Universität Bremen

# Praktische Informatik 1
# Datentypen und Bits


Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

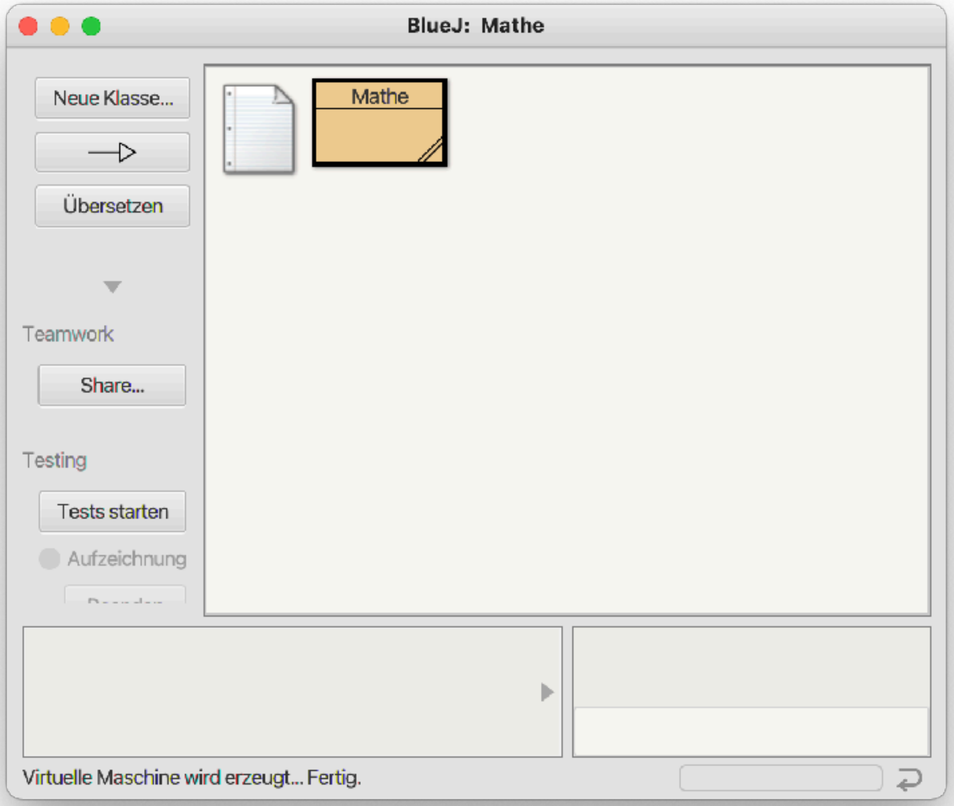
Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen



Bild: ©Michael Ihle

Universität Bremen

# Grenzen primitiver Datentypen: Demo



Universität Bremen

# Zahlensysteme

- Endlich (**β**) viele Ziffern, z.B.
  - Dezimalsystem: **0 … 9**, **β = 10**
  - Dual/Binärsystem: **0 … 1**, **β = 2**
- **β** wird auch Radix genannt
- Schreibung einer Zahl: $z_{n-1} \dots z_0$, $0 \leq z_i < \beta$
- Wert der Zahl: $z = \sum_{i=0}^{n-1} z_i \beta^i$

# Zahlensysteme: Dezimalsystem ↔ Dualsystem

- $42_{10} = 4_{10} \cdot 10^1{}_{10} + 2_{10} \cdot 10^0{}_{10}$

  $= 100_2 \cdot 1010^1{}_2 + 10_2 \cdot 1010^0{}_2$

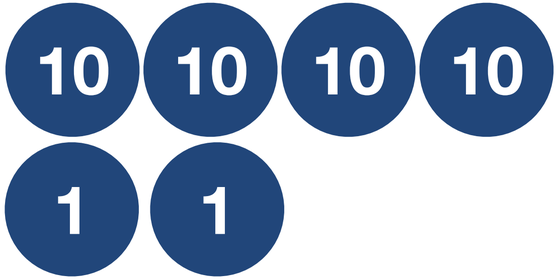
  $= 101000_2 + 10_2$

  $= 101010_2$

- $101010_2 = 1_2 \cdot 10^5{}_2 + 1_2 \cdot 10^3{}_2 + 1_2 \cdot 10^1{}_2$

  $= 1_{10} \cdot 2^5{}_{10} + 1_{10} \cdot 2^3{}_{10} + 1_{10} \cdot 2^1{}_{10}$

  $= 32_{10} + 8_{10} + 2_{10} = 42_{10}$





| Dezimal | Dual |
|---|---|
| 0 | ...0000 |
| 1 | ...0001 |
| 2 | ...0010 |
| 3 | ...0011 |
| 4 | ...0100 |
| 5 | ...0101 |
| 6 | ...0110 |
| 7 | ...0111 |
| 8 | ...1000 |
| 9 | ...1001 |

Universität Bremen

# Zahlensysteme: Algorithmus Dezimal → Dual

1. Wenn die Dezimalzahl gerade ist, schreibe eine **0**, sonst eine **1** (von rechts nach links)

2. Teile die Dezimalzahl durch **2** (mit Abrunden)

3. Wenn die Dezimalzahl noch ungleich **0**, mache bei Schritt 1 weiter

4. Alternativ für feste Anzahl von Dualziffern:
   Wenn noch nicht erforderliche Anzahl Dualziffern erreicht, mache bei Schritt 1 weiter

| Dual | Dezimal |
|---|---|
| | 13 |
| **1** | 6 |
| **0**1 | 3 |
| **1**01 | 1 |
| **1**101 | 0 |
| **0**1101 | 0 |

# Zahlensysteme: Algorithmus Dual → Dezimal

- Die Dezimalzahl ist anfangs **0**
- Solange noch Binärziffern da sind
  - Multipliziere die Dezimalzahl mit **2**
  - Addiere die linke Binärziffer hinzu
  - Streiche die linke Binärziffer weg

| Dezimal | Dual |
|---|---|
| 0 | **0**1101 |
| 0 | ~~0~~**1**101 |
| 1 | ~~01~~**1**01 |
| 3 | ~~011~~**0**1 |
| 6 | ~~0110~~**1** |
| 13 | ~~01101~~ |

Universität Bremen

# Negative Zahlen: Vorzeichenbit / Einerkomplement

- Vorzeichenbit
  - MSB für Vorzeichen reservieren
  - **0** → positiv, **1** → negativ
  - **$-3_{10} = -0011_2 \equiv \underline{1}011_2$**
- Einerkomplement
  - Alle Bits invertieren
  - **$-3_{10} = -0011_2 \equiv \underline{1}100_2$**

Universität Bremen

# Negative Zahlen: Zweierkomplement

- Alle Bits invertieren und eins dazuzählen
  - $-3_{10} = -0011_2 \equiv \underline{1}101_2$
- Entspricht: $16_{10} + -3_{10} = 13_{10} = 1101_2$
- Vorteil: $5_{10} - 3_{10} = 5_{10} + -3_{10}$

  $\equiv 0101_2 + 1101_2$

  $= \underline{1}0010_2$
- Grund: $= 5_{10} + (16_{10} - 3_{10}) = 16_{10} + (5_{10} - 3_{10})$

# Modulo-Arithmetik

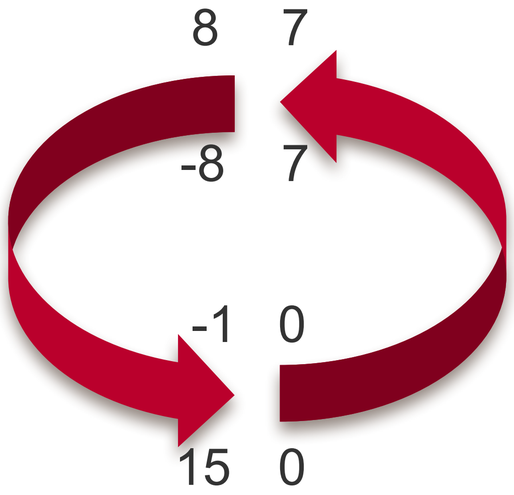
- $(5_{10} - 3_{10}) \bmod 2^4{}_{10}$

  $= (5_{10} + -3_{10}) \bmod 2^4{}_{10}$

  $= (0101_2 + 1101_2) \bmod 10000_2$

  $= 10010_2 \bmod 10000_2$

  $= 0010_2$



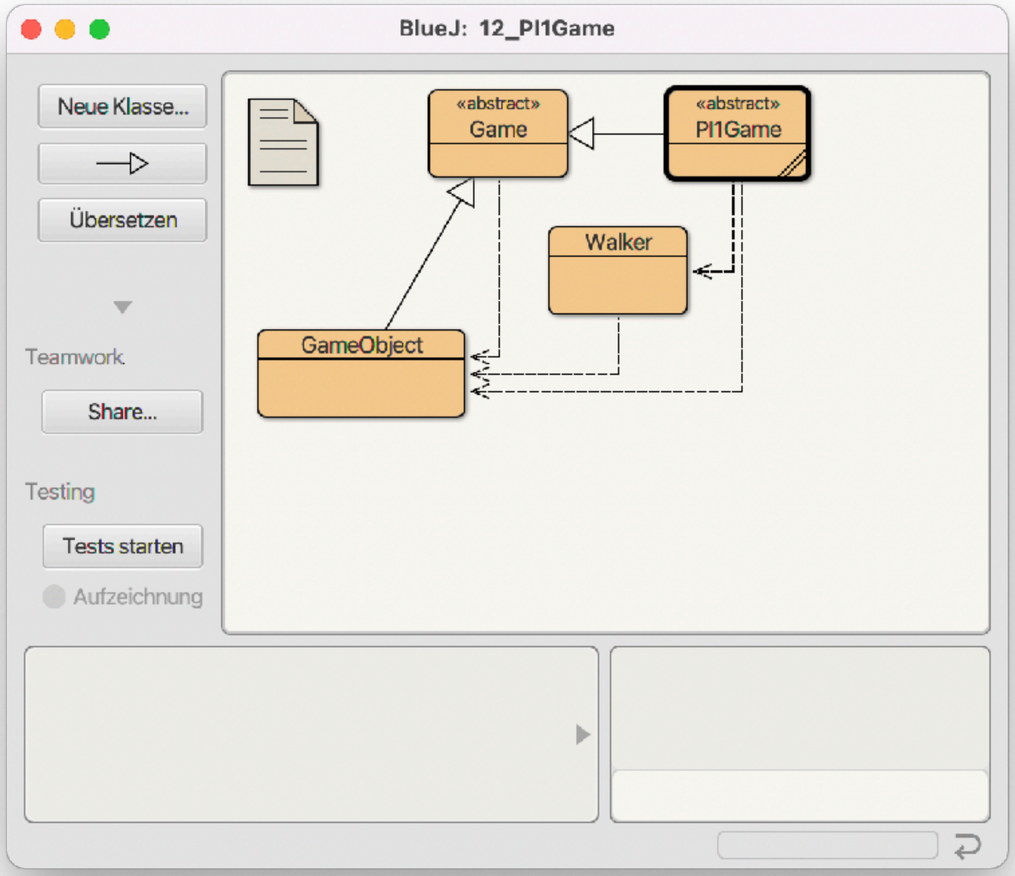

| Dezimal | Zweierkomplement |
|---|---|
| +8 | nicht darstellbar |
| +7 | 0111 |
| +6 | 0110 |
| +5 | 0101 |
| +4 | 0100 |
| +3 | 0011 |
| +2 | 0010 |
| +1 | 0001 |
| 0 | 0000 |
| −1 | 1111 |
| −2 | 1110 |
| −3 | 1101 |
| −4 | 1100 |
| −5 | 1011 |
| −6 | 1010 |
| −7 | 1001 |
| −8 | 1000 |

# Gleitkommazahlen (IEEE 754)

| | | | |
|---|---|---|---|
| float | v | 30 - Exponent - 23 | 22 - Mantisse - 0 |
| | 1 Bit | 8 Bit | 23 Bit |
| double | v | 62 - Exponent - 52 | 51 - Mantisse - 0 |
| | 1 Bit | 11 Bit | 52 Bit |

- Darstellung: Mantisse und Exponent
  - **$z = (-1)^v \cdot \text{Mantisse} \cdot 2^{\text{Exponent}}$**
  - Die Mantisse hat eigentlich ein Bit mehr, da die führende **1** nicht gespeichert wird
- Spezielle Werte: **+0**, **-0**, **-∞** (z.B. **-1/0**), **+∞** (z.B. **+1/0**), **NaN** (z.B. **0/0**, **$\sqrt{-1}$**)
- Vorteile
  - Erlauben Nachkommastellen
  - Decken sehr großen Zahlenbereich ab
- Nachteile: Abdeckung ist lückenhaft → Rundungsfehler (**2e7f + 1 == 2e7f**)

# Fließkomma-Genauigkeit: Demo



Universität Bremen

# Primitive Datentypen in Java

| Datentyp | Standard | Speicherplatz | Wertebereich |
|---|---|---|---|
| byte | 0 | 1 Byte | -128 bis 127 |
| short | 0 | 2 Bytes | -32768 bis 32767 |
| int | 0 | 4 Bytes | -2147483648 bis 2147483647 |
| long | 0L | 8 Bytes | -9223372036854775808L bis 9223372036854775807L |
| float | 0.0F | 4 Bytes | ±1.40239846E-45F bis ±3.40282347E+38F |
| double | 0.0 | 8 Bytes | ±4.94065645841246544E-324 bis ±1.79769313486231570E+308 |
| boolean | false | ? | false, true |
| char | '\u0000' | 2 Bytes | '\u0000' bis '\uFFFF', bzw. 0 bis 65535 |

Universität Bremen

# Typumwandlungen

- Automatisch (eigentlich direkt, z.B. **byte** → **double**)
  - **byte** → **short** → **int** → **long** → **float** → **double**
    **char** ↗
- Manuell (Expliziter **Typecast**)
  - Bei der Typumwandlung von **double** oder **float** in einen Ganzzahltyp werden die Nachkommastellen abgeschnitten
- Durch Funktionen

```
float f = 1.5F;
double d = -1.5;
byte b = (byte) f; // b == 1
f = (float) 178.2;
long l = (int) d; // l == -1
char c = (char) 32;
```

```
int i = Integer.parseInt(s);
String s = Integer.toString(i);
d = Double.parseDouble(s);
```

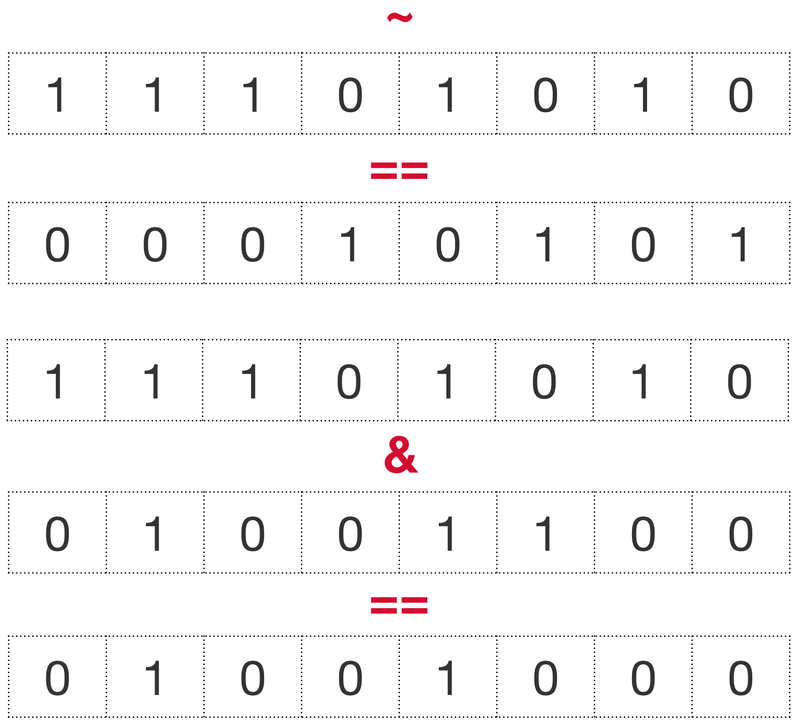
Universität Bremen

# Weitere Zahlensysteme

- Hexadezimalsystem (Literale in Java z.B. **0xf**, **0xABCDEF00**, **0x1234**)
  - Radix **β = 16**, Ziffern **0** … **9**, **A**, **B**, **C**, **D**, **E**, **F**
  - Eine Hexadezimalziffer entspricht 4 Bit
  - Ein Byte kann durch 2 Ziffern dargestellt werden
- Oktalsystem (Literale in Java z.B. **077**, **01234567**, **0123**)
  - Radix **β = 8**, Ziffern **0** … **7**
  - Eine Oktalziffer entspricht 3 Bit
  - Wird eher selten verwendet (z.B. Unix-Zugriffsrechte)
- Java unterstützt auch Binärzahl-Literale, z.B. **0b1**, **0b010**, **0b100**, **0b1000**

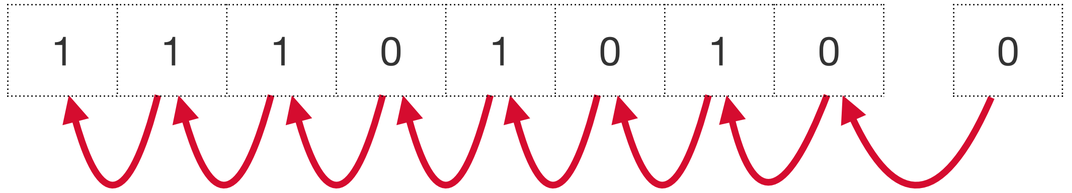
# Bit-Verknüpfungen

- Ganzzahltypen können als Menge von Bits betrachtet werden
- Logische Operatoren verknüpfen diese Mengen ähnlich wie beim Typ **boolean**
- Nicht: **~0xea**
- Und: **0xea & 0x4c**
- Oder: **0xea | 0x4c**
- Exklusiv-Oder: **0xea ^ 0x4c**
  (Bit-weises **!=**)

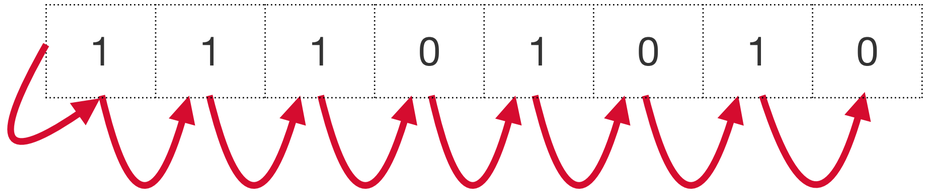


# Bit-Verschiebungen

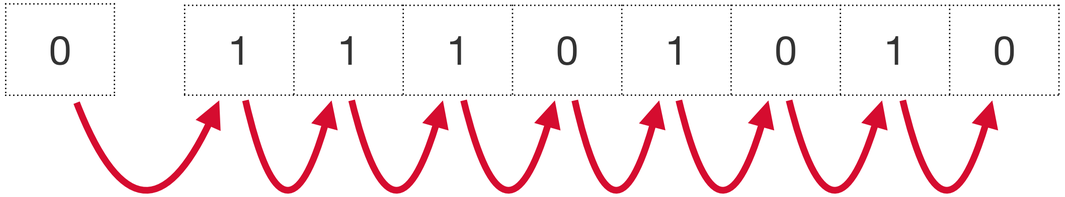
- Verschieben den linken Operanden um die rechts angegebene Anzahl von Bits
- Links: **0xea << 1**




- Rechts: **0xea >> 1**

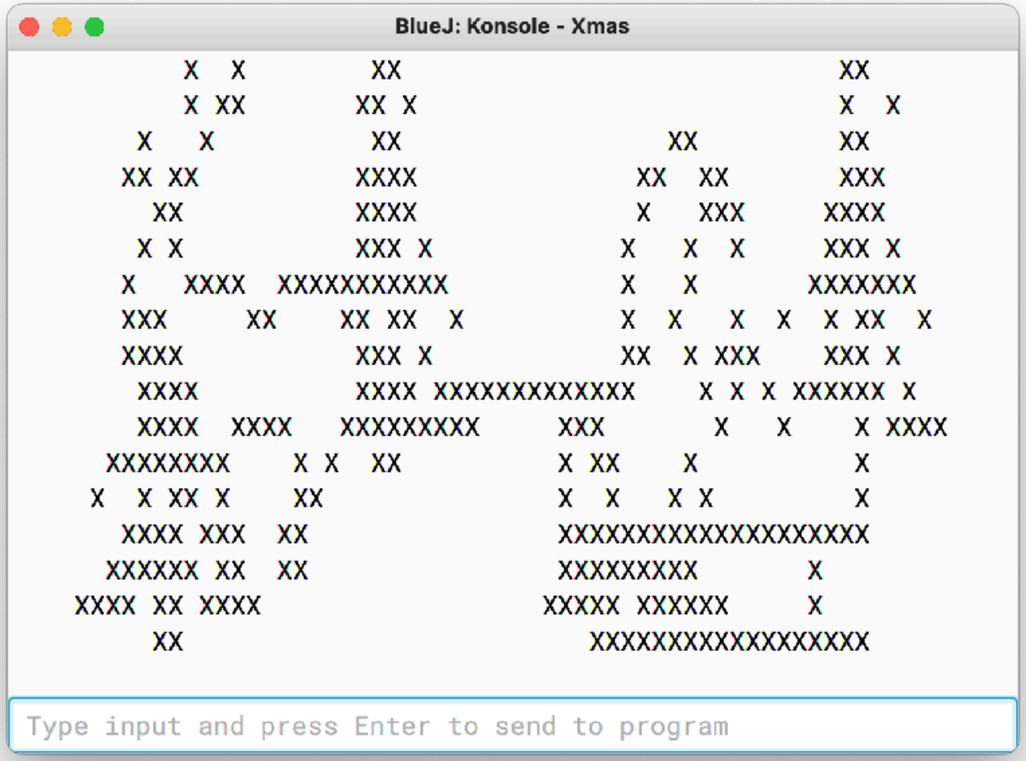



- Vorzeichenlos rechts: **0xea >>> 1**



Universität Bremen

# Bitverknüpfungen: Demo



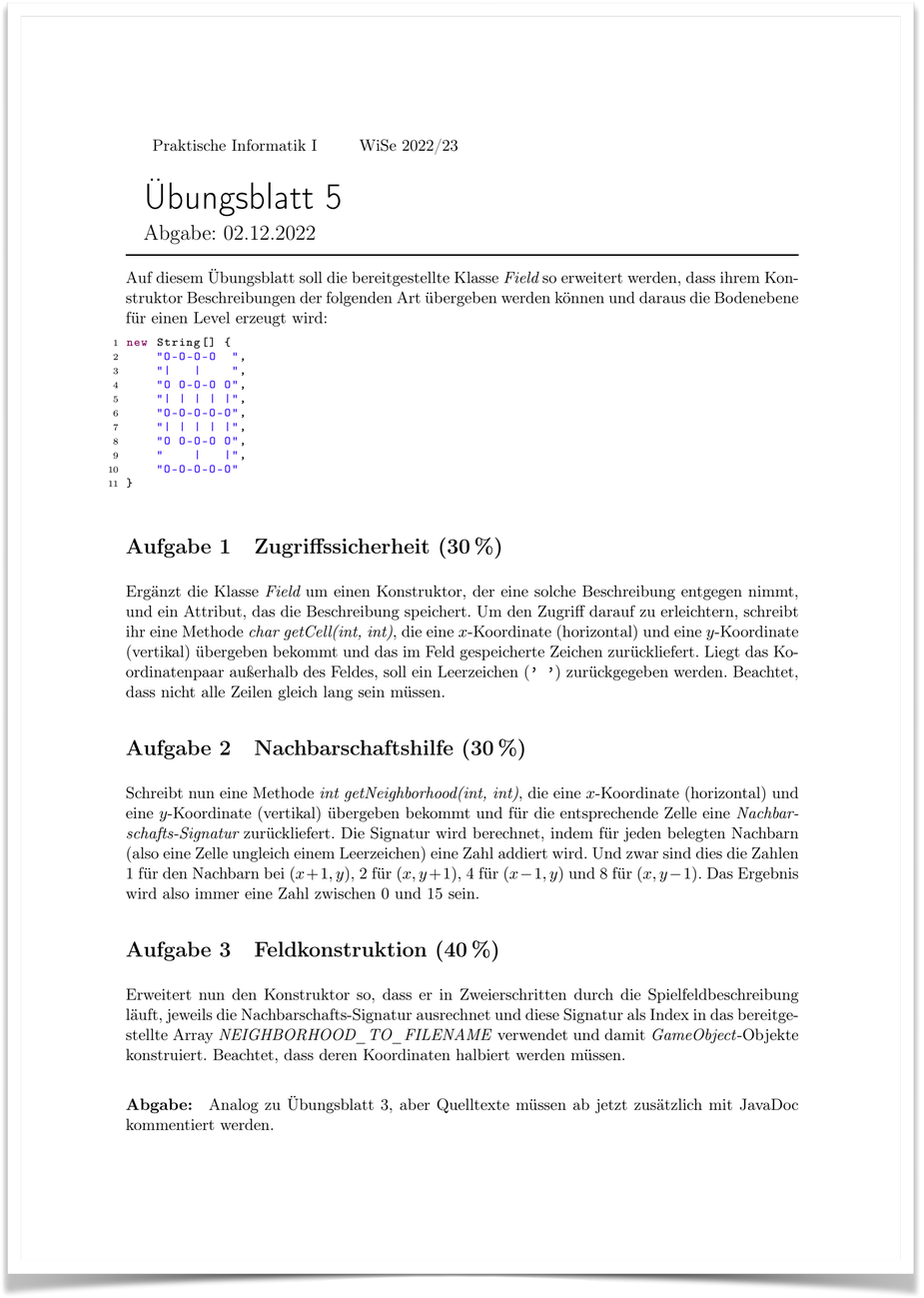
Universität Bremen

# Zusammenfassung der Konzepte

- **Zahlensysteme**, **Radix**
- Umwandlung **Dezimalsystem** ↔ **Dualsystem**
- **Einerkomplement**, **Zweierkomplement**
- **Wertebereiche** von primitiven Datentypen
- **Typumwandlung**
- Bit-Operationen **~**, **&**, **|**, **^**, **<<**, **>>**, **>>>**

# Übungsblatt 5

- Aufgabe 1: Grenzprüfung und Zugriff auf Array- und String-Element
- Aufgabe 2: Eigentlich Bitmaske erzeugen
  - Dies kann auch kurz mit Array und Schleife erledigt werden
- Aufgabe 3: Zwei ineinander geschachtelte Schleifen in Zweierschritten
- Quelltexte (auch) mit JavaDoc kommentieren
- Bewertet!



Praktische Informatik I WiSe 2022/23

## Übungsblatt 5

Abgabe: 02.12.2022

Auf diesem Übungsblatt soll die bereitgestellte Klasse *Field* so erweitert werden, dass ihrem Konstruktor Beschreibungen der folgenden Art übergeben werden können und daraus die Bodenebene für einen Level erzeugt wird:

```
new String[] {
    "O-O-O-O  ",
    "|   |    ",
    "O O-O-O O",
    "| | | | |",
    "O-O-O-O-O",
    "| | | | |",
    "O O-O-O O",
    "   |   |",
    "O-O-O-O-O"
}
```

### Aufgabe 1 Zugriffssicherheit (30 %)

Ergänzt die Klasse *Field* um einen Konstruktor, der eine solche Beschreibung entgegen nimmt, und ein Attribut, das die Beschreibung speichert. Um den Zugriff darauf zu erleichtern, schreibt ihr eine Methode *char getCell(int, int)*, die eine $x$-Koordinate (horizontal) und eine $y$-Koordinate (vertikal) übergeben bekommt und das im Feld gespeicherte Zeichen zurückliefert. Liegt das Koordinatenpaar außerhalb des Feldes, soll ein Leerzeichen (' ') zurückgegeben werden. Beachtet, dass nicht alle Zeilen gleich lang sein müssen.

### Aufgabe 2 Nachbarschaftshilfe (30 %)

Schreibt nun eine Methode *int getNeighborhood(int, int)*, die eine $x$-Koordinate (horizontal) und eine $y$-Koordinate (vertikal) übergeben bekommt und für die entsprechende Zelle eine *Nachbarschafts-Signatur* zurückliefert. Die Signatur wird berechnet, indem für jeden belegten Nachbarn (also eine Zelle ungleich einem Leerzeichen) eine Zahl addiert wird. Und zwar sind dies die Zahlen 1 für den Nachbarn bei $(x+1, y)$, 2 für $(x, y+1)$, 4 für $(x-1, y)$ und 8 für $(x, y-1)$. Das Ergebnis wird also immer eine Zahl zwischen 0 und 15 sein.

### Aufgabe 3 Feldkonstruktion (40 %)

Erweitert nun den Konstruktor so, dass er in Zweierschritten durch die Spielfeldbeschreibung läuft, jeweils die Nachbarschafts-Signatur ausrechnet und diese Signatur als Index in das bereitgestellte Array *NEIGHBORHOOD_TO_FILENAME* verwendet und damit *GameObject*-Objekte konstruiert. Beachtet, dass deren Koordinaten halbiert werden müssen.

**Abgabe:** Analog zu Übungsblatt 3, aber Quelltexte müssen ab jetzt zusätzlich mit JavaDoc kommentiert werden.