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 concatenation	`+`	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. Safe Programming: Assertions

Assertions

Sources of Errors

- Errors that the compiler can find: syntactical and some semantical errors
- Errors that the compiler cannot find: runtime errors (always semantical)

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior

   ≫ It's not a bug, it's a feature !!≪

2. Check at many places in the code if the program is still on track!
3. Question the (seemingly) obvious, there could be a typo in the code.

Against Runtime Errors: *Assertions*

```
assert expr;
```

- throws an error and halts the program if the boolean expression `expr` is false
- can be switched on or off when starting the Java-VM

Alternativ: `assert expr : expr2;`

- If the assertion doesn't hold, the value of `expr2` is shown in the console

Div-Mod Identity `a/b * b + a%b == a`

Check if the program is on track. . .

```
Out.println("Dividend a =? ");
int a = In.readInt();

Out.println("Divisor b =? ");
int b = In.readInt();

// check input
assert b != 0 : "User error: b must not be zero";
```

Input arguments for calculation

Precondition for the ongoing computation

Div-Mod identity $a/b * b + a\%b == a$

. . . and question the obvious!

```
// check input
assert b != 0 : "User error: b must not be zero";

// compute result
int div = a / b;
int mod = a % b;

// check result
assert div * b + mod == a;
...
```

← Div-Mod identity

8. Control Structures

Selection Statements, Iteration Statements, Termination, Blocks, Visibility, Local Variables, While Statement, Do Statement, Jump Statements

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 . . .

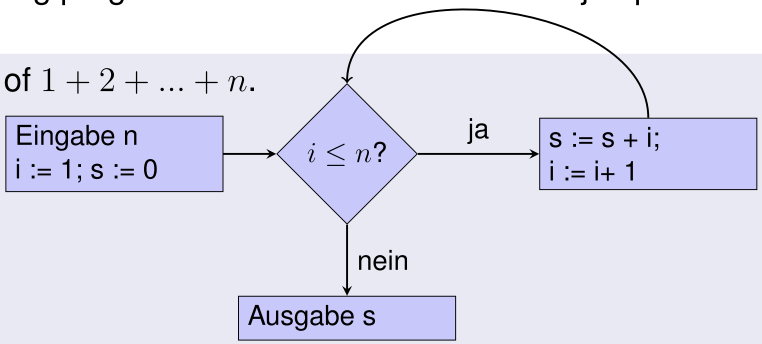
- a group of statements
- allowed wherever statements are allowed
- Represented by curly braces

{
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

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 Example

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

Assumptions: `n == 2`, `s == 0`

i		s
i==1	wahr	s == 1
i==2	wahr	s == 3
i==3	falsch	

s == 3

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.

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

Harmonic Numbers (Floating Point Rule 2)

```
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);
```


Harmonic Numbers (Floating Point Rule 2)

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
  ```


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

`for`-Statement: Termination

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

Hier und meistens:

- *expression* changes its value that appears in *condition* .
- After a finite number of iterations *condition* becomes false: *Termination*

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.[5]

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

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);
```

- Observation 1: After the `for`-statement it holds that $d \leq n$.
- Observation 2: n is a prime number if and only if finally $d = n$.

Revisit: 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*.

`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

`while`-Statement: Semantics

`while (` *condition* `)`
  statement

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

`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.

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

```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.

`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.

do-Statement: Example Calculator

Sum up integers (when 0 then stop):

```
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);
```


Conclusion

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

Jump Statements

- `break`
- `continue`

break-Statement

```
break;
```

- Immediately leave the enclosing iteration statement.
- useful in order to be able to break a loop "in the middle" [6]

[6] and indispensible for switch-statements.

Calculator with break

Sum up integers (stop when 0 occurs)

```
int a;
int s = 0;
do {
    Out.print("next number =? ");
    a = In.readInt();
    // irrelevant in letzter Iteration:
    s += a;
    Out.println("sum = " + s);
} while (a != 0);
```


Calculator with break

Suppress irrelevant addition of 0:

```
int a;
int s = 0;
do {
    Out.print("next number =? ");
    a = In.readInt();
    if (a == 0) break; // Abbruch in der Mitte
    s += a;
    Out.println("sum = " + s);
} while (a != 0)
```


Calculator with break

Equivalent and yet more simple:

```
int a;
int s = 0;
for (;;) {
    Out.print("next number =? ");
    a = In.readInt();
    if (a == 0) break; // Abbruch in der Mitte
    s += a;
    Out.println("sum = " + s);
}
```

Calculator with `break`

Version without break evaluates a twice and requires an additional block.

```
int a = 1;
int s = 0;
for (;a != 0;) {
    Out.print("next number =? ");
    a = In.readInt();
    if (a != 0) {
        s += a;
        Out.println("sum = " + s);
    }
}
```


`continue`-Statement

```
continue;
```

- Jump over the rest of the body of the enclosing iteration statement
- Iteration statement is *not* left.

Calculator with `continue`

Ignore negative input:

```
for (;;)
{
    Out.print("next number =? ");
    a = In.readInt();
    if (a < 0) continue; // springe zu }
    if (a == 0) break;
    s += a;
    Out.println("sum = " + s);
}
```


Equivalence of Iteration Statements

We have seen:

- `while` and `do` can be simulated with `for`

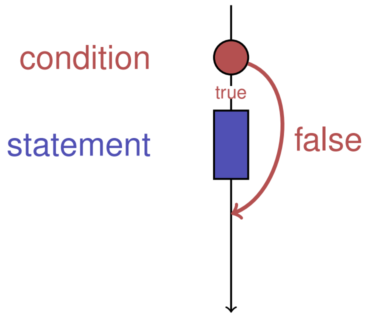
It even holds: Not so simple if a continue is used!

- The three iteration statements provide the same “expressiveness” (lecture notes)

Control Flow

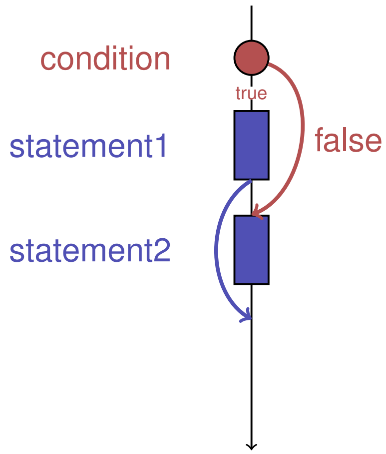
Order of the (repeated) execution of statements

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



`if` (*condition*)
    statement

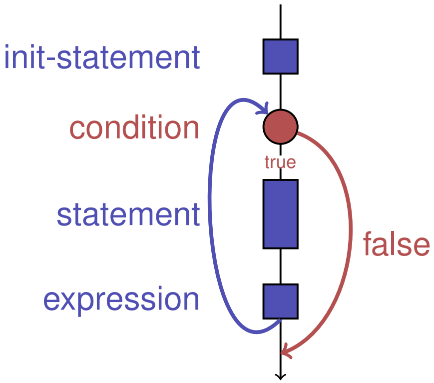
Control Flow `if else`



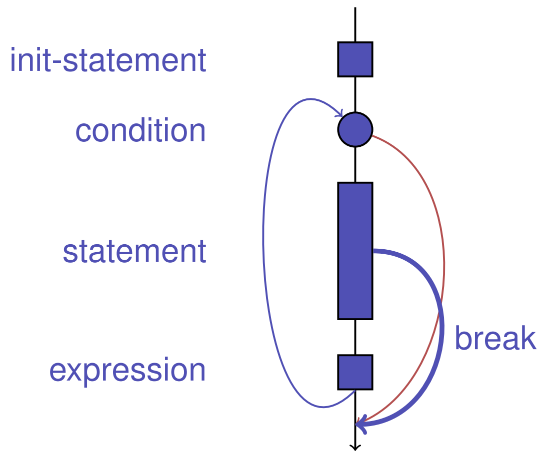
`if` (*condition*)
    statement1
else
    statement2

Control Flow `for`

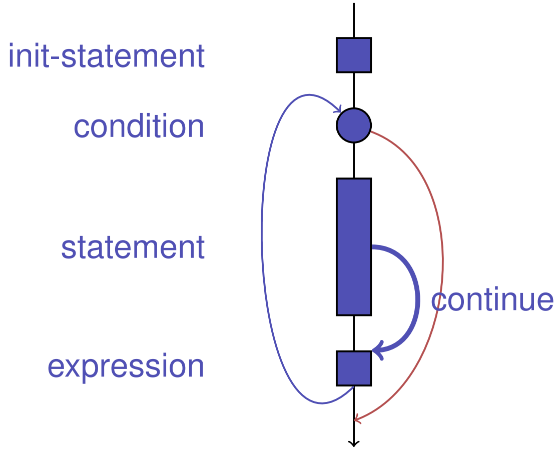
`for` (*init statement* *condition* ; *expression*)
    statement



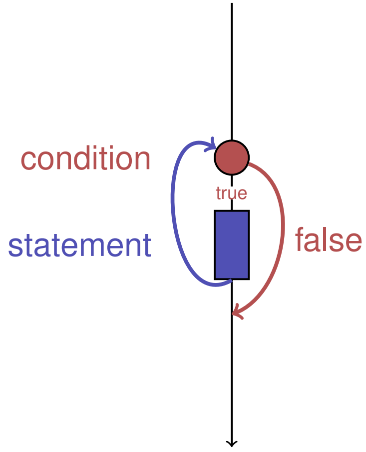
Kontrollfluss `break` in for



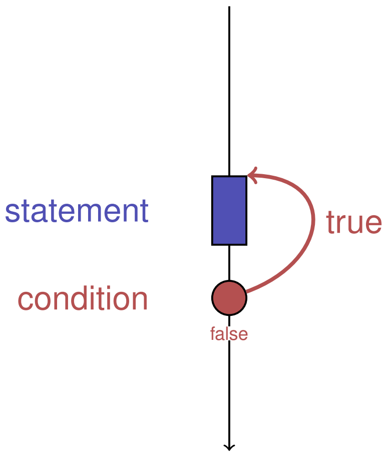
Control Flow `continue` in for



Control Flow `while`



Control Flow `do while`



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 $\{0, \dots, 100\}$

First (correct) attempt:

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


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

Less statements, *less* lines:

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


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

Less statements, *simpler* control flow:

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

This is the “right” iteration statement!

Jump Statements

- implement unconditional jumps.
- are useful, such as `while` and `do` but not indispensible
- should be used with care: only where the control flow is *simplified* instead of making it *more complicated*

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.

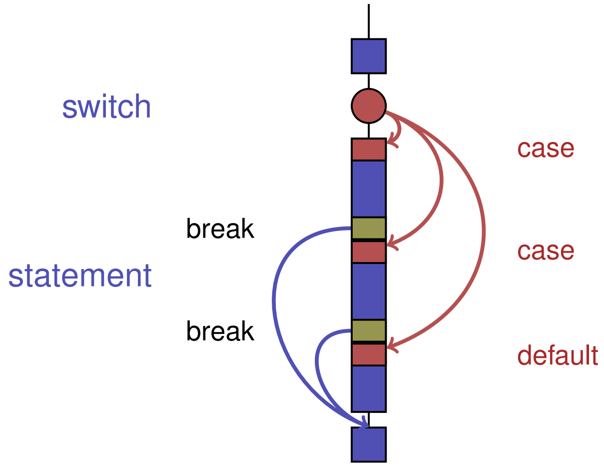
```
int Note;
...
switch (Note) {
    case 6:
        Out.print("super!");
        break;
    case 5:
        Out.print("gut!");
        break;
    case 4:
        Out.print("ok!");
        break;
    default:
        Out.print("schade.");
}
```


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.

Control Flow `switch`



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!");
}
```