Educational Objectives

- You have a good understanding how a computer represents numbers.
- You can transform integers in binary representation and perform computations.
- You understand how the value range of integers is chosen.
- You can describe the representation of floating-point numbers in general.
- You know the three floating-point rules.
- You can use booleans and boolean expressions in Java.

4. Number Representations

Domain of Types int, float and double Mixed Expressions and Conversion; Holes in the Domain; Floating Point Number Systems; IEEE Standard; Limits of Floating Point Arithmetics; Floating Point Guidelines

Binary Number Representations

Binary representation ("Bits" from \{0, 1\})

\[ b_n b_{n-1} \cdots b_1 b_0 \]

corresponds to the number

\[ b_n \cdot 2^n + \cdots + b_1 \cdot 2 + b_0 \]

Example: 101011 corresponds to 43.

Binary Numbers: Numbers of the Computer?

Truth: Computers calculate using binary numbers.
Binary Numbers: Numbers of the Computer?

Stereotype: computers are talking 0/1 gibberish

Computing Tricks

- Estimate the orders of magnitude of powers of two:\(^2\):
  \[
  2^{10} = 1024 = 1\text{Ki} \approx 10^3.
  \]
  \[
  2^{20} = 1\text{Mi} \approx 10^6,
  \]
  \[
  2^{30} = 1\text{Gi} \approx 10^9,
  \]
  \[
  2^{32} = 4 \cdot (1024)^3 = 4\text{Gi} \approx 4 \cdot 10^9.
  \]
  \[
  2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.
  \]

Definition: **Domain**

For numeric types the domain defines the numeric interval a the type can cover.

Domain of Type `int`

```java
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.

Where do these numbers come from?
Domain of the Type `int`

Representation with 32 bits. Domain comprises the $2^{32}$ integers:

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

Where does this partitioning come from?

---

Computing with Binary Numbers (4 digits)

**Simple Addition**

\[
\begin{array}{c@{\quad}c@{\qquad}c@{\qquad}c}
2 & 0010 \\
+3 & +0011 \\
\hline
5 & 0101
\end{array}
\]

**Simple Subtraction**

\[
\begin{array}{c@{\quad}c@{\qquad}c@{\qquad}c}
5 & 0101 \\
-3 & -0011 \\
\hline
2 & 0010
\end{array}
\]

---

Computing with Binary Numbers (4 digits)

**Addition with Overflow**

\[
\begin{array}{c@{\quad}c@{\qquad}c@{\qquad}c}
7 & 0111 \\
+9 & +1001 \\
\hline
16 & (1)0000
\end{array}
\]

**Negative Numbers?**

\[
\begin{array}{c@{\quad}c@{\qquad}c@{\qquad}c}
5 & 0101 \\
+(-5) & ???:? \\
\hline
0 & (1)0000
\end{array}
\]

---

Computing with Binary Numbers (4 digits)

**Simpler -1**

\[
\begin{array}{c@{\quad}c@{\qquad}c@{\qquad}c}
1 & 0001 \\
+(-1) & 1111 \\
\hline
0 & (1)0000
\end{array}
\]

**Utilize this:**

\[
\begin{array}{c@{\quad}c@{\qquad}c@{\qquad}c}
3 & 0011 \\
+? & +???? \\
\hline
-1 & 1111
\end{array}
\]
Computing with Binary Numbers (4 digits)

Invert!

\[
\begin{align*}
3 & \quad 0011 \\
+(-4) & \quad +1100 \\
\hline
-1 & \quad 1111 \equiv 2^B - 1
\end{align*}
\]

Negation: inversion and addition of 1

\[-a \equiv \bar{a} + 1\]

Wrap around semantics (calculating modulo \(2^B\))

\[-a \equiv 2^B - a\]

Why this works

Modulo arithmetics: Compute on a circle\(^3\)

\[
\begin{align*}
11 & \equiv 23 \equiv -1 \equiv \ldots \mod 12 \\
4 & \equiv 16 \equiv \ldots \mod 12 \\
3 & \equiv 15 \equiv \ldots \mod 12
\end{align*}
\]

\(^3\)The arithmetics also work with decimal numbers (and for multiplication).

Negative Numbers (3 Digits)

<table>
<thead>
<tr>
<th>a</th>
<th>-a</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>111</td>
</tr>
<tr>
<td>010</td>
<td>110</td>
</tr>
<tr>
<td>011</td>
<td>101</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

The most significant bit decides about the sign.
Two’s Complement

- Negation by bitwise negation and addition of 1
  
  
  \[-2 = -[0010] = [1101] + [0001] = [1110]\]

- Arithmetic operations (+, -, *) can lead to numbers outside the valid domain.
- Results can be incorrect!

\[
\text{power8: } 15^8 = -1732076671
\]

\[
\text{power20: } 3^{20} = -808182895
\]

- There is no error message!

Over- and Underflow

- Intuitive “wrap-around” conversion of negative numbers.
  
  \[-n \rightarrow 2^B - n\]

- Domain: \(-2^{B-1} \ldots 2^{B-1} - 1\)

Definition: Floating Point Numbers

Floating point numbers represent numbers from \(\mathbb{R}\) with a fixed number of significant number of digits, multiplied by a decimal power (base 10).

Book, on page 67

“Proper Calculation”

```java
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");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Out.print("Celsius: ");
        float celsius = In.readInt();
        
        Out.print("Celsius: ");
        
```
**Types float and double**

- are the fundamental types for floating point numbers
- approximate the field of real numbers \((\mathbb{R}, +, \times)\) from mathematics
- have a great domain, sufficient for many applications (double provides more places than float)
- are fast on many computers

**Fixpoint numbers**

- fixed number of integer places (e.g. 7)
- fixed number of decimal places (e.g. 3)

0.0824 = 0000000.082\(\overset{\text{third place truncated}}{\rightarrow}\)

**Nachteile**

- Domain is getting even smaller than for integers.
- If a number can be represented depends on the position of the comma.

**Floating point Numbers**

- fixed number of significant places (e.g. 10)
- plus position of the comma

\[
82.4 = 824 \cdot 10^{-1}
\]

\[
0.0824 = 824 \cdot 10^{-4}
\]

- Zahl ist  \textit{Mantissa} \times 10^{\textit{Exponent}}

**Domain**

**Integer Types:**

- Over- and Underflow relatively frequent, but ...
- the domain is contiguous (no “holes”): \(\mathbb{Z}\) is “discrete”.

**Floating point types:**

- Overflow and Underflow seldom, but ...
- there are holes: \(\mathbb{R}\) is “continuous”.

Holes in the Domain

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);
    }
}

Floating Point Rules

Rule 1
Do not test rounded floating point numbers for equality.

for (float i = 0.1; i != 1.0; i += 0.1){
    Out.println(i);
}
endless loop because i never becomes exactly 1

More explanations next time!

Floating Point Rules

Rule 2
Do not add two numbers providing very different orders of magnitude!

Floating Point Guidelines

Rule 3
Do not subtract two numbers with a very similar value.
Cancellation problems (without further explanations).

Floating Point Guidelines

Rule 4
Do not subtract two numbers with a very similar value.
Cancellation problems (without further explanations).
5. Logical Values

Boolean Functions; the Type boolean; logical and relational operators; shortcut evaluation

Our Goal

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

Behavior depends on the value of a Boolean expression

Boolean Values in Mathematics

Boolean expressions can take on one of two values:

- \( F \) or \( T \)

- \( F \) corresponds to "wrong"
- \( T \) corresponds to "true"

The Type boolean in Java

- represents logical values
- Literals false and true
- Domain \{false, true\}

```java
boolean b = true; // Variable with value true
```
Relational Operators

- $a < b$ (smaller than)
- $a \geq b$ (greater than)
- $a == b$ (equals)
- $a != b$ (unequal)

number type × number type → boolean

Boolean Functions in Mathematics

- Boolean function
  
  \[ f : \{F, T\}^2 \rightarrow \{F, T\} \]
  
  - $F$ corresponds to “false”.
  - $T$ corresponds to “true”.

AND($x, y$) $x \land y$

- “logical and”
  
  \[ f : \{F, T\}^2 \rightarrow \{F, T\} \]
  
  - $F$ corresponds to “false”.
  - $T$ corresponds to “true”.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>AND($x, y$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$F$</td>
<td>$T$</td>
<td>$F$</td>
</tr>
<tr>
<td>$T$</td>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Logischer Operator &&

- $a \&\& b$ (logical and)

int n = -1;
int p = 3;
boolean b = (n < 0) && (0 < p); // b = true
OR($x, y$) \hspace{1cm} x \lor y

**Logical Operator | |**

- **“logical or”**
- \( f : \{F, T\}^2 \rightarrow \{F, T\} \)
- \( F \) corresponds to “false”.
- \( T \) corresponds to “true”.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( \text{OR}(x, y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
</tbody>
</table>

Logical Operator \( \lor \) (logical or)

boolean \times boolean \rightarrow boolean

int \( n = 1; \)
int \( p = 0; \)
boolean \( b = (n < 0) \lor (0 < p); // b = false \)

NOT($x$) \hspace{1cm} \neg x

**Logical Operator !**

- **“logical not”**
- \( f : \{F, T\} \rightarrow \{F, T\} \)
- \( F \) corresponds to “false”.
- \( T \) corresponds to “true”.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \text{NOT}(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
</tr>
</tbody>
</table>

Logical Operator \( \neg \) (logical not)

boolean \rightarrow boolean

int \( n = 1; \)
boolean \( b = !(n < 0); // b = true \)
### Precedences

The unary logical operator `!` provides a stronger binding than binary arithmetic operators. These bind stronger than relational operators, and these bind stronger than binary logical operators.

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

Some parentheses on the previous slides were actually redundant.

### DeMorgan Rules

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

```
! (rich and beautiful) == (poor or ugly)
```

### Application: either ... or (XOR)

- `(x || y) && !(x && y) x or y, and not both`
- `(x || y) && (!x || !y) x or y, and one of them not`
- `!(!x && !y) && !(x && y) not none and not both`
- `!(!x && !y || x && y) not: both or none`
Shortcut Evaluation

- Logical operators `&&` and `||` evaluate the *left operand first*.
- If the result is then known, the right operand will *not be* evaluated.

```
x != 0 && z / x > y
⇒ No division by 0
```