Educational Objectives

- You know where you can find a table with all operators in it
- You understand the structure of a floating point number system
- You can compute the binary representation of a floating point number
- You know the most important control flow structures and you can use them in the right situation
- You understand the visibility of variables and you can show the scope of a variable

6. Operatoren

Tabular overview of all relevant operators

Table of Operators

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator</th>
<th>Arity</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object member access</td>
<td>.</td>
<td>2</td>
<td>16</td>
<td>left</td>
</tr>
<tr>
<td>Array access</td>
<td>( )</td>
<td>2</td>
<td>16</td>
<td>left</td>
</tr>
<tr>
<td>Method invocation</td>
<td>( )</td>
<td>2</td>
<td>16</td>
<td>left</td>
</tr>
<tr>
<td>Postfix increment/decrement</td>
<td>++ --</td>
<td>1</td>
<td>15</td>
<td>left</td>
</tr>
<tr>
<td>Prefix increment/decrement</td>
<td>++ --</td>
<td>1</td>
<td>14</td>
<td>right</td>
</tr>
<tr>
<td>Plus, minus, logical not</td>
<td>+ - !</td>
<td>1</td>
<td>14</td>
<td>right</td>
</tr>
<tr>
<td>Type cast</td>
<td>( )</td>
<td>1</td>
<td>13</td>
<td>right</td>
</tr>
<tr>
<td>Object creation</td>
<td>new</td>
<td>1</td>
<td>13</td>
<td>right</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>* / %</td>
<td>2</td>
<td>12</td>
<td>left</td>
</tr>
<tr>
<td>Additive</td>
<td>+ -</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>String concatenation</td>
<td>+</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>2</td>
<td>9</td>
<td>left</td>
</tr>
<tr>
<td>Type comparison</td>
<td>instanceof</td>
<td>2</td>
<td>9</td>
<td>left</td>
</tr>
<tr>
<td>(non-)equality</td>
<td>== !=</td>
<td>2</td>
<td>8</td>
<td>left</td>
</tr>
<tr>
<td>Logical and</td>
<td>&amp;&amp;</td>
<td>2</td>
<td>4</td>
<td>left</td>
</tr>
<tr>
<td>Logical or</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Conditional</td>
<td>? :</td>
<td>3</td>
<td>2</td>
<td>right</td>
</tr>
<tr>
<td>Assignments</td>
<td>* + - = *= /= %=</td>
<td>2</td>
<td>1</td>
<td>right</td>
</tr>
</tbody>
</table>

Table of Operators - Explanations

- The arity shows the number of operands
- A higher precedence means stronger binding
- In case of the same precedence, evaluation order is defined by the associativity
7. Floating Point Numbers

Floating Point Number Systems; IEEE Standard;

We remember from last time

```java
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);
    }
}
```

input 1.1
input 1.0
input 0.1
output 2.2351742E-8

What is going on here?

Why is this happening?

- Not all real numbers can be represented
- Rounding errors can propagate and amplify throughout program execution

We want to understand why this is happening!

Definition: Floating Point Number Systems

A floating point number system describes a sub-set of real numbers by restricting the precision and the value range.
A Floating Point Number System is defined by the four natural numbers:

- $\beta \geq 2$, the Basis,
- $p \geq 1$, the precision (number of places),
- $e_{\text{min}}$, the smallest possible exponent,
- $e_{\text{max}}$, the largest possible exponent.

Notation:

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

Representations of the decimal number 0.1:

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

**Definition:** *Normalized representation*

A representation is normalized iff the exist exactly one digit not equal 0 before the comma.
Normalized Representation

Normalized Number:

\[ \pm d_0 \cdot d_1 \ldots d_{p-1} \times \beta^e, \quad d_0 \neq 0 \]

**Bemerkung 1**
The normalized representation is unique and therefore preferred.

**Remark 2**
The number 0 (and all numbers smaller than \( \beta^{e_{\text{min}}} \)) have no normalized representation (we will deal with this later)!

Set of Normalized Numbers

\[ F^*(\beta, p, e_{\text{min}}, e_{\text{max}}) \]

Binary and Decimal Systems

- Internally the computer computes with \( \beta = 2 \) (binary system)
- Literals and inputs have \( \beta = 10 \) (decimal system)
- Inputs have to be converted!
Conversion Decimal $\rightarrow$ Binary

Angenommen, $0 < x < 2$.

- Hence: $x' = b_{-1} b_{-2} b_{-3} b_{-4} \ldots = 2 \cdot (x - b_0)$
- Step 1 (for $x$): Compute $b_0$:
  \[ b_0 = \begin{cases} 
  1, & \text{if } x \geq 1 \\
  0, & \text{otherwise} 
  \end{cases} \]
- Step 2 (for $x$): Compute $b_{-1}, b_{-2}, \ldots$:
  Go to step 1 (for $x' = 2 \cdot (x - b_0)$)

Binary representation of $1.1$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$b_i$</th>
<th>$x - b_i$</th>
<th>$2(x - b_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>$b_0 = 1$</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>0.2</td>
<td>$b_{-1} = 0$</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>0.4</td>
<td>$b_{-2} = 0$</td>
<td>0.4</td>
<td>0.8</td>
</tr>
<tr>
<td>0.8</td>
<td>$b_{-3} = 0$</td>
<td>0.8</td>
<td>1.6</td>
</tr>
<tr>
<td>1.6</td>
<td>$b_{-4} = 1$</td>
<td>0.6</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2</td>
<td>$b_{-5} = 1$</td>
<td>0.2</td>
<td>0.4</td>
</tr>
</tbody>
</table>

$\Rightarrow 1.00011$, periodic, not finite

Binary Number Representations of $1.1$ and $0.1$

- are not finite, there are errors when converting into a (finite) binary floating point system.
- $1.1f$ and $0.1f$ do not equal $1.1$ and $0.1$, but slightly inaccurate approximation of these numbers.

\[
\begin{align*}
1.1 & = 1.1000000000000000888178\ldots \\
1.1f & = 1.1000000238418\ldots
\end{align*}
\]

Computing with Floating Point Numbers

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

\[
1.111 \cdot 2^{-2} + 1.011 \cdot 2^{-1} = 1.001 \cdot 2^0
\]

1. adjust exponents by denormalizing of one number
2. binary addition of the mantissa
3. renormalize
4. round to $p$ significant places, if necessary
The IEEE Standard 754

Defines floating point number systems and their rounding behavior

- Single precision (float) numbers:
  \[ F^*(2, 24, -126, 127) \text{ plus } 0, \infty, \ldots \]

- Double precision (double) numbers:
  \[ F^*(2, 53, -1022, 1023) \text{ plus } 0, \infty, \ldots \]

- All arithmetic operations round the exact result to the next representable number

32-bit Representation of a Floating Point Number

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>± 2^{126}, \ldots, 2^{127}</td>
<td>1.00000000000000000000000...</td>
</tr>
<tr>
<td>± 0, \infty, \ldots</td>
<td>1.11111111111111111111111...</td>
</tr>
</tbody>
</table>

The IEEE Standard 754

Why

- 1 sign bit
- 23 bit for the mantissa (leading bit is 1 and is not stored)
- 8 bit for the exponent (256 possible values) (254 possible exponents, 2 special values: 0, \infty, \ldots)

⇒ 32 bit overall.

Why

- 1 sign bit
- 52 bit for the mantissa (leading bit is 1 and is not stored)
- 11 bit for the exponent (2046 possible exponents, 2 special values: 0, \infty, \ldots)

⇒ 64 bit overall.
8. Control Structures

Selection Statements, Iteration Statements, Termination, Blocks, Visibility, Local Variables, Switch Statement

Statements

A statement is ...
- comparable with a sentence in natural language
- a complete execution unit
- always finished with a `semicolon`

Example

```java
f = 9f * celsius / 5 + 32;
```

Statement types

Valid statements are:
- Declaration statement
- Assignments
- Increment/decrement expressions
- Method calls
- Object-creation expressions
- Null statement

Examples

```java
float aValue;
aValue = 8933.234;
aValue++;
Out.println(aValue);
new Student();
```

Blocks

A block is ...
- a group of statements
- allowed wherever statements are allowed
- Represented by curly braces

```
{ 
  statement1
  statement2
  ...
}
```

Control Flow

- up to now linear (from top to bottom)
- For interesting programs we need “branches” and “jumps”

Selection Statements

- implement branches
  - if statement
  - if-else statement

if-Statement

If condition is true then statement is executed

```
if ( condition ) 
  statement
```

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

If \( \text{condition} \) is true then \( \text{statement1} \) is executed, otherwise \( \text{statement2} \) is executed.

- \( \text{condition} \): expression of type boolean
- \( \text{statement1} \): body of the if-branch
- \( \text{statement2} \): body of the else-branch

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

**Layout!**

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

**Iteration Statements**

Implement "loops"

- for-statement
- while-statement
- do-statement

**Example: Compute** \( 1 + 2 + \ldots + n \)

```java
// input
Out.print("Compute the sum 1+...+n for n=?");
int n = In.readInt();
// computation of sum_{i=1}^n i
int s = 0;
for (int i = 1; i <= n; ++i){
    s += i;
}
// output
Out.println("1+...+" + n + " = " + s);
```
for-Statement: Syntax

for (init statement condition ; expression)
statement

- **init-statement**: expression statement, declaration statement, null statement
- **condition**: expression of type boolean
- **expression**: any expression
- **statement**: any statement (body of the for-statement)

for-Statement: semantics

for (init statement condition ; expression)
statement

- **init-statement** is executed
- **condition** is evaluated
  - true: Iteration starts
  - statement is executed
  - expression is executed
- false: for-statement is ended.

Example: Harmonic Numbers

- The \( n \)-the harmonic number is
  \[
  H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.
  \]

- This sum can be computed in forward or backward direction, which mathematically is clearly equivalent
Example: Harmonic Numbers

Results:

- Compute $H_n$ for $n = 10000000$
  - Forward sum = 15.4037
  - Backward sum = 16.686

- Compute $H_n$ for $n = 100000000$
  - Forward sum = 15.4037
  - Backward sum = 18.8079

Example: Harmonic Numbers

Observation:
- The forward sum stops growing at some point and is getting “really” wrong.
- The backward sum reasonably approximates $H_n$.

Erklärung:
- For $1 + 1/2 + 1/3 + \cdots$ the late terms are too small to actually contribute
- *Floating Point Rule 2*

Example: Prime Number Test

Def.: A natural number $n \geq 2$ is a prime number, if no $d \in \{2, \ldots, n-1\}$ divides $n$.

A loop that can test this:

```c
int d;
for (d=2; n%d != 0; ++d);
```

Example: Termination

```c
int d;
for (d=2; n%d != 0; ++d);
```

Progress: Initial value $d=2$, then plus 1 in every iteration ($++d$)

Exit: $n%d != 0$ evaluates to $true$ as soon as a divisor is found — at the latest, once $d == n$

Progress guarantees that the exit condition will be reached
Example: Correctness

```java
int d;
for (d=2; n%d != 0; ++d); // for n >= 2
```

Every potential divisor \(2 \leq d \leq n\) will be tested. If the loop terminates with \(d == n\) then and only then is \(n\) prime.

---

Endless Loops

- Endless loops are easy to generate:
  ```java
  for ( ; ; ) ;
  ```
  - Die empty condition is true.
  - Die empty expression has no effect.
  - Die null statement has no effect.
- ... but can in general not be automatically detected.
  ```java
  for ( e; v; e) r;
  ```

---

Halting Problem

Undecidability of the Halting Problem

There is no Java program that can determine for each Java-Program \(P\) and each input \(I\) if the program \(P\) terminates with the input \(I\).

This means that the correctness of programs can in general not be automatically checked.\(^4\)

---

Example: The Collatz-Sequence \((n \in \mathbb{N})\)

- \(n_0 = n\)
- \(n_i = \begin{cases} 
  \frac{n_{i-1}}{2}, & \text{falls } n_{i-1} \text{ gerade} \\
  3n_{i-1} + 1, & \text{falls } n_{i-1} \text{ ungerade}
\end{cases}, \quad i \geq 1.\)

\(n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots \) (Repetition bei 1)

---

\(^4\)Alan Turing, 1936. Theoretical questions of this kind were the main motivation for Alan Turing to construct a computing machine.
The Collatz-Sequence in Java

// Input
Out.println("Compute Collatz sequence, n =? ");
int n = In.readInt();

// Iteration
while (n > 1) { // stop when 1 reached
    if (n % 2 == 0) { // n is even
        n = n / 2;
    } else { // n is odd
        n = 3 * n + 1;
    }
    Out.print(n + " ");
}

Die Collatz-Folge in Java

n = 27:
82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242,
121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233,
700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336,
668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276,
638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429,
7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232,
4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20,
10, 5, 16, 8, 4, 2, 1

The Collatz-Sequence

Does 1 occur for each n?

- It is conjectured, but nobody can prove it!
- If not, then the while-statement for computing the Collatz-sequence can theoretically be an endless loop for some n.

while-statement: why?

- In a for-statement, the expression often provides the progress ("counting loop")

```java
for (int i = 1; i <= n; ++i){
    s += i;
}
```

- If the progress is not as simple, while can be more readable.
while-Statement: Semantics

while ( condition )
statement

- condition is evaluated
  - true: iteration starts
  - false: while-statement ends.

while Statement

while ( condition )
statement

- statement: arbitrary statement, body of the while statement.
- condition: expression of type boolean.

Example: Mini-Calculator

```java
int a; // next input value
int s = 0; // sum of values so far
do {
   Out.print("next number =? ");
a = In.readInt();
s += a;
   Out.println("sum = " + s);
} while (a != 0);
```
do Statement

```java
do
    statement
while ( expression );
```

- `statement`: arbitrary statement, body of the do statement.
- `expression`: expression of type boolean.

do Statement: Semantics

- Iteration starts
  - `statement` is executed.
- `expression` is evaluated
  - `true`: iteration begins
  - `false`: do-statement ends.

Blocks

- Example: body of the main function
  ```java
  public static void main(String[] args) {
      ...
  }
  ```

- Example: loop body
  ```java
  for (int i = 1; i <= n; ++i) {
      s += i;
      Out.println("partial sum is "+ s);
  }
  ```
Visibility

Declaration in a block is not “visible” outside of the block.

```java
public static void main(String[] args) {
    {  
        int i = 2;
    }
    Out.println(i); // Fehler: undeclared Name
}
```

Control Statement defines Block

In this regard, statements behave like blocks.

```java
public static void main(String[] args) {
    {  
        for (int i = 0; i < 10; ++i){
            s += i;
        }
        Out.println(i); // Fehler: undeclared Name
    }
}
```

Scope of a Declaration

Scope: from declaration until end of the part that contains the declaration.

```java
{  
    int i = 2;
}
```

in control statement

```java
for (int i = 0; i < 10; ++i) {s += i; ... }
```

Automatic Memory Lifetime

Local Variables (declaration in block)

- are (re-)created each time their declaration are reached
- memory address is assigned (allocation)
- potential initialization is executed
- are deallocated at the end of their declarative region (memory is released, address becomes invalid)
Local Variables

```java
public static void main(String[] args) {
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        Out.println(++i); // outputs 6, 7, 8, 9, 10
        int k = 2;
        Out.println(--k); // outputs 1, 1, 1, 1, 1
    }
}
```

Local variables (declaration in a block) have automatic lifetime.

Equivalence of Iteration Statements

We have seen:
- while and do can be simulated with for

It even holds:
- The three iteration statements provide the same “expressiveness” (lecture notes)

The “right” Iteration Statement

Goals: readability, conciseness, in particular
- few statements
- few lines of code
- simple control flow
- simple expressions

Often not all goals can be achieved together.

Conclusion

- Selection (conditional branches)
  - if and if-else-statement
- Iteration (conditional jumps)
  - for-statement
  - while-statement
  - do-statement
- Blocks and scope of declarations
Odd Numbers in \{0, \ldots, 100\}

First (correct) attempt:

```java
for (int i = 0; i < 100; ++i) {
    if (i % 2 == 0){
        continue;
    }
    Out.println(i);
}
```

Less statements, less lines:

```java
for (int i = 0; i < 100; ++i) {
    if (i % 2 != 0){
        Out.println(i);
    }
}
```

Odd Numbers in \{0, \ldots, 100\}

Less statements, simpler control flow:

```java
for (int i = 1; i < 100; i += 2) {
    Out.println(i);
}
```

This is the “right” iteration statement!

The switch-Statement

```java
int Note;
...
switch (Note) {
    case 6:
        Out.print("super!");
        break;
    case 5:
        Out.print("gut!");
        break;
    case 4:
        Out.print("ok!");
        break;
    default:
        Out.print("schade.");
}
```

- **Condition**: Expression, convertible to integral type
- **Statement**: arbitrary statement, in which case and default-labels are permitted, break has a special meaning.
Semantics of the switch-statement

switch (condition)
statement

- condition is evaluated.
- If statement contains a case-label with (constant) value of condition, then jump there.
- Otherwise jump to the default-label, if available. If not, jump over statement.
- The break statement ends the switch-statement.

Definition: Control Flow

Order of the (repeated) execution of statements

- generally from top to bottom...
- ...except in selection and iteration statements

Kontrollfluss switch in general

If break is missing, continue with the next case.

7: Keine Note!
6: bestanden!
5: bestanden!
4: bestanden!
3: oops!
2: ooops!
1: oooops!
0: Keine Note!

Control Flow

switch (Note) {
  case 6:
  case 5:
  case 4:
    Out.print("bestanden!");
    break;
  case 1:
    Out.print("o");
  case 2:
    Out.print("o");
  case 3:
    Out.print("oops!");
    break;
  default:
    Out.print("Keine Note!");
}
Control Flow if else

if ( condition )
  statement1
else
  statement2

Control Flow for

for( init statement  condition ; expression )
  statement

Control Flow while

while ( condition )
  statement

Control Flow do while

do
  statement
while ( condition )
Control Flow switch

- switch
- break
- case
- case
- default
- break