebereich |
|---|---|---|---|
| `char` | 16 | 2 | U+0000 bis U+FFFF bzw. 0-65536 |

- Der Datentyp `char` dient zur Darstellung eines einzelnen Zeichens in Java
- Mit der internen Kodierung kommt ein Programmierer (wenn er es nicht explizit möchte) kaum/nicht in Berührung
- Java verwendet UTF-16 mit einem 16-Bit Wort zur (internen) Kodierung von Zeichen

- `char` ist ein Zahlentyp (addieren, multiplizieren,…)
- Nur bei Ein-/Ausgabe und textspezifischen Methoden werden Werte als Repräsentationen von Zeichen interpretiert

# Zeichenliterale

- Ein Literal vom Typ `char` wird angegeben, indem man es in (einfache!) Hochkommata einschließt
- Beispiel: `'a'`, `'9'`, `'!'`
- Ausnahme innerhalb Kochkommas: siehe Tabelle
- U.a. dafür: Escape-Sequenzen der Form `'\x'`

| Angabe | Bedeutung |
|---|---|
| `\'` | Hochkomma selber |
| `\\` | Rückwärtsschrägstrich |
| `\b` | Backspace (Positionierung auf vorangehendes Zeichen) |
| `\t` | Tabulator |
| `\n` | Zeilenende (Line Feed) |
| `\r` | Zeilenende (Carriage Return) |
| `\f` | Seitenende |
| `\"` | Anführungszeichen |
| `\uxxxx` | Hexadezimalcode xxxx des Zeichens |
| `\ooo` | Sequenz von bis zu 3 Oktalziffern (Oktalcode des Zeichens) |

# Beispiel

```
public class CharBeispiel2 {
    public static void main(String[] args) {

        // Definition dreier Variablen vom Typ char
        char c1 = 'a';
        char c2 = '\n';
        char c3 = 'b';

        // Ausgabe der drei Zeichen
        System.out.println("" + c1 + c2 + c3);
    }
}
```

Ausgabe auf Bildschirm:

```
a
b
```

# Zeichen werden intern wie Zahlen behandelt

- Der Compiler wandelt eine Angabe wie `'A'` automatisch in den intern genutzten Unicode um, also einen ganzzahligen Wert
- Der interne Unicode kann ähnlich wie ein int-Wert behandelt werden (der Datentyp `char` gehört auch zu den numerischen Typen!)
- Wenn man zwei Codes voneinander abzieht, bekommt man den Differenzbetrag als int-Wert
  Beispiel: `'9' - '0' = 9`
- Wenn man zu einem Zeichen/Code zum Beispiel einen int-Wert x addiert, bekommt man den Code, der x Position weiter ist.
  Aber: an dieser Stelle muss ich eine Typanpassung (cast) des Resultatwertes vornehmen (Gründe und Details dazu später)
  Beispiel: `(char)(9 + '0')`

# Beispiel

```
public class CharBeispiel3 {

    public static void main(String[] args) {
        // Berechne Distanz zweier Codes
        char c = '9';
        int wert = c - '0';
        System.out.println("Der Wert der Ziffer " + c + " ist " + wert);

        // Rechne mit Code (hier addieren)
        wert = 9;
        c = (char)(wert + '0');
        System.out.println("Die Ziffer zum Wert " + wert + " ist " + c);
    }
}
```

Ausgabe auf Bildschirm:

```
Der Wert der Ziffer 9 ist 9
Die Ziffer zum Wert 9 ist 9
```

# Zwischenstand

- Variablen des Datentyps `char` können genau ein Zeichen aufnehmen
- Zeichenkonstanten werden in Apostrophe eingeschlossen
- Es gibt Escape-Sequenzen für Zeichen „außerhalb der Tastatur“
- Mit Zeichen kann man „rechnen“

**Reflektion**

- Würde ein Ausdruck wie `'A'+0.5` auch Sinn machen? Wieso / wieso nicht?

# Zeichenketten / Strings

- Meist ist man nicht nur an einem einzigen Zeichen interessiert, sondern an einer ganzen Folge von Zeichen
- Eine Folge von Zeichen wird als String bezeichnet
- Ein String kann leer sein, kann genau ein Zeichen enthalten oder viele Zeichen enthalten

- Strings spielen eine wichtige Rolle, insbesondere in der kommerziellen Welt

# Strings in Java

- Achtung: es gibt keinen primitiven Datentyp für Strings in Java
- Es gibt statt dessen eine Klasse mit Namen `String`, über die mit Strings gearbeitet werden kann
- Das hat große Konsequenzen, auf die wir später zurück kommen werden
- In vielerlei Hinsicht verwendet man Strings aber analog zu Daten primitiver Typen

- String-Konstanten: Folge von Zeichen inklusive Escape-Sequenzen in (doppelten) Anführungszeichen
- Beispiele: `"hallo"` , `""`, `"A"` , `"neue Zeile\nist hier"`

- Vorsicht: `"A"` ist vom Typ `String`, `'A'` ist vom Typ `char`

- Deklaration einer Stringvariablen analog zu vorher: `String str;`

# Operationen auf Strings

- Einfachste Operation ist neben der Zuweisung das Aneinanderhängen von Strings mit +
- Beispiel: `"abc" + "def"` ergibt den String `"abcdef"`
- Das Ergebnis ist ein neuer String mit dem zusammengefügten Inhalt der beiden alten Strings!
- Es wird in Java also **nicht** ein existierender String erweitert/verändert!
- Wenn ein Argument des binären Plus-Operators ein String ist, wird das andere Argument automatisch in einen String umgewandelt (aber später auch zu beachten: Priorität von Operatoren)
- Beispiele:
  - `"abc" + 3` ergibt `"abc3"`
  - `3 + "abc"` ergibt `"3abc"`

- Weiterhin gibt es Operationen nur im Zusammenhang mit einem String (Achtung: hier sieht man schon den Unterschied zu primitiven Datentypen)
- Beispiel: `"abc".length()`

# Methoden auf Strings

| Methodenname | Beispiel | Ergebnis | Bedeutung |
|---|---|---|---|
| `equals(str)` | `"abc".equals("bcd")` | `false` | Vergleich zweier Strings (kein == ) |
| `charAt(i)` | `"abc".charAt(1)` | `'b'` | Zeichen an der i-ten Position |
| `indexOf(c)` | `"abc".indexOf('b')` | `1` | erste Position des Zeichens |
| `length()` | `"abc".length()` | `3` | Länge des Strings in Zeichen |
| `substring(x,y)` | `"abc".substring(1,2)` | `"b"` | Teilstring von Position x bis y |

- Alle Methoden nur im Zusammenhang mit einem String (siehe Beispiele oben)
- Statt Stringkonstanten können natürlich auch Stringvariablen dabei genutzt werden (allgemein: beliebige Ausdrücke vom Typ `String`)
- Positionen beginnen bei 0
- Fehlerquelle: Strings können nicht mit == **auf gleichen Inhalt** verglichen werden
- Viele weitere Stringmethoden vorhanden (siehe Dokumentation Klasse `String` in der API-Dokumentation)

# Beispiel

```
public class StringBeispiel3 {
    public static void main(String[] args) {

        // Definition einer Variablen vom Typ String
        String s = "01234567689";

        // bestimme Laenge des Strings
        int len = s.length();

        // extrahiere den Teilstring ohne erstes und letztes Zeichen
        s = s.substring(1,len-1);

        // fuege vorne und hinten Zeichen an
        s = "->" + s + "<-";

        System.out.println(s);
    }
}
```

Ausgabe auf Bildschirm:

```
->123456768<-
```

# Zwischenstand

- Der Datentyp `String` ist kein primitiver Datentyp
- Variablen des Datentyps `String` können beliebig viele Zeichen aufnehmen inklusive leerer String
- Stringkonstanten werden zwischen Anführungszeichen angegeben
- Der + Operator mit 2 Strings erzeugt einen neuen String mit den aneinandergehängten Inhalten der beiden Argument-Strings
- In der Klasse String gibt es viele Methoden zur Verarbeitung von Strings

**Reflektion**

- Welche Vor-/Nachteile könnte es haben, wenn beim + Operator ein neuer String entsteht? Alternativ könnte man ja auch das erste Argument verändern?

# Zusammenfassung

- Datentyp `char` für ein Zeichen, die Klasse `String` für Zeichenketten
- Kodierung von Zeichen durchgehend in Unicode
- Zahlreiche Operationen auf Strings über Methoden möglich

- Steckbrief zu Zeichen und Zeichenketten

| Name | `char` |
|---|---|
| Wertemenge | Zeichen |
| Kodierung | UCS-2 |
| Konstanten | ein Zeichen in Apostrophen `'...'` |
| Operationen | Zuweisung, Vergleichsoperatoren, Zeichenmethoden, arithmetische Operationen wie `int` (automatische Umwandlung) |
| Nutzung | für genau ein Zeichen |
| Besonderheiten | |

| Name | `String` |
|---|---|
| Wertemenge | Zeichenfolgen |
| Kodierung | UTF-16 |
| Konstanten | beliebig viele Zeichen in Anführungszeichen `"..."` |
| Operationen | Zuweisung, +, Stringmethoden |
| Nutzung | für Zeichenketten |
| Besonderheiten | kein primitiver Datentyp |

# Inhalt dieser Veranstaltung

1. Einführung, Algorithmenbegriff
2. Programmiersprachen
3. Grundlegende Daten mit den darauf definierten Operationen
   1. Grundlagen
   2. Wahrheitswerte
   3. ganze Zahlen
   4. Fließkommazahlen
   5. Zeichen und Zeichenketten
   6. Ein-/Ausgabe
4. Kontrollstrukturen, Variablen, Ausdrücke
5. Methoden
6. Komplexere Datentypen
7. Abstrakte Datentypen
8. Einführung in die Objektorientierte Programmierung
9. Einführung in das Testen und Verifizieren
10. Programmierparadigmen und Typkonzepte

# Einfache Ein-/Ausgabe von Daten

- Bisher:
  - Ausgabe von Daten mit `System.out.println(...);`
  - Eingabe von Daten: -

- Jetzt: wie kann man in einem Programmlauf Daten einlesen, die zur Übersetzungszeit noch nicht vorhanden sind?

- Zwei einfache Antworten für diese Veranstaltung:
  - Daten über die Tastatur eingeben bei Aufruf des Programms
    Formulierung in Aufgaben: „...in der Kommandozeile übergeben…“
  - Daten über die Tastatur eingeben, während das Programm läuft
    Formulierung in Aufgaben: „…von der Tastatur einlesen…“

- Java kennt das umfangreiche Stream-Konzept, mit dem sich flexibel Daten ein- und ausgeben lassen (Dateien, Netzwerk, Webseiten,…; nicht in dieser Veranstaltung)

# Daten bei Programmstart übergeben

- Beim Start eines Programms kann man in der Kommandozeile Daten an das Programm übergeben
- Formulierung dazu in Aufgaben / Prüfung: **in der Kommandozeile**
- Dies geschieht implizit in eclipse
- Bei Start eines Java-Programms
  - in der Kommandozeile: `java MeinProgramm 1 2.0 drei`
  - In eclipse über Menü `Run`, dann `Run Configurations`, dann `(x)=Arguments`

- Im Programm kann man auf diese Daten als Strings zugreifen
- Dazu dient `args` in `public static void main(String[] args)`
- `args` ist ein Vektor (später: ein Feld) von Strings, `args[i]` der i-te String

- Umwandeln eines Strings in einen Wert (falls Syntax korrekt) geschieht über spezielle Umwandlungsmethoden (siehe nachfolgendes Beispiel)
- Man muss dabei den Zieltyp wissen

# Beispiel

```
/* Aufruf des Programms zum Beispiel mit:
      java Kommandozeilenargumente 1 2 3 4 5.7  3.1  x  str
   Das i-te Argument muss mit der entsprechenden Operation lesbar sein
*/
public class Kommandozeilenargumente {
    public static void main(String[] args) {
        // 1. Argument muss eine ganze Zahl im Wertebereich byte sein
        byte b = Byte.parseByte(args[0]);
        // 2. Argument muss eine ganze Zahl im Wertebereich short sein
        short s = Short.parseShort(args[1]);
        // 3. Argument muss eine ganze Zahl im Wertebereich int sein
        int i = Integer.parseInt(args[2]);
        // 4. Argument muss eine ganze Zahl im Wertebereich long sein
        long l = Long.parseLong(args[3]);
        // 5. Argument muss eine Fließkommazahl im Wertebereich float sein
        float f = Float.parseFloat(args[4]);
        // 6. Argument muss eine Fließkommazahl im Wertebereich double sein
        double d = Double.parseDouble(args[5]);
        // 7. Argument muss ein Character sein
        char c = args[6].charAt(0);
        // 8. Argument muss ein String sein
        String str = args[7];
    }
}
```

# Einlesen von Daten während des Programmlaufs

- An dieser Stelle pragmatische Erklärung, weniger Hintergrunddetails
- Formulierung in Aufgaben / Prüfung: **von der Tastatur (zur Laufzeit)**

- Mit einem Programm sind zwei Ein- und Ausgabedateien vordefiniert
- Für die Ausgabe: `System.out`
- Für die Eingabe: `System.in`

- Mit Hilfe eines `Scanner` lassen sich darüber Daten von der Tastatur während eines Programmlaufs einlesen

- Zu beachten: die Leseoperationen sind blockierend (sie warten, bis die Eingabe erfolgt ist, abgeschlossen durch ↵ `/ Return`)

# Beispiel

```
import java.util.*;         // hier kommt der Scanner her
public class EingabeTastatur {
    public static void main(String[] args) {
        // Scanner von der Tastatur anlegen
        Scanner sc = new Scanner(System.in);

        System.out.println("Geben Sie Daten folgender Typen auf der Tastatur ein: "
                           + "byte, short, int, long, float, double, "
                           + "char (eigene Zeile), String (eigene Zeile)");
        // Daten ueber den Scanner lesen
        byte b = sc.nextByte();
        short s = sc.nextShort();
        int i = sc.nextInt();
        long l = sc.nextLong();
        float f = sc.nextFloat();
        double d = sc.nextDouble();
        String s1 = sc.next();
        char c = s1.charAt(0);
        String s2 = sc.next();
        // eingelesene Daten ausgeben
        System.out.println("Die eingelesenen Werte sind: " + b + " "
                + s + " " + i + " " + l + " " + f + " " + d + " " + c + " " + s2);
        // Scanner abschliessen
        sc.close();
    }
}
```

# Zusammenfassung

- Zwei sehr einfache Möglichkeiten der Eingabe von Daten:
  - Beim Programmstart über die Kommandozeile
  - Zur Laufzeit des Programms über `System.in` und einen Scanner

- Später sollten sinnvollerweise die erweiterten Möglichkeiten der Streams und weiterer Ansätze in Java genutzt werden