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 imporant control flow stuctures 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

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

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

```
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.

Floating Point Number Systems

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),
- $e_{\min}$, the smallest possible exponent,
- $e_{\max}$, the largest possible exponent.

Notation:

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

Floating Point Number Systems

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

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

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

represented with Basis β:

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

Floating Point Number Systems

Example

- $\beta = 10$

Representations of the decimal number 0.1

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

Definition: *Normalized representation*

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

Normalized Representation

Normalized Number:

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

Bemerkung 1

The normalized representation is unique and therefore prefered.

Remark 2

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

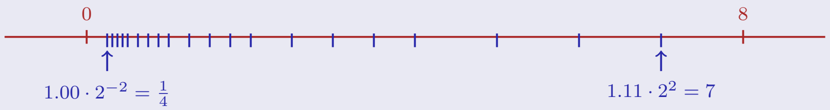
Set of Normalized Numbers

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

Normalized Representation

Example $F^*(2, 3, -2, 2)$ (only positive numbers)

$d_0{\bullet}d_1d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
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



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

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

$\Rightarrow 1.0\overline{0011}$, 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.

$$\texttt{1.1} = \underline{1.1000000000000000}888178\ldots$$
$$\texttt{1.1f} = \underline{1.1000000}238418\ldots$$

Computing with Floating Point Numbers

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

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

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:

  $F^*(2, 24, -126, 127)$ plus $0, \infty, \ldots$

- Double precision (`double`) numbers:

  $F^*(2, 53, -1022, 1023)$ plus $0, \infty, \ldots$

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

32-bit Representation of a Floating Point Number

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$\pm$ Exponent Mantisse

$\pm$ $2^{-126}, \ldots, 2^{127}$ / $0, \infty, \ldots$

$1.00000000000000000000000$ … $1.11111111111111111111111$

The IEEE Standard 754

Why

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

- 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, ∞,…)

$\Rightarrow$ 32 bit overal.

The IEEE Standard 754

Why

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

- 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, ∞,…)

$\Rightarrow$ 64 bit overal.

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

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


Statement types

Valid statements are:

- Declaration statement
- Assignments
- Increment/decrement expressions
- Method calls
- Object-creation expressions
- Null statement

Statement types

Examples

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

Blocks

A block is . . .

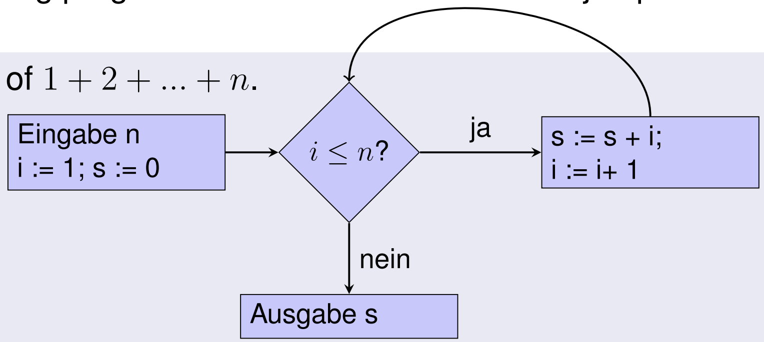
- a group of statements
- allowed wherever statements are allowed
- Represented by curly braces

```
{
   statement1
   statement2
   ⋮
}
```


Control Flow

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

Computation of $1 + 2 + ... + n$.



Selection Statements

implement branches

- `if` statement
- `if-else` statement

`if`-Statement

```
if ( condition )
    statement
```

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

If *condition* is true then *statement* is executed

- *statement*: arbitrary statement (*body* of the `if`-Statement)
- *condition*: expression of type `boolean`

`if-else`-statement

if (*condition*)
    statement1
else
    statement2

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

If *condition* is true then *statement1* is executed, otherwise *statement2* is executed.

- *condition*: expression of type `boolean`
- *statement1*: *body* of the `if`-branch
- *statement2*: *body* of the `else`-branch

Layout!

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


Iteration Statements

implement “loops”

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

Example: Compute $1 + 2 + ... + n$

```
// 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

```
Out.print("Compute H_n for n =? ");
int n = In.readInt();

float fs = 0;
for (int i = 1; i <= n; ++i){
    fs += 1.0f / i;
}
Out.println("Forward sum = " + fs);

float bs = 0;
for (int i = n; i >= 1; --i){
    bs += 1.0f / i;
}
Out.println("Backward sum = " + bs);
```

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:

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


Example: Termination

```
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

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

Every potential divisor `2 <= d <= 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:

```
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.

```
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]

[4] Alan Turing, 1936. Theoretical quesitons of this kind were the main motivation for Alan Turing to construct a computing machine.

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

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

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

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”)

```
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
    statement is executed
  - `false`: `while`-statement ends.

`while` Statement

```
while ( condition )
  statement
```

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

`while` Statement

```
while ( condition )
  statement
```

is equivalent to

```
for ( ; condition ; )
  statement
```


Example: Mini-Calculator

```
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

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

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

`do` Statement

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

is equivalent to

```
statement
while ( expression )
  statement
```


`do`-Statement: Semantics

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

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

Blocks

- Example: body of the main function

```
public static void main(String[] args) {
     ...
}
```

- Example: loop body

```
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.

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

main block

block

„Blickrichtung“

Control Statement defines Block

In this regard, statements behave like blocks.

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

block

Scope of a Declaration

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

in the block

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

scope

in function body

```
void main(String[] args) {
    int i = 2;
    ...
}
```

scope

in control statement

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

scope

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

```
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*.

Conclusion

- Selection (conditional *branches*)
  - `if` and `if-else`-statement
- Iteration (conditional *jumps*)
  - `for`-statement
  - `while`-statement
  - `do`-statement
- Blocks and scope of declarations

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.

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

First (correct) attempt:

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


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

Less statements, *less* lines:

```
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:

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

This is the "right" iteration statement!

The `switch`-Statement

`switch` (*condition*)
statement

- *condition*: Expression, convertible to integral type
- *statement* : arbitrary statemet, in which `case` and `default`-lables are permitted, `break` has a special meaning.

```
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.");
}
```

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`-lable, if available. If not, jump over `statement`.
- The `break` statement ends the `switch`-statement.

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!

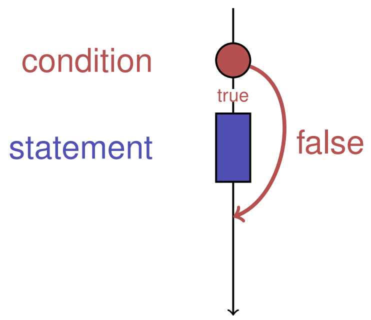
```
switch (Note) {
    case 6:
    case 5:
    case 4:
        Out.print("bestanden!");
        break;
    case 1:
        Out.print("o");
    case 2:
        Out.print("o");
    case 3:
        Out.print("oops!");
        break;
    default:
        Out.print("Keine Note!");
}
```


Definition: *Control Flow*

Order of the (repeated) execution of statements

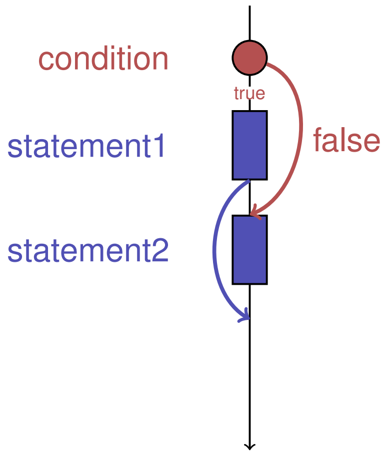
Control Flow

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



`if` (*condition*)
statement

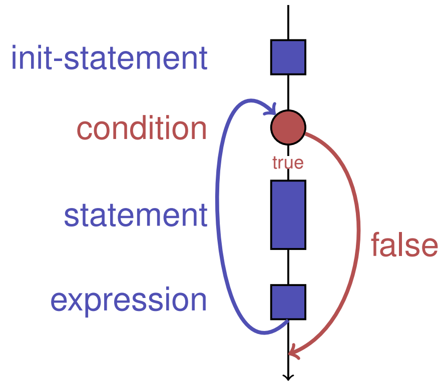
Control Flow `if else`



if (*condition*)
    statement1
else
    statement2

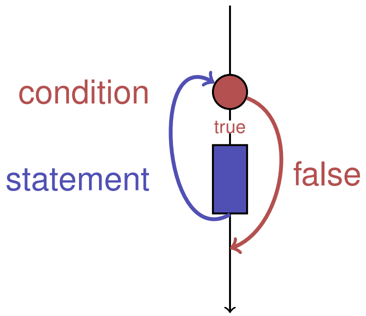
Control Flow `for`

for (*init statement* *condition* ; *expression*)
    statement



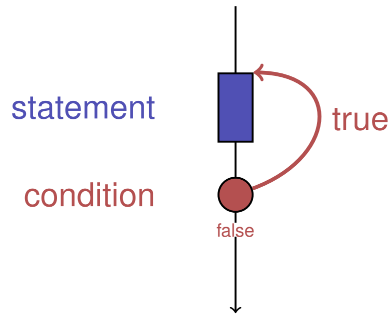
Control Flow `while`

while (condition)
    statement



Control Flow `do while`

do
    statement
while (condition)



Control Flow `switch`