

Programmieren II: Java

Grundlagen

Prof. Dr. Christopher Auer

Sommersemester 2024



Syntaktische Elemente von Java

Anweisungen, Ausdrücke und Blöcke

Datentypen

Lokale Variablen

Einschub: Ein- und Ausgabe auf der Konsole

Operatoren

if-then-else, switch-case: Bedingte Ausführung

while- und for-Schleifen und Schleifen-Kontrollfluss

Methoden, Signaturen, Rekursion

Inhalt

Syntaktische Elemente von Java

Token, Schlüsselwörter, Bezeichner und Co.

Bezeichner

Kommentare und JavaDoc

Inhalt

Syntaktische Elemente von Java

Token, Schlüsselwörter, Bezeichner und Co.

Token

- ▶ Java-Compiler zerlegt Quellcode in **Token**
- ▶ Arten von Token
 - ▶ **Whitespaces**: Leerzeichen, Tabulator, Vorschub
 - ▶ **Separatoren**: () { } [] ; , @ ::
 - ▶ **Bezeichner**: Methoden-, Klassen-, Variablennamen HelloWorld _A_VARIABLE ABC012
 - ▶ **Literale**: Integer, Gleitkommazahlen, Strings, Zeichen 3.1415f "Hello!" 42 **null** 'A'
 - ▶ **Schlüsselwörter**: von Java reservierte Worte **class public for while static**
 - ▶ **Operatoren**: Operatoren für Zuweisung und Berechnungen = == / * ^ && |

Beispiel

```
public class ExampleClass{  
    private static int square(int i){  
        return i * i;  
    }  
    public static void main(String[] args){  
        int zahl = 4;  
        System.out.println("Die Zahl "+zahl+" quadriert ist "+  
            square(zahl));  
    }  
}
```

Bezeichner, Operatoren, Literale, Schlüsselworte, Separatoren

Java-Schlüsselwörter

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Inhalt

Syntaktische Elemente von Java

Bezeichner

Bezeichner

Spezifikation:

- ▶ Identifier: IdentifierChars aber kein Schlüsselwort, **null**, **true** oder **false**
- ▶ IdentifierChars: `JavaLetter(JavaLetterOrDigit)*`, d.h. zuerst ein `JavaLetter` und dann beliebig viele `JavaLetterOrDigits`
- ▶ `JavaLetter`: A–Z, a–z, \$, _ und Unicode-Characters
- ▶ `JavaLetterOrDigit`: `JavaLetter` oder eine der Ziffern 0–9

Gültig

- ✓ Zahl
- ✓ `area51`
- ✓ `_u_n_d_e_r_s_c_o_r_e`
- ✓ `many$`
- ✓ `café`
- ✓ Käsesoßenrührlöffel

Ungültig

- ✓ große Zahl
- ✓ `class`
- ✓ `_`
- ✓ `4ever`
- ✓ `true`

Namenskonventionen

Java kommt mit Namenskonventionen

- ▶ **Variablen:** „lowerCamelCase“
counter, addressBook, sortedCalenderEntries
- ▶ **Klassen/Enums/Interfaces:** Nomen, „UpperCamelCase“
ArrayList, JFrame, SQLQuery
- ▶ **Methoden:** Verben, „lowerCamelCase“ getSize, connect, removeEntry
- ▶ **Konstanten/Enum-Werte:** „SCREAMING_SNAKE_CASE“
RED, ACTIVE_STATE, GRAVITATIONAL_CONSTANT
- ▶ **Packages:** „all lowercase‘ java.lang, de.hawlandshut.java1.basics



Beispiel einer Klasse I

```
package de.hawlandshut.java1.basics;

public class CelestialBody
{
    public static final double GRAVITATIONAL_CONSTANT = 6.67430e-11;
    private final double mass;
    private final String name;

    // Konstruktor
    public CelestialBody(String name, double mass)
    {
        this.mass = mass;
        this.name = name;
    }

    // Getter-Methode
    public String getName() {
        return name;
    }

    // Getter-Methode
    public double getMass() {
```

Beispiel einer Klasse II

```
    return mass;
}

// Methode
public double computeForce(CelestialBody otherBody, double distance){
    return GRAVITATIONAL_CONSTANT
        * mass * otherBody.getMass() / (distance*distance);
}
}
```

 CelestialBody.java

Inhalt

Syntaktische Elemente von Java

Kommentare und JavaDoc

Kommentare

- ▶ Wie in C/C++

- ▶ Einzeilig

```
// here be dragons
```

- ▶ Mehrzeilig

```
/*  
 * Die Sterne links sind optional, machen den  
 * Kommentar aber übersichtlicher.  
 */
```

- ▶ Kommentare werden als ein Token gelesen

```
1/*2*/3
```

Token: **int**-Literal, Kommentar, **int**-Literal

- ▶ Kommentare innerhalb von Zeichenketten werden ignoriert

```
"Das ist kein /* Kommentar */ sondern ein String!"
```

- ▶ Wird mit `/**` eingeleitet
- ▶ Inline-Dokumentation

```
/**
 * Computes the gravitional force between this
 * and the other body.
 * @param otherBody other body on which the force acts.
 * @param distance distance (>0) between the two bodies.
 * @return force between the bodies in Newton.
 */
public double computeForce(CelestialBody otherBody,
double distance)
{
    return GRAVITATIONAL_CONSTANT
        * mass * otherBody.getMass() / (distance*distance);
}
```

- ▶ Dokumentation (z.B. HTML) wird über javadoc-Tool erstellt

Inhalt

Anweisungen, Ausdrücke und Blöcke

Anweisungen

Methodenaufrufe

Ausdrücke

Blöcke

Inhalt

Anweisungen, Ausdrücke und Blöcke

Anweisungen

Anweisungen

- ▶ Java ist **imperativ**
 - ▶ in Methoden-Implementierungen
 - ▶ Anweisungen definieren die Abarbeitungsschritte
- ▶ Beispiel für Anweisungen
 - ▶ Methodenaufruf

```
System.out.println("Hello World!");
```

- ▶ Zuweisungen

```
answer = 42;
```

- ▶ Kontrollstrukturen

```
if (answer != 42)  
    System.out.println("Not the right answer!");  
while (answer != 42)  
    answer = findAnswer();
```

- ▶ ...

```
package de.hawlandshut.java1.basics;
🐘 runHelloWorldAdvanced --args="Name"
public class HelloWorldAdvanced{
    private static final String DEFAULT_NAME = "World";

    public static void main(String[] args){
        // snippet: statements
        String name = DEFAULT_NAME;
        if (args.length > 0)
            name = args[0];
        System.out.println("Hello " + name + "!");
        // snippet: /statements
    }
}
```

📄 HelloWorldAdvanced.java

Klassendeklaration

```
public class HelloWorldAdvanced{  
    /* ... */  
}
```

- ▶ **public**: Sichtbarkeit
- ▶ **class**: Schlüsselwort
- ▶ HelloWorldAdvanced: Bezeichner, **muss** so heißen wie Datei
- ▶ Block { } mit Klassendefinition
 - ▶ Definition der Konstanten DEFAULT_NAME

```
private static final String DEFAULT_NAME = "World";
```

- ▶ Definition der Methode main

```
public static void main(String[] args){  
    /* Anweisungen */  
}
```

Klassenvariable

```
private static final String DEFAULT_NAME = "World";
```

- ▶ **private**: Sichtbarkeit nur für die Klasse
- ▶ **static**: Klassenvariable
- ▶ **final**: Konstante
- ▶ **String**: Typ Zeichenkette
- ▶ **DEFAULT_NAME**: Bezeichner
- ▶ **"World"**: Wert

Methodensignatur

```
public static void main(String[] args){  
    /* Anweisungen */  
}
```

- ▶ **public**: für jeden sichtbar
- ▶ **static**: Klassenmethode
- ▶ **void**: Rückgabewert (hier kein Rückgabewert)
- ▶ **main**: Bezeichner
- ▶ **(String[] args)**: Parameterliste
- ▶ **{ }**: Anweisungen in Block
- ▶ **main** ist eine **besondere Methode**
 - ▶ Einstiegspunkt für Hauptprogramm
 - ▶ **public** und **static**
 - ▶ Signatur: **(String[] args)** (Kommandozeilenparameter)

```
10 String name = DEFAULT_NAME;  
11 if (args.length > 0)  
12     name = args[0];  
13 System.out.println("Hello " + name + "!");
```

📄 HelloWorldAdvanced.java

Lokale Variablendeklaration

```
String name = DEFAULT_NAME;
```

- ▶ String: Typ
- ▶ name: Bezeichner
- ▶ =: Zuweisungsoperator
- ▶ DEFAULT_NAME: Initialwert (aus Konstante)

if-Anweisung

```
if (args.length > 0)
    name = args[0];
```

- ▶ **if-Anweisung**
 - ▶ **if**: Schlüsselwort bedingte Ausführung
 - ▶ `(args.length > 0)`: Boolescher Ausdruck, wird zu **true** oder **false** ausgewertet
- ▶ **Zuweisung**
 - ▶ `name`: Bezeichner der Zielvariable (auch „L-value“ genannt)
 - ▶ `=`: Zuweisungsoperator
 - ▶ `args[0]`: Wert der zugewiesen werden soll
 - ▶ `;`: Anweisungsende

Inhalt

Anweisungen, Ausdrücke und Blöcke

Methodenaufrufe

Methodenaufruf


```
System.out.println("Hello " + name + "!");
```

- ▶ System: Klasse mit zahlreichen Hilfsmethoden
- ▶ out: Klassenvariable von System mit der Referenz zum Standard-Ausgabestrom vom Typ `PrintStream`
- ▶ `println`: Methode der Klasse `PrintStream` zur Ausgabe von Strings
- ▶ `("Hello " + name + "!")`: String, aus drei Teilen konkateniert (über `+`-Operator)
- ▶ `;`: Anweisungsende

Ergänzung zu Methodenaufruf — Überladene Methoden


```
PrintStream.println(boolean x);  
PrintStream.println(char x);  
PrintStream.println(String x);  
PrintStream.println(float x);  
/* ... */
```

- ▶ **derselbe** Methodenname, **unterschiedliche** Signaturen
- ▶ Compiler entscheidet welche Implementierung verwendet wird

```
7  runOverloadingExample  
8 System.out.println(); // Leerzeile  
9 System.out.println(true); // boolean  
10 System.out.println("Hello!"); // String  
11 System.out.println(42); // int  
12 System.out.println(3.14159265359); // double  
13 System.out.println(3.14159265359f); // float
```

 PrintDemos.java

Ergänzung zu Methodenaufruf — Variable Argumente (Varargs)

```
19  runVarargsExample  
20 System.out.printf("PI ist ungefaehr %f%n", 3.14159265359f);  
21 System.out.printf("Ein boolean kann %b oder %b sein%n",  
22     true, false);  
23 System.out.printf("%d ist nur die halbe Wahrheit", 42/2);
```

 PrintDemos.java


- ▶ `PrintStream.printf` akzeptiert **variable Anzahl** Parameter:
 - ▶ string mit Formatanweisungen (ähnlich `printf` in C)
 - ▶ Parameter mit denen Formatanweisungen ersetzt werden
 - ▶ Beispiele
 - ▶ `%d` ganze Zahl
 - ▶ `%f` Dezimalzahl
 - ▶ `%b` Bool'scher Wert **true** oder **false**
 - ▶ `%n` neue Zeile (kein Parameter notwendig)

Inhalt

Anweisungen, Ausdrücke und Blöcke

Ausdrücke

Ausdrücke


```
7  runExpressionExample
8 System.out.println((3908-1234) / 2);
9 System.out.println(Math.PI * Math.PI);
10 System.out.println("The cake is a lie!");
11 System.out.println(true == false);
```

 Expressions.java

- ▶ Ein Ausdruck (engl. „expression“)...
 - ▶ wird **ausgewertet**
 - ▶ und ergibt ein **Resultat**
 - ▶ von einem gewissen **Typ** (z.B. **int**, **boolean** oder eine Referenz)

Auswertung von Ausdrücken

Compiler wertet möglichst viel zur Übersetzungszeit aus


```
17  runExpressionEvaluationExample  
18 double x = Math.random();  
19 double c = (3908-1234)/2; // wird von Compiler berechnet  
20 System.out.println(x + c);
```

 Expressions.java

Bytecode (gekürzt):

```
invokestatic java/lang/Math.random  
dstore_0      // Zufallszahl in 0 speichern  
ldc2_w        double 1337.0d // von Compiler ausgewertet  
dstore_1      // 1337.0 in 1 speichern  
dload_0       // Zufallszahl laden  
dload_1       // 1337.0 laden  
dadd          // die beiden addieren
```


Boolesche Ausdrücke

```
27  runBooleanExpressionExample  
28 System.out.println(true != false);  
29 System.out.println(42 == 21*2);  
30 if (Math.PI == Math.E)  
31     System.out.println("Die Mathematik bricht zusammen.");
```

 Expressions.java

- ▶ **Boolesche Ausdrücke**: Ausdrücke vom Typ **boolean**
- ▶ Verwendung unter anderem bei...
 - ▶ **if**-Anweisung

```
if ( /* boolescher Ausdruck */ ) { /* ... */ }
```

- ▶ Abbruchbedingungen für Schleifen

```
while ( /* boolescher Ausdruck */ ) { /* ... */ }
```

Ausdrucksanweisungen

- **Ausdrucksanweisungen:** Ausdrücke, die auch als Anweisungen funktionieren

```
Math.random(); // Methodenaufruf, liefert Rückgabewert
i++;           // liefert Wert von i, inkrementiert dann
--i;           // dekrementiert i, liefert Wert von i
i = 5;         // weist i Wert zu, liefert Wert von i
new CelestialBody("ISS", 419_700.0);
                // erstellt Instanz, liefert Referenz
```

- Ergebnis wird **verworfen**
- **Nicht jeder Ausdruck** ist eine Ausdrucksanweisung:

```
1 + 2;
a || b && !c;
Math.PI;
```

Liefern jeweils **Compiler-Fehler**


Inhalt

Anweisungen, Ausdrücke und Blöcke Blöcke

```
if ( /* Boolescher Ausdruck */ )  
    Anweisung; // Einzahl!
```

- ▶ `if` akzeptiert nur **eine** Anweisung
- ▶ Was tun bei **mehreren Anweisungen**?
- ▶ Blöcke...
 - ▶ werden von { } umschlossen
 - ▶ fassen mehrere Anweisungen zu **einer Anweisung** zusammen
 - ▶ können **geschachtelt** werden
 - ▶ Variablen in Blöcken sind **lokal**

Blöcke: Beispiel

```
7   runBlocksExample
8  double x = Math.random();
9  if (x > 0) { // if-Block
10     String ausgabe = "Die Zufallszahl ist: %f%n";
11     // Bloecke kann auch man zur Strukturierung verwenden
12     {
13         String s = "Dieser String ist nur hier sichtbar";
14         // sichtbar: x, ausgabe, s
15         System.out.println(s);
16         System.out.printf(ausgabe, x);
17     }
18     // sichtbar: x, ausgabe
19     // System.out.println(s); // FEHLER "unknown symbol s"
20     System.out.printf(ausgabe, x);
21 }
22
23
24 }
```

 Blocks.java

Inhalt

Datentypen

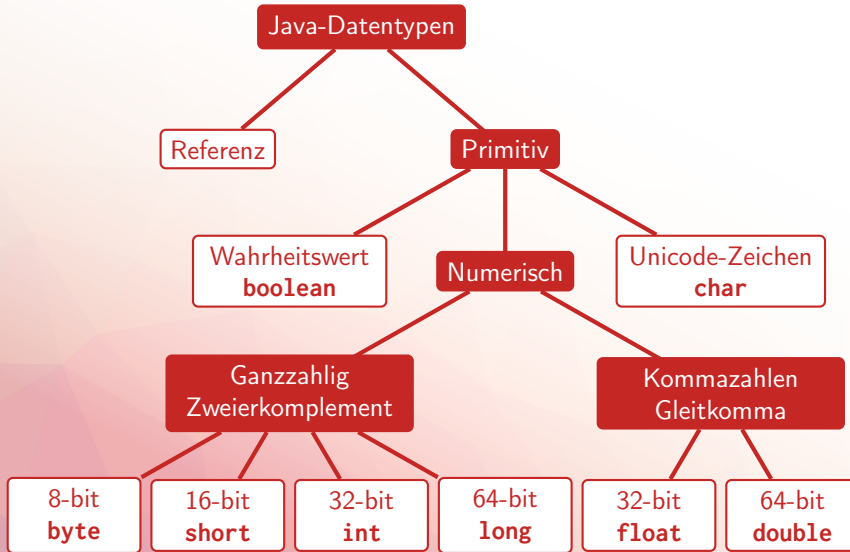
Wertebereiche

Literale

Konvertierung

Überlauf

Java-Datentypen: Übersicht




Inhalt

Datentypen

Wertebereiche

Primitive Typen: Wertebereiche

```
40  runPrintTypeRanges
41 println("boolean: "+Boolean.FALSE+" - "+Boolean.TRUE);
42 println("char: "+Character.MIN_VALUE+" - "+Character.MAX_VALUE);
43 println("byte: "+Byte.MIN_VALUE+" - "+Byte.MAX_VALUE);
44 println("short: "+Short.MIN_VALUE+" - "+Short.MAX_VALUE);
45 println("int: "+Integer.MIN_VALUE+" - "+Integer.MAX_VALUE);
46 println("long: "+Long.MIN_VALUE+" - "+Long.MAX_VALUE);
47 println("float: "+Float.MIN_VALUE+" - "+Float.MAX_VALUE);
48 println("double: "+Double.MIN_VALUE+" - "+Double.MAX_VALUE);
```

 PrimitiveTypes.java

Primitive Typen: Wertebereiche

```
boolean: false, true  
char: - ?  
byte: -128 - 127  
short: -32768 - 32767  
int: -2147483648 - 2147483647  
long: -9223372036854775808 - 9223372036854775807  
float: 1.4E-45 - 3.4028235E38  
double: 4.9E-324 - 1.7976931348623157E308
```

Primitive Typen: Wertebereiche

Datentyp	Wertebereich
boolean	true und false
char	alle Unicode-Zeichen (2 Byte): 0x0000 bis 0xFFFF
byte	−128 bis 127
short	−32.768 bis 32.767
int	−2.147.483.648 bis 2.147.483.647
long	−9.223.372.036.854.775.808 bis −9.223.372.036.854.775.807
float	$\approx 1.401298464324817 \cdot 10^{-45}$ bis $\approx 3.4028235 \cdot 10^{38}$
double	$\approx 4.9406564584124654 \cdot 10^{-324}$ bis $\approx 1.7976931348623158 \cdot 10^{308}$

Inhalt

Datentypen

Literale

Primitive Typen: Literale

- ▶ **boolean:** `true` und `false`
- ▶ **char** (Unicode Characters)
 - ▶ Als Zeichen in einfachen Hochkommas
 - ▶ Zeichen: `'a'`, `'b'`, `'c'`, `'A'`, `'B'`, `'C'`, `'%'`
 - ▶ Spezielle Zeichen (Escape-Sequenzen)

Escape-Sequenz	Bedeutung	Auswirkung
<code>'\b'</code>	Backspace	Cursor springt ein Zeichen nach links
<code>'\t'</code>	Tabulator	Cursor springt um Tabulator nach rechts
<code>'\f'</code>	Form Feed	löscht den Bildschirm
<code>'\r'</code>	Carriage Return	Bewegt Cursor an Zeilenanfang
<code>'\"'</code>	doppeltes Hochkomma	
<code>'\''</code>	einfaches Hochkomma	
<code>'\\'</code>	Backslash	


Primitive Typen: Literale (Fortsetzung char)

- ▶ **Oktal-Darstellung** für ASCII-Code-Zeichen: \YYY
 - ▶ YYY ist eine Oktalzahl von 000_8 bis 377_8 (255_{10})
 - ▶ Beispiele:

```
char A = '\101'; // 'A';  
char a = '\141'; // 'a';  
char qmark = '\077'; // '?';
```

Primitive Typen: Literale (Fortsetzung char)

- ▶ **Unicode-Darstellung:** \uXXYY
 - ▶ XX und YY sind Bytes in Hexadezimaldarstellung
 - ▶ Niedrigster Wert \u0000
 - ▶ Höchster Wert \uFFFF

```
11  runUnicodeExample  
12 char j = '\u0399'; // Greek capital Iota  
13 char a = '\u03AC'; // Greek small Alpha with Tonos  
14 char v = '\u03B2'; // Greek small Beta  
15 char a2 = '\u03B1'; // Greek small Alpha  
17 System.out.printf("%c%c%c%c\n", j, a, v, a2);
```

 PrimitiveTypes.java

Primitive Typen: Literale (ganze Zahlen)

- ▶ Literale für `byte`, `short`, `int`, `long`
 - ▶ **Präfix** definiert Basis des Zahlensystems:

Präfix	Basis	Beispiel
	10 (Dezimal)	42
0b, 0B	2 (Binär)	0b101010
0	8 (Oktal)	052
0x, 0X	16 (Hexadezimal)	0x2A

- ▶ Negatives Vorzeichen durch vorangestelltes „-“


-42, -0b101010, -052, -0x2A

- ▶ Numerische Literale werden als **int** interpretiert
- ▶ **Explizite Definition von long** mit Suffix l (kleines „L“) oder L:

42L, 0b101010L, 052L, 0x2AL

Primitive Typen: Literale (ganze Zahlen)

```

23  runIntegerNumberLiteralsExample
24 byte b = 0b11111111;
25 short s = -077;
26 int i = 0x03AC;
27 // ohne den Suffix L ist der int-Wert "out of range"
28 long l = 0xFFFFFFFFFFL;
30 System.out.printf("b = 0x%x%n", b);
31 System.out.printf("s = %d%n", s);
32 System.out.printf("i = 0x0%n", i);
33 System.out.printf("l = 0b%s%n", Long.toBinaryString(l));

```

PrimitiveTypes.java

```
b = 0x7f  
s = -63  
i = 01654  
l = 0b1111111111111111111111111111111111111111111111111111111
```

Primitive Typen: Literale (Gleitkommazahlen)

- ▶ Literale für **float** und **double**
- ▶ Darstellung von Gleitkommazahlen:

$$V(\text{orzeichen}) \cdot M(\text{antisse}) \cdot B(\text{asis})^{E(\text{xponent})}$$

mit $b \in \{-1, 1\}$ und Basis = 2 oder Basis = 10

- ▶ Einfache **Dezimalpunkt-Darstellung**:

```
3.1415 // V=+1, M=3.1415, B=10, E=0
-.1415 // V=-1, M=0.1415, B=10, E=0
-3.    // V=-1, M=3.0,    B=10, E=0
```

- ▶ **Exponenten-Darstellung**: <Vorzeichen><Mantisse>e<Exponent> oder
<Vorzeichen><Mantisse>E<Exponent>

```
3.1415e4 // V=+1, M=3.1415, B=10, E=4
-.1415e-8 // V=-1, M=0.1415, B=10, E=-8
-3.E2    // V=-1, M=3.0    B=10, E=2
```

Primitive Typen: Literale (Gleitkommazahlen)

- ▶ **Hexadezimale Exponenten-Darstellung:** <Vorzeichen>0x<Mantisse>p<Exponent> oder <Vorzeichen>0x<Mantisse>P<Exponent>
- ▶ Mantisse und Exponent werden als **Hexadezimalzahlen** angegeben
- ▶ Exponent (nach p/P) ist **zwingend**
- ▶ **Basis = 2**
- ▶ Beispiele:

```
0x1.eadcp14      // V=+1, M=0xEADC,   B=2, E=0x14
0x1.84f3c6p-30   // V=-1, M=0x84F3C6,  B=2, E=-0x30
0x1.2cP8         // V=-1, M=1.2C,      B=2, E=0x8
```

- ▶ **Verwendung:** verlustfreie Definition von **float** und **double**-Zahlen (sonst eher selten)

```
0x1.ffffffffffffP1023 // Double.MAX_VALUE
0x1.0p-1024           // Double.MIN_VALUE
```

Primitive Typen: Literale (Gleitkommazahlen)

- ▶ Gleitkomma-Literal wird standardmäßig als **double** interpretiert
- ▶ Explizite Festlegung durch **Suffix**

Suffix	Bedeutung	Beispiel
f, F	float -Literal	3.14159f
d, D	double -Literal	3.14159265359d

- ▶ Oft gemachter Fehler:

```
float f = 3.1415;
```

- ▶ **Fehler:** „Type mismatch: cannot convert double to float“
- ▶ **Richtig:**

```
float f = 3.1415f;
```

Unterstriche in numerischen Literalen

- **Unterstriche** zwischen Ziffern zur Strukturierung von numerischen Literalen

```
long creditCardNumber = 1234_5678_9012_3456L;  
long socialSecurityNumber = 999_99_9999L;  
float pi = 3.14_15F;  
long hexBytes = 0xFF_EC_DE_5E;  
long hexWords = 0xCAFE_BABA;  
long maxLong = 0x7fff_ffff_ffff_ffffL;  
byte nybbles = 0b0010_0101;  
long bytes = 0b11010010_01101001_10010100_10010010;
```

- **Mehrere Unterstriche nebeneinander** sind erlaubt

```
long creditCardNumber = 1234__5678__9012__3456L;  
int longAnswer = 4_____2;
```

Unterstriche in numerischen Literalen

- ▶ Unterstriche sind **nur zwischen Ziffern** erlaubt, **nicht**
 - ▶ am Anfang oder Ende des Literals
 - ▶ neben einem Zeichen, das **keine Ziffer oder ein Unterstrich** ist
- ▶ Beispiele für ungültige Verwendung:

```
_3.1415f;    // _ am Anfang des Literals  
3_.1415f;    // _ neben .  
3.1415_f;    // _ neben Suffix  
2.14e_3;     // _ neben e  
0x_CAFE;     // _ neben x  
0_b101010;   // _ neben b
```

Inhalt

Datentypen Konvertierung

Konvertierung zwischen numerischen Typen

`byte < short, char < int < long < double`

- ▶ Konvertierung von „kleinerem zu größerem“ Datentyp

```
byte b = 21;  
int i = b;
```

- ▶ „widening primitive conversion“
- ▶ **Kein Problem:** implizit, keine explizite Konvertierung notwendig


- ▶ Konvertierung von „größerem zu kleinerem“ Datentyp

```
short s = 500;  
byte b = s; // Compiler-Fehler: possible loss of precision
```

- ▶ „narrowing primitive conversion“
- ▶ **Problem:** Informationsverlust, Compiler-Fehler
- ▶ Expliziter Cast notwendig:

```
short s = 500;  
byte b = (byte) s; // Informationsverlust!
```


Widening Conversion: Beispiel

```
54  runWideningConversionExample
55 byte b = 21;
56 short s = b;
57 int i = s;
58 long l = i;
59 float f = l;
60 double d = f;
61 println("b = " + b); // b = 21
62 println("s = " + s); // s = 21
63 println("i = " + i); // i = 21
64 println("l = " + l); // l = 21
65 println("f = " + f); // f = 21.0
66 println("d = " + d); // d = 21.0
```


 PrimitiveTypes.java

Narrowing Cast: Konvertierungsregeln

`byte < short, char < int < long < double`

- ▶ `double` → `float`: IEEE 754 Rundungsregeln
- ▶ `k`-Bit Ganzzahl → `l`-Bit Ganzzahl ($k > l$)
 - ▶ es werden nur die `l` niederwertigsten Bits verwendet
 - ▶ **Probleme**: Informationsverlust (sogar Vorzeichenwechsel möglich)
- ▶ `double, float` → `long/int`
 1. IEEE 754 Rundungsregeln
 2. wenn nach Runding zu groß bzw. klein → `Long/Integer.MAX_VALUE` bzw. `Long/Integer.MIN_VALUE`
 3. sonst gerundeter Wert
- ▶ `double, float` → `byte/char/short`
 1. Konvertierung nach `int` (s. oben)
 2. dann `int` → `byte/char/short` (s. oben)

Narrowing Conversion: Beispiel


```
72  runNarrowingConversionExample
73 double d = Math.pow(Math.PI, 20);
74 float f = (float) d; // expliziter cast
75 long l = (long) f;
76 int i = (int) l;
77 short s = (short) i;
78 byte b = (byte) s;
79 println("d = " + d); // d = 8.769956796082693E9
80 println("f = " + f); // f = 8.7699569E9
81 println("l = " + l); // l = 8769956864
82 println("i = " + i); // i = 180022272
83 println("s = " + s); // s = -5120
84 println("b = " + b); // b = 0
```

 PrimitiveTypes.java

Inhalt

Datentypen Überlauf

Primitive Typen: Ein Experiment

```
90  runOverflowExample
91 byte b = Byte.MAX_VALUE;
92 println("byte: ++b = " + (++b));
94 short s = Short.MAX_VALUE;
95 println("short: ++s = " + (++s));
97 int i = Integer.MAX_VALUE;
98 println("int: ++i = " + (++i));
100 long l = Long.MAX_VALUE;
101 println("long: ++l = " + (++l));
```

 PrimitiveTypes.java

Primitive Typen: Ein Experiment — das Ergebnis

```
byte: ++b = -128  
short: ++s = -32768  
int: ++i = -2147483648  
long: ++l = -9223372036854775808
```

- ▶ „overflow“ auf den niedrigsten Wert
- ▶ entsprechend „underflow“ auf höchsten Wert bei Subtraktion
- ▶ Java erzeugt **keinen Fehler** (Exception) bei einem Überlauf

Inhalt

Lokale Variablen

Variablendeklaration

Inhalt

Lokale Variablen

Variablendeklaration

Variablendeklaration

- ▶ **Lokale Variablen** in Methoden/Blöcken:

```
Datentyp Bezeichner [= Initialwert];
```

- ▶ **Datentyp**: primitiv oder Referenztyp
 - ▶ **Bezeichner**: siehe [↗ Bezeichner](#)
 - ▶ **Initialwert (optional)**: **Ausdruck** vom entsprechenden **Datentyp**
- ▶ Deklaration **mehrerer Variablen** vom gleichen Typ:

```
float alpha, beta, gamma;  
int f0 = 1, f1 = 1, f2 = f0 + f1;
```

- ▶ **Achtung**:

```
// nur gamma wird initialisiert!  
float alpha, beta, gamma = 1.234f;
```

Implizite Typendeklaration mit `var`

► Redundanz in Deklarationen

```
String s = "Hello World!";  
int i = 0;  
CelestialBody iss = new CelestialBody("ISS", 419_700d);
```

► Datentyp ist oft durch Initialwert festgelegt

► Vermeidung von Redundanz: `var`

```
var s = "Hello World!"; // String  
var i = 42; // int  
var iss = new CelestialBody("ISS", 419_700d); // CelestialBody
```

- `Compiler` ermittelt den passenden Typen
- Besonders praktisch für `Generics` (später)

```
var list1 = new List<String>(); // List<String>  
var list2 = List.of(1, 2.0f, 3.0d); // List<Number>
```

Hinweise zu var

- ▶ Trotz **var**: Variable hat **definierten Typ**
- ▶ **var** **kann** den Code lesbarer machen
- ▶ **var** nur verwenden, wenn **Typ ablesbar** ist
- ▶ **Negativbeispiele**

```
var f = (2.0f * 3.1415f) / 3.0;  
var c = customers.asList();  
var list2 = List.of(1, 1.0f, 1.0d);
```

- ▶ **Besser**

```
double twoThirdsOfPi = (2.0f * 3.1415f) / 3.0;  
List<Customer> customers = customers.asList();  
var numberList = List.of(1, 2.0f, 3.0d); // sprechender Name
```

Inhalt

Einschub: Ein- und Ausgabe auf der Konsole

Ausgabe über `print/ln` und `printf`

Eingabe über die `Scanner`-Klasse

Ein-/Ausgabe auf der Konsole

► Drei Datenströme

Java-Name	Typ	Bedeutung
System.out	PrintStream	Standardausgabe (stdout)
System.in	InputStream	Standardeingabe (stdin)
System.err	PrintStream	Fehlerausgabe (stderr)

- System.out für alle **erwarteten Ergebnisse**
- System.in für alle **Benutzereingaben** (oder anderen Quellen)
- System.err für alle **unerwarteten Ergebnisse/Fehlermeldungen**

Inhalt

Einschub: Ein- und Ausgabe auf der Konsole

Ausgabe über `print/ln` und `printf`

PrintStream.print/ln

- ▶ `print/println`: Ausgabe ohne/mit Zeilenvorschub
- ▶ `print/ln` sind für alle primitiven Typen und Objekte überladen
 - ▶ `print/ln(int x)`
 - ▶ `print/ln(boolean x)`
 - ▶ `print/ln(String x)`
 - ▶ ...

`runPrintExample`

```
9
10 System.out.print("Hello World!\n");
11 System.out.println(123);
12 System.out.print("Die Kreiszahl PI ist: ");
13 System.out.println(Math.PI);
```

 ConsoleIO.java

```
Hello World!
123
Die Kreiszahl PI ist: 3.141592653589793
```

PrintStream.println(): String-Konkatenation

► String-Konkatenation

- Strings lassen sich über den **+-Operator** aneinanderhängen

- Beispiel:

```
String s = "Die Kreiszahl PI ist " + Math.PI;
```

- Ist der **linke Operand** von + ein String so
 - wird der rechte Operand in einen String **umgewandelt**
 - und dann **konkateniert**

- Vorsicht!


```
System.out.println( "2+2 = " + 2 + 2);
```

```
2+2 = 22
```

- **Erstes +:** links String, rechts **int**, Ergebnis "2+2 = 2"
 - **Zweites +:** links String, rechts **int**, Ergebnis "2+2 = 22"
- Richtig:

```
System.out.println( "2+2 = " + (2 + 2));
```


PrintStream.println(): String-Konkatenation


```
19  runPrintConcatExample
20 double radius = 2.0f;
21 System.out.println(
22     "Ein Kreis mit Radius " + radius
23     + " hat eine Fläche von " + (Math.PI * radius * radius));
```

 ConsoleIO.java

Ein Kreis mit Radius 2.0 hat eine Fläche von 12.566370614359172

PrintStream.printf

- ▶ **printf**: Ausgabe mit Formatanweisungen
- ▶ **Signatur**: `printf(String format, Object... args)`

```
37  runPrintfExample  
38 double radius = 2.0;  
39 System.out.printf( "%d + %d = %d\n", 2, 2, 2 + 2);  
40 System.out.printf("Gravitationskonstante %e m^3/(kg*s^2)\n", ←  
    CelestialBody.GRAVITATIONAL_CONSTANT);  
41 System.out.printf("PI ist ungefähr: %f\n", Math.PI);  
42 System.out.printf(  
43     "Ein Kreis mit Radius %.2f hat eine Fläche von %.2f\n",  
44     radius, (Math.PI * radius * radius));
```

 ConsoleIO.java

```
2 + 2 = 4  
Gravitationskonstante 6,674300e-11 m^3/(kg*s^2)  
PI ist ungefähr: 3,141593  
Ein Kreis mit Radius 2,00 hat eine Fläche von 12,57
```

PrintStream.printf: Nützliche Formatanweisungen

	Beschreibung	Beispielausgabe
%b	boolean -Wert	true, false
%s	String-Repräsentation	Hello World!
%c	Unicode-Character	ü
%d	Dezimaldarstellung	1337
%x	Hexadezimaldarstellung	A3F
%e	Gleitkomma Exponentendarstellung	4.02114e2
%f	Gleitkomma Dezimaldarstellung	402.114
%.pf	<i>p</i> -Nachkommastellen	%.2f → 402.11
%%	Prozentzeichen	%
%n	Zeilenvorschub	

Inhalt

Einschub: Ein- und Ausgabe auf der Konsole


Eingabe über die Scanner-Klasse

System.in und die Scanner-Klasse

- ▶ **System.in** (Typ InputStream)
 - ▶ Zum Einlesen von **bytes**
 - ▶ Für primitive Typen **ungeeignet**
- ▶ **Scanner**-Klasse
 - ▶ **Interpretiert** Daten aus einem InputStream
 - ▶ Methoden zum Lesen von primitiven Datentypen

Typ	Methode	Eingabebeispiel
boolean	nextBoolean	true, false
byte	nextByte	-94
short	nextShort	1024
int	nextInt	64000
long	nextLong	2147483648
float/double	nextFloat/Double	3,1415
String	nextLine	Hello Java!<ENTER>

Scanner: Ein Beispiel

```
50  runSimpleScannerExample
51 var scanner = new Scanner(System.in);
53 System.out.println("Radius: ");
54 double radius = scanner.nextDouble();
56 scanner.nextLine();
58 System.out.println("Einheit: ");
59 String unit = scanner.nextLine();
61 System.out.printf(
62     "Kreisfläche: %.2f %s^2%n", (Math.PI * radius * radius), unit);
64 scanner.close();
```

 ConsoleIO.java

Scanner: Ein Beispiel (Ausgabe)

```
Radius:  
3,5  
Einheit:  
m  
Kreisfläche: 38,48 m^2
```

- ▶ Eingabe: 3,5\nm\n
- ▶ Scanner:
 - ▶ nextDouble: 3,5\nm\n
 - ▶ nextLine: 3,5\nm\n
 - ▶ nextLine: 3,5\nm\n

Operatoren

Zuweisungsoperator

Arithmetische Operatoren

Inkrement- und Dekrementoperator

Relationale Operatoren

Bit-Operatoren

Verbundoperatoren

Logische Operatoren

Cast-Operator

Konkatenations-Operator

instanceof-Operator

Bedingungsoperator

Rangfolge der Operatoren

Was sind Operatoren?

- ▶ Ein Operator
 - ▶ verknüpft
 - ▶ Operanden
 - ▶ zu einem Ergebnis
- ▶ Stelligkeit: Anzahl der Operanden

Stelligkeit	Typ	Beispiel	Anwendung
1	unär	!, negiert boolean	!b
2	binär	*, Multiplikation	3*4
3	ternär	?:, Bedingungsoperator	i<0 ? -1 : +1

Inhalt

Operatoren

Zuweisungsoperator

Zuweisungsoperator

LValue = Ausdruck

- ▶ Binär
- ▶ **Linker Operand**: etwas, das **Werte aufnehmen** kann (z.B. Variable)
- ▶ **Rechter Operand**: Ausdruck, vom gleichen Typ wie LValue
- ▶ **Operation**: weißt LValue den Wert des Ausdrucks zu
- ▶ **Ergebnis**: Wert von LValue nach Zuweisung
- ▶ **Beispiel**:

```
System.out.println(i = 2+2); // weißt i den Wert 4 zu
```

Ausgabe:

4

Mehrere Zuweisungen in einem Schritt

► Beispiel:

```
i = j = k = 4
```

► Auswertung:

1. **Äquivalent:** $i = (j = (k = 4))$
2. $k = 4$, Ergebnis 4
3. $j = (k=4)$ und damit $j = 4$, Ergebnis 4
4. $i = (j=4)$ und damit $i = 4$, Ergebnis 4

► Alle Variablen haben den Wert 4

► Negativbeispiel:

```
i = (j = 4) + 3 * (k = 1 + m);
```

► Besser:

```
j = 4;  
k = 1 + m;  
i = j + 3 * k;
```

Inhalt

Operatoren

Arithmetische Operatoren

Arithmetische Operatoren

Operator	Typ	Bedeutung	Beispiel
+	unär	unäres Plus	$+x$
-	unär	unäres Minus	$-x$
+	binär	Addition	$x+y$
-	binär	Subtraktion	$x-y$
*	binär	Multiplikation	$x*y$
/	binär	Division	x/y
%	binär	Modulo	$x\%y$

Auswertungsreihenfolge

► Auswertungsreihenfolge

1. Klammern zuerst
2. Negation/Identität
3. $*$ / $%$ im Ausdruck von links nach rechts
4. $+$ - im Ausdruck von links nach rechts

► Beispiele:

► $2 * 4 + 5 \% 3$

1. $2 * 4 = 8$

2. $5 \% 3 = 2$

3. $8 + 2 = 10$

► $(2 * 4 + 5) \% 3$

1. $2 * 4 = 8$

2. $8 + 5 = 13$

3. $13 \% 3 = 1$

► $10 * 4 / 5 \% 4$

1. $10 * 4 = 40$

2. $40 / 5 = 8$

3. $8 \% 4 = 0$


► $-(2 + 2) * 3$

1. $2 + 2 = 4$

2. -4

3. $-4 * 3 = -12$

Ein Experiment

```
108  runOverflowExample2
109 byte b = Byte.MAX_VALUE;
110 println("byte: b+1 = " + (b+1));
112 short s = Short.MAX_VALUE;
113 println("short: s+1 = " + (s+1));
115 int i = Integer.MAX_VALUE;
116 println("int: i+1 = " + (i+1));
118 long l = Long.MAX_VALUE;
119 println("long: l+1 = " + (l+1));
```

 PrimitiveTypes.java

Das Ergebnis

```
byte: b+1 = 128  
short: s+1 = 32768  
int: i+1 = -2147483648  
long: l+1 = -9223372036854775808
```


- ▶ Überlauf nur bei **int** und **long**
- ▶ Was passiert bei der Auswertung von $b+1$?
- ▶ „Type Promotion“ in arithmetischen Ausdrücken

Type Promotion

```
byte b = Byte.MAX_VALUE;  
println("byte: b = " + (b+1));
```

- ▶ Java wandelt numerische Typen in arithmetischen Ausdrücken automatisch um
- ▶ ...und bestimmt dadurch den Ergebnistyp des Ausdrucks
- ▶ **Grund:** Vermeidung von Informationsverlust
- ▶ **Regeln** (in dieser Reihenfolge)
 1. **byte, short** → **int**
 2. sobald ein **long**-Operand vorkommt → **long**
 3. sobald ein **float**-Operand vorkommt → **float**
 4. sobald ein **double**-Operand vorkommt → **double**

Ein Experiment — Auflösung

```
108  runOverflowExample2
109 byte b = Byte.MAX_VALUE;
110 println("byte: b+1 = " + (b+1));
112 short s = Short.MAX_VALUE;
113 println("short: s+1 = " + (s+1));
115 int i = Integer.MAX_VALUE;
116 println("int: i+1 = " + (i+1));
118 long l = Long.MAX_VALUE;
119 println("long: l+1 = " + (l+1));
```

 PrimitiveTypes.java

- ▶ (b+1), (s+1): b und s werden zu `int` promotet, **kein Überlauf**
- ▶ (i+1), (l+1): i und l behalten ihren Typ, **Überlauf**
- ▶ Frage: Was passiert mit (i+1L)? i wird zu `long`, **kein Überlauf**
- ▶ Noch eine Frage: Was passiert mit (l+1.0)? l wird zu `double`, **kein Überlauf**

Type Promotion: Beispiele

- ▶ **Variablen:** (Werte irrelevant)

```
byte b; short s; int i; long l; float f; double d;
```

- ▶ Beispiele

- ▶ $b + s : \text{int}$

- ▶ $i * l : \text{long}$

- ▶ $(b+s) * i : \text{int}$

- ▶ $(b+s) * l : \text{long}$

- ▶ $(b - s) / f : \text{float}$

- ▶ $d * (f + i) : \text{double}$

- ▶ Aufpassen bei **Zuweisungen**

```
byte b2 = 123;  
short s2 = b + b2; // incompatible types: possible lossy conversion from int ↵  
    to short
```

- ▶ ...und bei **var**

```
var mystery = b; // Typ: byte  
var mystery2 = b+b2; // Typ: int
```

Ganzzahlige Typen: + - * /

- ▶ + - * funktionieren wie erwartet
- ▶ **Aber** Überläufe beachten

```
int i = Math.MAX_VALUE * Math.MAX_VALUE; // == 1
```

- ▶ Division /:
 - ▶ **Integer-Division:** a / b = Ergebnis der Division ohne Nachkommastellen
 - ▶ **Beispiele:**
 - ▶ $4/2 = 2$
 - ▶ $5/3 = 1$
 - ▶ $-2/5 = 0$
 - ▶ $-13/3 = -4$

Ganzzahlige Typen: Modulo

- ▶ **Modulo:** $a \% b$ = ganzzahliger Rest nach Division
- ▶ **Definition** (als Javacode)

```
int p, q;          // ganze Zahlen mit q != 0
int k = p / q;     // Integer-Division
int l = p % q;
p == k * q + l;
```

- ▶ **Beispiele:**

- ▶ $8 \% 2 = 0$

- ▶ $23 \% 3 = 2$

- ▶ $-8 \% 2 = 0$

- ▶ $-58 \% 11 = -3$

- ▶ $15 \% -5 = 0$

- ▶ $19 \% -7 = 5$

- ▶ **Anwendungen von Modulo**


- ▶ **Teilbarkeit** (Vorzeichen beachten!)

```
i % 2 == 0 // i gerade
i % 2 == 1 // i ungerade
i % 3 == 0 // i durch 3 teilbar
```

- ▶ **Zyklisches Durchlaufen** eines Arrays

Ganzzahlige Typen: Division durch 0

- ▶ Was passiert bei einer **Division durch 0**?


```
8   runDivisionByZero  
9  int p = 10;  
10 int q = 0;  
11 System.out.printf("%d / %d = %d", p, q, p/q);
```

 Operators.java

- ▶ Java wirft eine **ArithmeticException** („/ by zero“)
- ▶ Kann prinzipiell **gefangen** werden
- ▶ **Besser**: Division durch 0 vermeiden

Gleitkommazahlen: + - * /

- ▶ Funktionieren wie erwartet
- ▶ Aufpassen auf Rechengenauigkeit

```
17  runFloatPrecisionExample
18 float f1 = 1.0f;
19 float f2 = 1.001f;
20 System.out.printf("%.15f%n", f2 - f1);
```

 Operators.java

0,001000046730042

- ▶ **float** oder **double** niemals verwenden, wenn Genauigkeit gefragt ist (z.B. für den Kontostand)
- ▶ Alternative: BigInteger

Gleitkommazahlen: Überlauf und spezielle Konstanten

- Gleitkomma-Konstanten in den Klassen `Float` und `Double`

Konstante	Bedeutung	Beispiel
<code>POSITIVE_INFINITY</code>	$+\infty$	<code>2.0*MAX_VALUE</code>
<code>NEGATIVE_INFINITY</code>	$-\infty$	<code>-1.0/0.0</code>
<code>NaN</code>	„not a number“	<code>0.0/0.0</code>

- Zur Erinnerung: `int` und Co. springen bei einem Überlauf an das andere Ende des Wertebereichs
- `float` und `double` springen auf die symbolischen Konstanten `POSITIVE_INFINITY` und `NEGATIVE_INFINITY`

Gleitkommazahlen: Rechenregeln

c ist echt positive **float**- oder **double**-Zahl

	0	c	$-c$	∞	$-\infty$
$\infty +$	∞	∞	∞	∞	NaN
$\infty -$	∞	∞	∞	NaN	∞
$\infty *$	NaN	∞	$-\infty$	∞	$-\infty$
$\infty /$	∞	∞	$-\infty$	NaN	NaN
$\infty \%$	NaN	NaN	NaN	NaN	NaN
$c +$	c	$2*c$	0.0	∞	$-\infty$
$c -$	c	0.0	$-2*c$	$-\infty$	∞
$c *$	0.0	$c*c$	$-c*c$	∞	$-\infty$
$c /$	∞	1.0	-1.0	0.0	-0.0 [sic!]
$c \%$	NaN	0.0	0.0	c	$-c$


Gleitkommazahlen: Rechenregeln

- ▶ Entsprechend für $-\infty$ (freiwillige Übung)
- ▶ NaN ergibt sich aus
 - ▶ den Fällen in vorheriger Tabelle
 - ▶ Jeder Operation mit NaN
 - ▶ bestimmten Aufrufen mathematischer Hilfsfunktionen, z.B.

```
Math.sqrt(-1); // == NaN
```


Gleitkommazahlen: Modulo

- ▶ **Modulo %** ist auch auf Gleitkommazahlen definiert
- ▶ Entspricht **fmod** aus C/C++
- ▶ Experiment:

```
27  runFloatModuloExample  
28 System.out.println(5.0 % 2.0); // 1.0  
29 System.out.println(5.25 % 2.0); // 1.25  
30 System.out.println(5.0 % 2.5); // 0.0  
31 System.out.println(5.25 % 2.5); // 0.25
```

 Operators.java

Gleitkommazahlen: Modulo nachimplementiert

```
40 public static double fmod(double p, double q){
41     double d = truncate(p / q); // verwirft Nachkommastellen
42     return p - d * q;
43 }
44
45 public static void floatManualModuloExample() {
46      runFloatManualModulo
47     System.out.println(fmod(5.0, 2.0)); // 1.0
48     System.out.println(fmod(5.25, 2.0)); // 1.25
49     System.out.println(fmod(5.0, 2.5)); // 0.0
50     System.out.println(fmod(5.25, 2.5)); // 0.25
51
52 }
```

 Operators.java

Inhalt


Operatoren

Inkrement- und Dekrementoperator

Inkrement- und Dekrementoperator

```
i++; i--; ++i; --i;
```

- ▶ Unär
- ▶ **Operand:** LValue, numerischer Typ
- ▶ **Operation:**
 - ▶ ++ weißt i den Wert i+1 zu
 - ▶ -- weißt i den Wert i-1 zu
- ▶ **Ergebnis:**
 - ▶ alter Wert bei i++ und i--
 - ▶ neuer Wert bei ++i und --i
- ▶ **Beispiel:**

```
57  runIncrementDecrementExample  
58 int i = 0;  
59 System.out.printf("i++ : %d\n", i++); // 0  
60 System.out.printf("i : %d\n", i); // 1  
61 System.out.printf("--i : %d\n", --i); // 0  
62 System.out.printf("i : %d\n", i); // 0
```

 Operators.java

Inkrement und Dekrement sind atomar

- ▶ **Hinweis:** `i++` wird **nicht** mit `i=i+1` implementiert!
- ▶ Inkrement `i++` als Bytecode:

```
iinc i, 1 // i um 1 erhöhen
```

- ▶ Inkrement `i=i+1` als Bytecode:

```
iload i // i laden  
iconst 1 // 1 laden  
iadd // addieren  
istore i // in i speichern
```

- ▶ Eine Operation vs. vier Operationen
 - ▶ `i++` kann **nicht unterbrochen** werden (Threads, Programmieren III)
 - ▶ Bei `i=i+1` findet evtl. **Type Promotion** statt, bei `i++` nicht
- ▶ Entsprechendes gilt auch für `i--`; `++i`; `--i`

Inhalt

Operatoren


Relationale Operatoren

Relationale Operatoren

```
x == y, x != y
```

- ▶ Binär
- ▶ Operanden: primitive Typen oder Referenzen
- ▶ Operation: prüft auf Gleichheit
 - ▶ primitive Typen: Wertgleichheit
 - ▶ Referenzen: Gleichheit der Referenz
- ▶ Ergebnis:
 - ▶ `==` **true** bei Gleichheit, sonst **false**
 - ▶ `!=` **false** bei Gleichheit, sonst **true**


Gleichheitsoperatoren: Ganzzahlige Typen

```
68  runIntEqualityExample
69 int i = 42;
70 byte b = 42;
72 System.out.printf("i == 42 : %b%n", i == 42); // true
73 System.out.printf("i == b : %b%n", i == b); // true
```

 Operators.java

- ▶ Type Promotion vor dem Vergleich
- ▶ b wird zu **int** vor Vergleich

Gleichheitsoperatoren: Gleitkommazahlen


```
91  runFloatEqualityExample
92 float f1 = 1f;
93 float f2 = 1.001f;
95 System.out.printf("f1 == 1f : %b\n", f1 == 1f); // true
96 System.out.printf("f2 - f1 == 0.001 : %b\n",
97     (f2-f1 == 0.001f)); // false
```

 Operators.java

- ▶ **Achtung:** Rechenungenauigkeiten!
- ▶ Gleitkommazahlen ($\neq \pm\infty$) **niemals** mit == oder != vergleichen!
- ▶ Besser:

```
boolean approx(double f1, double f2, double eps){
    return Math.abs(f1-f2) < eps;
}
```

Gleichheitsoperatoren: Referenzen

```
79  runReferenceEqualityExample  
80 CelestialBody iss = new CelestialBody("ISS", 419_700d);  
81 CelestialBody iss2 = iss;  
82 CelestialBody issDup = new CelestialBody("ISS", 419_700d);  
84 System.out.printf("iss == iss2 : %b%n", iss == iss2); // true  
85 System.out.printf("iss == issDup: %b%n", iss == issDup); // false
```

 Operators.java

- ▶ Hinweis: **Strings** nicht mit `==` oder `!=` vergleichen
- ▶ Besser:

```
string s1, s2;  
if (s1.equals(s2))  
    // ...
```

Relationale Operatoren


$x < y$, $x \leq y$, $x > y$, $x \geq y$

- ▶ Binär
- ▶ **Operanden**: numerische primitive Typen
- ▶ **Operation**: prüft Relation
- ▶ **Ergebnis**: **true** wenn Relation gilt, sonst **false**
- ▶ **Type Promotion** vor Vergleich:

```
byte b; int i;  
if (b < i) // b wird für Vergleich zu int  
    // ...
```

```
long l; double d;  
if (l >= d) // l wird für Vergleich zu double  
    // ...
```

Hinweise zu Gleitkommazahlen

```
104  runFloatInequalityExample
105 float f1 = 1f;
106 float f2 = 1.001f;
108 System.out.printf("f2 - f1 > 0.001 : %b%n",
109     (f2-f1 > 0.001f)); // true
```

 Operators.java

- ▶ **Rechenungenauigkeiten** beachten!
- ▶ Vergleiche mit POSITIVE_INFINITY und NEGATIVE_INFINITY
 - ▶ POSITIVE_INFINITY ist **größer** als jeder Wert
 - ▶ NEGATIVE_INFINITY ist **kleiner** als jeder Wert
- ▶ Vergleich mit NaN liefern **immer false**

Inhalt

Operatoren


Bit-Operatoren

Bit-Operatoren

- ▶ Manipulation von Bits in ganzzahligen Werten
- ▶ Übersicht:

Op.	Typ	Beschreibung	Beispiel
~	unär	Negation	$\sim 0b001100 == 0b110011$
&	binär	Und	$0b0011 \& 0b0101 == 0b0001$
	binär	Oder	$0b0011 0b0101 == 0b0111$
^	binär	exklusives Oder	$0b0011 \wedge 0b0101 == 0b0110$
<<	binär	Linksverschiebung	$0b0000_1011 \ll 3 == 0b0101_1000$
>>	binär	Rechtsverschiebung	$0b0000_1011 \gg 3 == 0b0000_0001$


Bit-Operatoren Anwendung: Bitmasken I

```
119  runBitmaskExample
120 final int OPTION_1 = 1 << 0;
121 final int OPTION_2 = 1 << 1;
122 final int OPTION_3 = 1 << 2;
124 System.out.printf("OPTION_1 = %s\n", toBinary(OPTION_1));
125 System.out.printf("OPTION_2 = %s\n", toBinary(OPTION_2));
126 System.out.printf("OPTION_3 = %s\n", toBinary(OPTION_3));
128 int selection = OPTION_2 | OPTION_3;
129 System.out.printf("selection = %s\n", toBinary(selection));
131 int inverted = ~selection;
132 System.out.printf("inverted = %s\n", toBinary(inverted));
134 int anotherSelection = OPTION_1 | OPTION_3;
136 int union = selection | anotherSelection;
137 System.out.printf("union = %s\n", toBinary(union));
139 int intersection = selection & anotherSelection;
140 System.out.printf("intersection = %s\n", toBinary(intersection));
```

```
OPTION_1 = 1  
OPTION_2 = 10  
OPTION_3 = 100  
selection = 110  
inverted = 1111111111111111111111111111001  
union = 111  
intersection = 100
```

Bit-Operatoren Anwendung: Multiplikation/Division

- ▶ $i \ll j$ entspricht Multiplikation mit 2^j
- ▶ $i \gg j$ entspricht Division durch 2^j
- ▶ Beispiel:

```
147  runBitMultiplicationExample
148 int i = 1337;
149 System.out.printf("D: %8d B: %s\n", i, toBinary(i));
150 System.out.printf("D: %8d B: %s\n", i << 5, toBinary(i << 5));
151 System.out.printf("D: %8d B: %s\n", i >> 5, toBinary(i >> 5));
```

 Operators.java

```
D:      1337 B: 10100111001
D:     42784 B: 1010011100100000
D:         41 B: 101001
```

- ▶ **Hinweis:** \gg verwendet das erste Bit (Vorzeichen) um die linke Seite damit aufzufüllen
- ▶ \ggg füllt die linke Seite mit Nullen auf

Inhalt

Operatoren


Verbundoperatoren

Verbundoperatoren

`+=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, |=, ^=`

- ▶ Binär
- ▶ Linker Operand: LValue (z.B. Variable)
- ▶ Rechter Operand: Ausdruck
- ▶ Operation: LValue $\text{?} = x$
 - ▶ Auswertung von LValue ? x
 - ▶ Zuweisung des Resultats an LValue
- ▶ Ergebnis: neuer Wert von LValue
- ▶ Hinweis:
 - ▶ rechter Operand (x oben) wird **zuerst ausgewertet**
 - ▶ $x \text{ *= } y + z$ entspricht $x = x * (y+z)$ und **nicht** $x = x * y + z$

Verbundoperatoren: Beispiel

```
159  runIntAssignmentOperationExample
160 int i = 1;
161 i += 1;      // i == 2
162 i *= (i+1);  // i = i * (i+1) = 2 * 3 = 6
163 i <<= 2;     // i = 4 * i = 24
164 i |= 0b1;    // i == 25
```


 Operators.java

25

Verbundoperatoren: Übersicht

Operator	Interpretation	Beispiel	
$x += y$	$x = x + y$	$x += 3.1415$	$x = x + 3.1415$
$x -= y$	$x = x - y$	$x -= y + z$	$x = x - (y + z)$
$x *= y$	$x = x * y$	$x *= y + z$	$x = x * (y + z)$
$x /= y$	$x = x / y$	$x /= y * z$	$x = x / (y * z)$
$x \% = y$	$x = x \% y$	$x \% = y$	$x = x \% y$
$x << = y$	$x = x << y$	$x << = 2$	$x = x << 2$
$x >> = y$	$x = x >> y$	$x >> = 5$	$x = x >> 5$
$x >>> = y$	$x = x >>> y$	$x >>> = 5$	$x = x >>> 5$
$x \& = y$	$x = x \& y$	$x \& = (y z)$	$x = x \& (y z)$
$x = y$	$x = x y$	$x = (y \& z)$	$x = x (y \& z)$
$x \wedge = y$	$x = x \wedge y$	$x \wedge = (y \wedge z)$	$x = x \wedge (y \wedge z)$

Verbundoperatoren: Die ganze Wahrheit

```
172  runAssignmentOperationExample  
173 int i = 0;  
174 i += Math.PI;  
175 System.out.printf("%d%n", i); // 3
```

 Operators.java

► Bytecode

```
iload      i  
i2d        // int -> double  
load       Math.Pi  
dadd       // Addition  
d2i        // double -> int  
istore i    // i speichern
```

► Somit entspricht `i += Math.PI`

```
i = (int) ((double) i + Math.PI);
```

► Konvertierung über explizite (evtl. verlustbehaftete) **Casts**

Inhalt

Operatoren

Logische Operatoren

Logische Operatoren

!, &&, ||, ^

- ▶ **!** unär, &&, ||, ^ binär
- ▶ **Operanden**: boolesche Ausdrücke
- ▶ **Operation**: wertet die boolesche Aussage aus
- ▶ **Ergebnis**: Ergebnis der booleschen Aussage

a	b	!a	a && b	a b	a ^ b
false	false	true	false	false	false
false	true	true	false	true	true
true	false	false	false	true	true
true	true	false	true	true	false


Logische Operatoren: Ein Beispiel

```
181 public static boolean isEven(int i){  
182     boolean isEven = (i % 2 == 0);  
183     System.out.printf("isEven(%d) == %b\n", i, isEven);  
184     return isEven;  
185 }
```

Operators.java

- ▶ Gibt **true** zurück wenn **i gerade** ist, sonst **false**
- ▶ Ausgabe um Aufrufe nachzuvollziehen

Logische Operatoren: Ein Beispiel

```
190  runLogicOperatorsExample
191 int two = 2, five = 5, nine = 9;
192 boolean result;
194 result = !isEven(five);
195 System.out.printf("!isEven(five): %b%n%n", result);
197 result = isEven(two) && isEven(five);
198 System.out.printf("isEven(two) && isEven(five): %b%n%n", result);
200 result = isEven(five) && isEven(nine);
201 System.out.printf("isEven(five) && isEven(nine): %b%n%n", result);
203 result = isEven(two) || !isEven(nine);
204 System.out.printf("isEven(two) || !isEven(nine): %b%n%n", result);
206 result = isEven(two) ^ isEven(nine);
207 System.out.printf("isEven(two) ^ !isEven(nine): %b%n%n", result);
```

 Operators.java

Logische Operatoren: Ein Beispiel

```
isEven(5) == false  
!isEven(five): true
```

```
isEven(2) == true  
isEven(5) == false  
isEven(two) && isEven(five): false
```

```
isEven(5) == false  
isEven(five) && isEven(nine): false
```

```
isEven(2) == true  
isEven(two) || !isEven(nine): true
```

```
isEven(2) == true  
isEven(9) == false  
isEven(two) ^ !isEven(nine): false
```

Evaluation logischer Operatoren

- ▶ Erkenntnis: Ein Operand wird nur ausgewertet, wenn sich das Endergebnis noch ändern kann
- ▶ `x && y`: `x == false` \Rightarrow `y` wird nicht ausgewertet
- ▶ `x || y`: `x == true` \Rightarrow `y` wird nicht ausgewertet
- ▶ `x ^ y`: beide Operanden werden immer ausgewertet
- ▶ `&&` und `||` heißen Kurzschluss-Operatoren
- ▶ Achtung: bei Methodenaufrufen in `if` nie auf die Ausführung verlassen

```
if (x > 10 && importantMethod())  
    /* ... */
```


Methode wird nicht aufgerufen wenn `x <= 10`

- ▶ Besser:

```
boolean result = importantMethod();  
if (x > 10 && result)  
    /* ... */
```

Nicht-Kurzschluss-Operatoren

- ▶ Was ist, wenn Kurzschluss **nicht erwünscht** ist?
- ▶ Nicht-Kurzschluss-Operatoren & und |
- ▶ Beispiel: [Insel, S. 154]

```
215  runNonBypassLogicOperatorsExample  
216 int a = 0, b = 0, c = 0, d = 0;  
217 System.out.println( true || a++ == 0 ); // true, a nicht erhöht  
218 System.out.println( a ); // 0  
219 System.out.println( true | b++ == 0 ); // true, b erhöht  
220 System.out.println( b ); // 1  
221 System.out.println( false && c++ == 0 ); // false, c nicht erhöht  
222 System.out.println( c ); // 0  
223 System.out.println( false & d++ == 0 ); // false, d erhöht  
224 System.out.println( d ); // 1
```

 Operators.java

Inhalt

Operatoren

Cast-Operator

Cast-Operator

(Typ) Ausdruck

- ▶ Binär
- ▶ **Linker Operand**: Typ, z.B. **int**, CelestialBody
- ▶ **Richter Operand**: Ausdruck
- ▶ **Operation**: wandelt das Ergebnis des Ausdrucks in den angegebenen Typ um
- ▶ **Ergebnis**: umgewandelter Ausdruck
- ▶ Beispiele:

```
int i = (int) Math.PI; // verlustbehaftet
byte b = (byte) 1;     // verlustfrei
CelestialBody iss = (String) "ISS"; // FEHLER: nicht möglich
```

Cast-Operator

- ▶ **Primitive Typen:** siehe Folie 56
 - ▶ `byte < short, char < int < long < double`
 - ▶ von **kleinerem** zu **größerem Typ**: kein Cast notwendig
 - ▶ von **größerem** zu **kleinerem Typ**: Cast notwendig (**Informationsverlust!**)
 - ▶ `boolean` kann in **keine Richtung** gewandelt werden
- ▶ **Referenztypen:** später
- ▶ **Zwischen primitiven und Referenztypen:** nicht möglich

```
String s = (String) 42; // FEHLER  
double rock = (double) new CelestialBody("rock", 10); // FEHLER
```

Inhalt

Operatoren

Konkatenations-Operator


Konkatenations-Operator

s1 + s2

- ▶ Binär
- ▶ **Operand**: Strings oder Typen, die in Strings umgewandelt werden können
- ▶ **Operation**: hängt die Operanden als Strings hintereinander (**Konkatenation**)
- ▶ **Ergebnis**: konkatenierter String
- ▶ **Auswertungsreihenfolge**: von links nach rechts

```
System.out.println("2+2 = " + 2 + 2);    // 2+2 = 22  
System.out.println("2+2 = " + (2 + 2));  // 2+2 = 4
```

Konkatenations-Operator: Beispiel

```
230  runConcatenationExample  
231 System.out.println("Hello" + " " + "World!");  
232 System.out.println("Antwort: " + 42);  
233 System.out.println(381 + " ist durch 3 teilbar: " + (381%3==0));  
235 CelestialBody superman = new CelestialBody("Superman", 100);  
236 System.out.println("It's a bird, it's a plane, it's " + superman);
```

 Operators.java

```
Hello World!  
Antwort: 42  
381 ist durch 3 teilbar: true  
It's a bird, it's a plane, it's de.hawlandshut.java1.basics.CelestialBody@28bbb6ac
```

Inhalt

Operatoren

instanceof-Operator


instanceof-Operator

Objekt **instanceof** Referenztyp

- ▶ Binär
- ▶ Linker Operand: Referenz
- ▶ Rechter Operand: Referenztyp
- ▶ Operation: Prüft ob Objekt eine Instanz von Referenztyp ist
- ▶ Ergebnis: **true** wenn das der Fall ist, sonst **false**
- ▶ Referenztyp kann Bezeichner einer Klasse oder eines Interfaces sein (später)
- ▶ **instanceof** berücksichtigt die Ableitungshierarchie
- ▶ Gilt obj **instanceof** Typ, so kann obj auf Typ gecastet werden

```
Typ t = (Typ) obj; // möglich da obj instance of Typ
```


instanceof-Operator: Beispiel

```
241  runInstanceOfExample
242 public static void instanceofExample(Object mystery) {
243     boolean result;
244     System.out.printf("%nmystery: %s%n", mystery);
246     result = mystery instanceof Object;
247     System.out.printf("mystery instanceof Object: %b%n", result);
249     result = mystery instanceof String;
250     System.out.printf("mystery instanceof String: %b%n", result);
252     result = mystery instanceof Double;
253     System.out.printf("mystery instanceof Double: %b%n", result);
255     result = mystery instanceof Number;
256     System.out.printf("mystery instanceof Number: %b%n", result);
257 }
```

 Operators.java

instanceof-Operator: Beispiel

```
instanceOfExample("Hello World!");  
instanceOfExample((Double) 3.1415); // aka new Double(3.1415)
```

```
mystery: Hello World  
mystery instanceof Object: true  
mystery instanceof String: true  
mystery instanceof Double: false  
mystery instanceof Number: false  
  
mystery: 3.1415  
mystery instanceof Object: true  
mystery instanceof String: false  
mystery instanceof Double: true  
mystery instanceof Number: true
```

Hinweis: Double leitet von Number ab

Inhalt

Operatoren

Bedingungsoperator

Bedingungsoperator

Bedingung ? Ausdruck1 : Ausdruck2

- ▶ Ternär
- ▶ 1. Operand: Bedingung, **boolescher Ausdruck**
- ▶ 2. Operand: Ausdruck1, **Ergebnis im positiven Fall**
- ▶ 3. Operand: Ausdruck2, **Ergebnis im negativen Fall**
- ▶ Operation:
 - 1. Auswertung der **Bedingung**
 - 2. **Bedingung positiv**: Auswertung von Ausdruck1
 - 3. **Bedingung negativ**: Auswertung von Ausdruck2
- ▶ **Ergebnis**: Ergebnis von Ausdruck1 im positiven Fall, sonst Ergebnis von Ausdruck2
- ▶ Ausdruck1 und Ausdruck2 müssen den **gleichen Typ** haben
- ▶ **Hinweis**: Ausdruck1 wird nur im **positiven Fall** ausgewertet
- ▶ entsprechend Ausdruck2 nur im **negativen Fall**

Bedingungsoperator: Beispiel

262  **runConditionalOperatorExample**

```
263 int i = 5, j = 10, k = 7;  
265 String text = i % 2 == 0 ? "gerade" : "ungerade";  
266 System.out.printf("i ist %s\n", text);  
268 boolean largerIsEven = i < j ? isEven(j) : isEven(i);  
269 System.out.printf("Die größere Zahl ist gerade: %b\n", largerIsEven);  
271 int max = i < j ? (k < j ? j : k) : (i < k ? k : i);  
272 System.out.printf("Größte Zahl: %d\n", max);
```

 Operators.java

```
i ist ungerade  
isEven(10) == true  
Die größere Zahl ist gerade: true  
Größte Zahl: 10
```

Bedingungsoperator: Ergänzung

► (Fehlerhaftes) Beispiel:

```
int evenNumber, oddNumber;  
int i = 5;  
(i % 2 == 0 ? evenNumber : oddNumber) = i;
```

- **Fehler:** „Left-hand side of assignment must be a variable.“
- Bedingungsoperator liefert **keinen LValue...**
- ...sondern den **Wert des Ausdrucks**
- **Alternative:**

```
int evenNumber, oddNumber;  
int i = 5;  
if (i % 2 == 0)  
    evenNumber = i;  
else  
    oddNumber = i;
```

Inhalt

Operatoren

Rangfolge der Operatoren

Rangfolge der Operatoren

- ▶ Beispiel:

```
i << j << k
```

- ▶ In welcher Reihenfolge werden die Operatoren ausgewertet?

- ▶ Assoziativität eines Operators
- ▶ $(i << j) << k$ links-assoziativ (\rightarrow)
- ▶ $i << (j << k)$ rechts-assoziativ (\leftarrow)

- ▶ Java: << ist links-assoziativ

- ▶ Noch ein Beispiel:

```
i << j + k
```

- ▶ Was ist hier die Reihenfolge?

- ▶ Rangfolge zwischen Operatoren
- ▶ $(i << j) + k$ << hat höheren Rang
- ▶ $i << (j + k)$ + hat höheren Rang

- ▶ Java: + hat höheren Rang als <<

Rangfolge der Operatoren

#	Op.	Beschreibung	Ass.
16	[]	Array-Zugriff	→
	.	Member-Zugriff	
	()	Klammeroperator	
15	++	Post-Inkrement	—
	--	Post-Dekrement	
14	++	Pre-Inkrement	←
	--	Pre-Dekrement	
	+	unäres Plus	
	-	unäres Minus	
	!	Negation	
	~	bitweise Neg.	

Rangfolge der Operatoren

#	Op.	Beschreibung	Ass.
13	()	Cast	←
	new	Obj.erzeugung	
12	* / %	Arithmetik	→
11	+ -	Arithmetik	→
	+	Konkatenation	
10	<< >>	Bitshift	→
	>>>		
9	< <=	Relationen	—
	< >=		
	instanceof		
8	== !=	Gleichheit	→

Rangfolge der Operatoren

#	Op.	Beschreibung	Ass.
7	&	bitweises Und	→
6	^	bitweises XOR	→
5		bitweises Oder	→
4	&&	logisches Und	→
3		logisches Oder	→
2	?:	Bedingungsoperator	←
1	= += -= *= /= %= &= ^= = <<= >>= >>>=	Zuweisungen	←

Rangfolge der Operatoren: (Unrealistische!) Beispiele

- ▶ `i << j >> 1` → `(i << j) >> 1`
- ▶ `(byte)(short)(int) 42L` → `(byte)((short)((int) 42L))`
- ▶ `"" + 2*2 << 1` → `"" + (2*2) << 1`
Fehler: `<<` auf String nicht definiert
- ▶ `w ^ !x && y || !z` → `((w ^ (!x)) && y) || (!z)`
- ▶ `i += ~++i >>> 1` → `i += ((~(++i)) >>> 1)`

Rangfolge der Operatoren: Praxis

- ▶ Wer kann sich all diese Regeln **merken**?
 - ▶ Für die **Klausur**?
 - ▶ In der **Praxis**?
 - ▶ Sie?
 - ▶ Ich auch nicht!
- ▶ In der Praxis
 - ▶ Komplexe Ausdrücke **aufteilen**: `1 << 1 | 1 << 2 == 3`

```
int i = 1 << 1;  
int j = 1 << 2;  
if (i | j == 3)  
    /* ... */
```

- ▶ **Klammern verwenden** (selbst wenn nicht notwendig):

```
(a | !b) && (d || !c)
```

Inhalt

if-then-else, switch-case: Bedingte Ausführung

if-then-else

switch-case

Inhalt

if-then-else, switch-case: Bedingte Ausführung

if-then-else

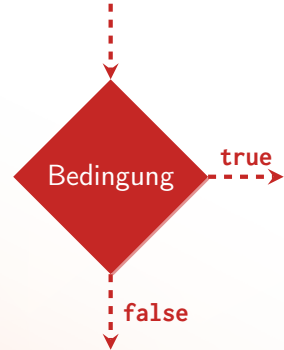
if-then-else: Bedingte Ausführung

- ▶ Im Folgenden sei **Bedingung**
 - ▶ ein **boolescher Ausdruck**
 - ▶ d.h. ein Ausdruck, der nach der Auswertung **true** oder **false** liefert
 - ▶ **Beispiele:**

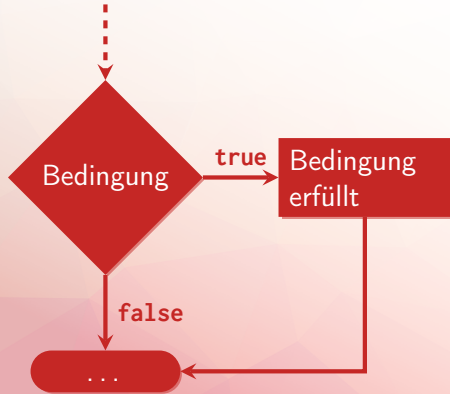
```
i > 0  
!customerList.isEmpty()  
(i % 2 == 1) && (i % 3 == 0)
```

- ▶ Allgemeine Form der **if**-Anweisung

```
if (Bedingung)  
    // Anweisung für Bedingung == true  
else  
    // Anweisung für Bedingung == false
```

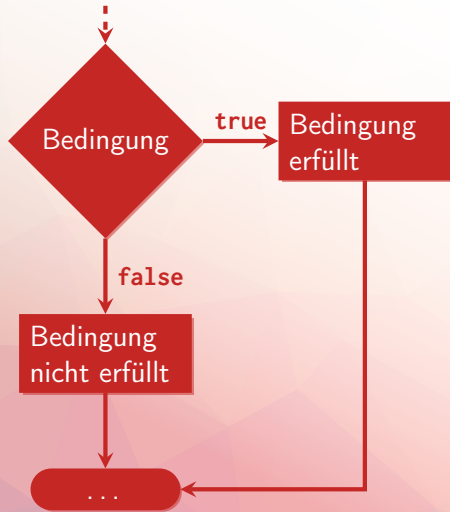


if-then: Einfacher Fall



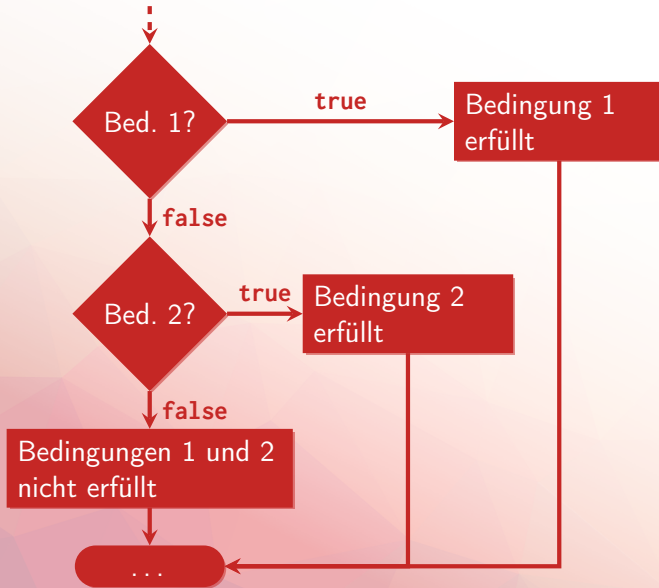
```
if (Bedingung)  
    Bedingung erfüllt
```

if-then-else: Vollständiger Fall



```
if (Bedingung)
    Bedingung erfüllt
else
    Bedingung nicht erfüllt
```

if-then-else if-else: Mehrfachverzweigung

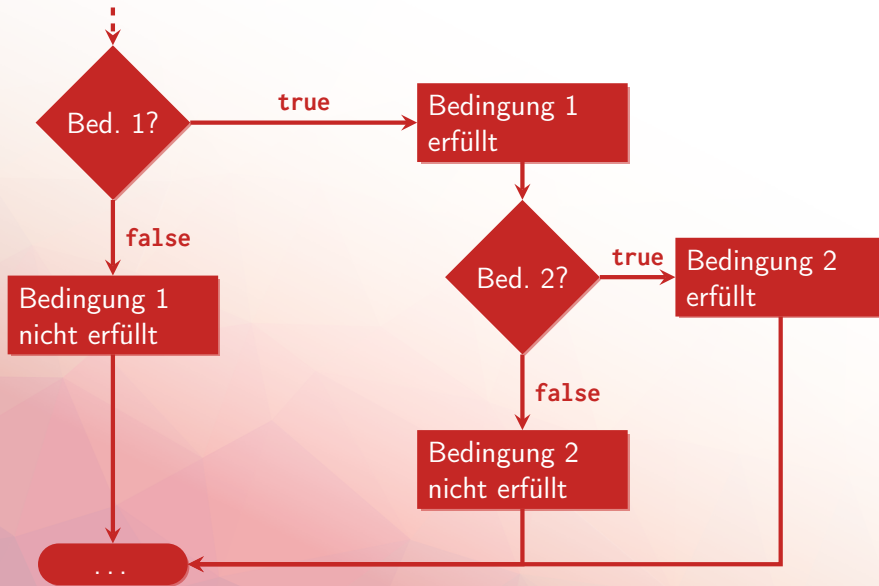


if-then-else if-else: Mehrfachverzweigung

```
if (Bedingung 1)
    Bedingung 1 erfüllt
else if (Bedingung 2)
    Bedingung 2 erfüllt (aber nicht Bedingung 1)
else
    Bedingungen 1 und 2 nicht erfüllt
```

```
if (Bedingung 1)
    Bedingung 1 erfüllt
else if (Bedingung 2)
    Bedingung 2 erfüllt (aber nicht Bedingung 1)
/* ... */
else if (Bedingung n)
    Bedingung n erfüllt (aber nicht Bedingungen 1 bis (n-1))
else
    keine Bedingung erfüllt
```


if-then-else: Verschachtelung



if-then-else: Verschachtelung


```
if (Bedingung 1){  
    Bedingung 1 erfüllt  
    if (Bedingung 2)  
        Bedingungen 1 und 2 erfüllt  
    else  
        Bedingung 1 erfüllt (aber nicht Bedingung 2)  
}  
else  
    Bedingung 1 nicht erfüllt
```

if-then-else: Fehlerquellen

```
11  runBadIfExample1  
12 if (now.get(Calendar.YEAR) == 2050 && now.get(Calendar.MONTH) == Calendar.MARCH);  
13     System.out.println("We are living in the future!");
```


 IfThenElse.java

- ▶ **Semikolon** am Ende der **if**-Anweisung:
 - ▶ Die auszuführende Anweisung im positiven Fall ist **leer**
 - ▶ Die **nachfolgende Anweisung** wird **immer ausgeführt**
- ▶ **Abhilfe**: Lange Bedingungen vereinfachen

```
21  runImprovedIfExample1  
22 boolean is2050 = now.get(Calendar.YEAR) == 2050;  
23 boolean isMarch = now.get(Calendar.MONTH) == Calendar.MARCH;  
24 if (is2050 && isMarch)  
25     System.out.println("We are living in the future!");
```


 IfThenElse.java

if-then-else: Fehlerquellen

```
32  runBadIfExample2
33 int i = 13, j = 2020;
34 if (i > 10 && i > j)
35     System.out.println("i is greater than 10");
36     System.out.println("i is greater than j");
```


 IfThenElse.java

- ▶ **if**-Anweisung akzeptiert nur **eine Anweisung**:
 - ▶ Das erste System.out.println wird im **positiven Fall** ausgeführt
 - ▶ Das zweite System.out.println wird **immer ausgeführt**
- ▶ **Abhilfe**: Immer Blöcke bilden

```
43  runImprovedIfExample2
44 int i = 13, j = 2020;
45 if (i > 10 && i > j){
46     System.out.println("i is greater than 10");
47     System.out.println("i is greater than j");
48 }
```


 IfThenElse.java

if-then-else: Fehlerquellen

```
55  runBadIfExample3
56 int i = 13, j = 2020;
57 if (i > 10)
58     if (i > j)
59         System.out.println("i > 10 && i > j");
60 else
61     System.out.println("i <= 10");
```

 IfThenElse.java

- ▶ Ein **else**-Zweig wird der **nächst innersten if-Anweisung** zugeordnet (wenn keine Blöcke vorhanden sind)
- ▶ **Abhilfe**: Wieder Blöcke bilden

```
69  runImprovedIfExample3
70 if (i > 10){
71     if (i > j)
72         System.out.println("i > 10 && i > j");
73 }
74 else
75     System.out.println("i <= 10");
```

 IfThenElse.java

if-then-else: Fehlerquellen


```
if (Bedingung1){  
    if (Bedingung2){  
        if (Bedingung3){  
            // ...  
        } else {  
            // ...  
        }  
    }else{  
        // ...  
    }  
}
```

- ▶ **Verschachtelungstiefe** (3 in Beispiel)
- ▶ „Code Smell“: Macht Code
 - ▶ unlesbar
 - ▶ schwer wartbar
 - ▶ fehleranfällig
- ▶ **Abhilfe:**
 - ▶ Refactoring
 - ▶ z.B. Auslagern in Methoden

Inhalt

if-then-else, switch-case: Bedingte Ausführung
switch-case

Warum switch-case? I


```
11  runPrintMonthDaysIf
12 public static void printMonthDaysIf(int month, boolean isLeapYear){
13     if (month == Calendar.JANUARY
14         || month == Calendar.MARCH
15         || month == Calendar.MAY
16         || month == Calendar.JULY
17         || month == Calendar.AUGUST
18         || month == Calendar.OCTOBER
19         || month == Calendar.DECEMBER){
20         System.out.println("31 Tage");
21     }else if (month == Calendar.APRIL
22         || month == Calendar.JUNE
23         || month == Calendar.SEPTEMBER
24         || month == Calendar.NOVEMBER){
25         System.out.println("30 Tage");
26     }else if (month == Calendar.FEBRUARY) {
27
28     }
29
30     if (isLeapYear){
31         System.out.println("29 Tage");
```

Warum switch-case? II

```
32     } else {  
33         System.out.println("28 Tage");  
34     }  
35 }else{  
36     System.out.println("Ungültiger Monat");  
37 }  
38 }
```

SwitchCase.java


Darum switch-case! I

```
42  runPrintMonthDaysSwitch
43 public static void printMonthDaysSwitch(int month, boolean isLeapYear){
44     switch (month){
45         case Calendar.JANUARY:
46         case Calendar.MARCH:
47         case Calendar.MAY:
48         case Calendar.JULY:
49         case Calendar.AUGUST:
50         case Calendar.OCTOBER:
51         case Calendar.DECEMBER:
52             System.out.println("31 Tage");
53             break;
54
55         case Calendar.APRIL:
56         case Calendar.JUNE:
57         case Calendar.SEPTEMBER:
58         case Calendar.NOVEMBER:
59             System.out.println("30 Tage");
60             break;
```

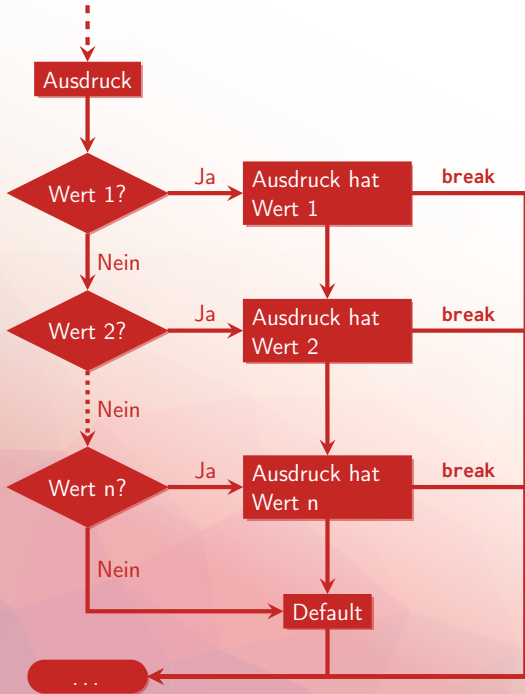
Darum switch-case! II

```
62     case Calendar.FEBRUARY:
63         if (isLeapYear){
64             System.out.println("29 Tage");
65         } else {
66             System.out.println("28 Tage");
67         }
68         break;
69     default:
70         System.out.println("Ungültiger Monat");
71     }
72 }
73 }
```


SwitchCase.java

- ▶ **switch-case** für die bedingte Ausführung über mehrere Möglichkeiten
- ▶ Wert des **Ausdruck** wird der Reihe nach mit den Vergleichswerten (Wert1...n) verglichen
- ▶ **Treffer**: Fall wird ausgeführt
- ▶ Zulässige Typen:
 - ▶ **byte, char, short, int**
 - ▶  **String**
 - ▶ Enumerationen
- ▶ **Vergleichswerte** müssen **konstante Ausdrücke** vom gleichen Typ sein
- ▶ Mehrere Vergleichswerte können zum **selben Fall** gehören

```
switch (Ausdruck){  
    case Wert1:  
        /* ... */  
        break;  
    case Wert2:  
    case Wert3:  
        /* ... */  
    case Wert4:  
        /* ... */  
    default:  
        /* ... */  
}
```

switch-case: Beispiel

```
78  runSwitchCaseExample
79 switch (n % 5){
80     case 0:
81         System.out.println("Rest 0");
82         break;
84     case 1:
85     case 2:
86         System.out.println("Rest 1 oder 2");
87         break;
89     case 4:
90         System.out.println("Rest 4");
92     default:
93         System.out.println("Default");
94 }
```

 SwitchCase.java

n == 25
Rest 0


n == 31
Rest 1 oder 2

n == 32
Rest 1 oder 2

n == 48
Default

n == 99
Rest 4
Default

switch-case: Beispiel — unter der Haube


```
78  runSwitchCaseExample
79 switch (n % 5){
80     case 0:
81         System.out.println("Rest 0");
82         break;
83
84     case 1:
85     case 2:
86         System.out.println("Rest 1 oder 2");
87         break;
88
89     case 4:
90         System.out.println("Rest 4");
91
92     default:
93         System.out.println("Default");
94 }
```

 SwitchCase.java

```
1: iload n
2: iconst 5
3: irem
4: tableswitch {
    0: 5
    1: 7
    2: 7
    3: 10
    4: 9
    default: 10
}
5: p("Rest 0")
6: goto 11 // break
7: p("Rest 1 oder 2")
8: goto 11 // break
9: p("Rest 4")
10: p("Default")
11: return
```

switch-case: ↗ String s |

Als Vergleichswerte sind auch ↗ String s möglich:

```
104  runSwitchCaseStringExample
105 switch (userInput.toUpperCase()){
106     case "JA":
107     case "YES":
108         System.out.println("Nutzer sagt 'Ja'!");
109         break;
111     case "NEIN":
112     case "NO":
113         System.out.println("Nutzer sagt 'Nein'!");
114         break;
116     case "VIELLEICHT":
117     case "MAYBE":
118         System.out.println("Nutzer ist sich nicht sicher!");
119         break;
121     default:
```

```
]switch-case: ↗ String s
```

```
122     System.out.println("Eingabe nicht verstanden: " + userInput);  
123 }
```

[SwitchCase.java](#)

switch-case: ↗ String s

- Es sind allerdings nur **konstante** ↗ String s als Vergleichswerte erlaubt

```
String yes = "YES";  
switch (userInput.toUpperCase()){  
    case yes:    // FEHLER  
        /* ... */  
}
```

Fehler: „case expression must be a constant expression“

```
final String yes = "YES";  
switch (userInput.toUpperCase()){  
    case yes:    // kein Fehler  
        /* ... */  
}
```

switch-case: Konstante Ausdrücke

- ▶ Allgemein sind als Vergleichswerte nur **konstante Ausdrücke** erlaubt
 - ▶ Compiler **berechnet** die Ausdrücke **vor**
 - ▶ Während Laufzeit sind die Vergleichswerte **Konstanten**
- ▶ Ausdrücke sind **konstant**, wenn
 - ▶ sie nur aus **Literalen** zusammengesetzt sind
 - ▶ alle verwendeten Bezeichner **final** sind
- ▶ Beispiele:

```
final int konstante = 10;  
int variable = 5;  
konstante * 10 / 5      // konstanter Ausdruck  
variable * konstante + 5 // kein konstanter Ausdruck  
Math.random() * 10     // kein konstanter Ausdruck
```

- ▶ Der zu **vergleichende Wert** kann ein beliebiger Ausdruck sein

switch-case: Konstante Ausdrücke

```
130 final int theAnswer = 42;
131 switch ((int) (Math.random()*100)) {
132     case theAnswer:
133         System.out.println("Die ganze Wahrheit");
134         break;
136     case theAnswer/2:
137         System.out.println("Die halbe Wahrheit");
138         break;
140     case theAnswer*2:
141         System.out.println("Die doppelte Wahrheit");
142         break;
144     default:
145         System.out.println("Was anderes");
146 }
```

SwitchCase.java

switch-case: Blick in die Zukunft

- ▶ In **Preview** in Java 13 (javac/jsshell -enable-preview)
- ▶ Comma-Separated Labels

```
boolean confirmed;  
switch (input.toUpperCase()){  
    case "JA", "YES", "OUI":  
        confirmed = true;  
        break;  
    case "NEIN", "NO", "NON", default:  
        confirmed = false;  
        break;  
}
```

- ▶ Switch Labeled Rules: kein **break** mehr

```
boolean confirmed;  
switch (input.toUpperCase()){  
    case "JA", "YES", "OUI" -> confirmed = true;  
    case "NEIN", "NO", "NON", default -> confirmed = false;  
}
```

switch-case: Blick in die Zukunft

► Switch Expression

```
boolean confirmed =  
switch (input.toUpperCase()){  
    case "JA", "YES", "OUI" -> true;  
    case "NEIN", "NO", "NON", default -> false;  
}
```

- `switch` selbst ist ein Ausdruck
- Idee kommt aus **Pattern Matching** der funktionalen Programmierung

while- und for-Schleifen und Schleifen-Kontrollfluss

- while- und do-while-Schleifen

- „Klassische“ for-Schleife

- for-each-Schleife

- Fehlerquelle Abbruchbedingung

- Geschachtelte Schleifen

- Schleifen-Marken

Inhalt

while- und for-Schleifen und Schleifen-Kontrollfluss

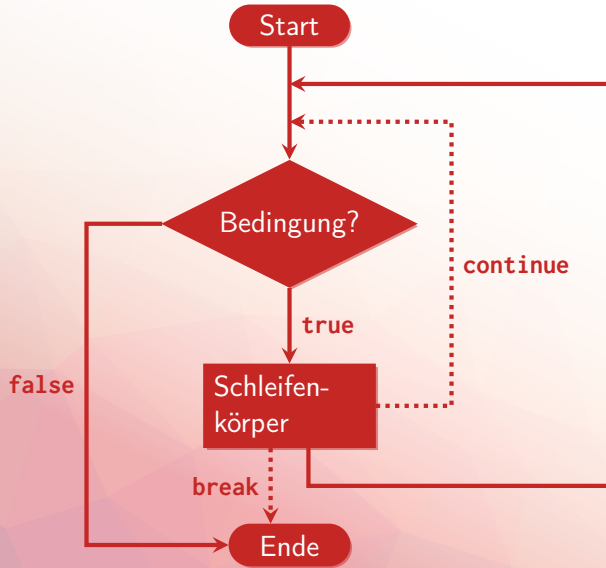
while- und do-while-Schleifen

while-Schleife

```
while (Bedingung)  
    Schleifenkörper
```


- ▶ **Bedingung**: boolescher Ausdruck
- ▶ **Schleifenkörper**: zu wiederholende Anweisung (meist **Block**)
- ▶ Funktionsweise:
 - ▶ **Einstieg** nur wenn Bedingung erfüllt ist
 - ▶ **Wiederholung** solange bis Bedingung nicht mehr erfüllt ist
- ▶ Änderung des **Schleifen-Kontrollflusses**
 - ▶ **break** verlässt die Schleife
 - ▶ **continue** springt zur Prüfung der **Schleifenbedingung**
 - ▶ (**return** verlässt Methode — und damit Schleife)

while-Schleife: Flussdiagramm



while-Schleife: Beispiel I

findContainingString sucht nach dem Vorkommen von searchString in einer Aufzählung von Strings (stringsIterator)

```
10  runFindContainingString  
11 public static void findContainingString(  
12     Iterator<String> stringsIterator,  
13     String searchString) {  
14     String match = null;  
15     while (stringsIterator.hasNext()){  
16         String candidate = stringsIterator.next();  
17         // zu kurze Strings sofort verwerfen  
18         if (candidate.length() < searchString.length()){  
19             System.out.printf("\'%s\' ist zu kurz.%n", candidate);  
20             continue;  
21         }  
22         if (candidate.contains(searchString)){  
23             match = candidate;  
24             break;  
25         }  
26     }
```

while-Schleife: Beispiel II

```
27     }else{
28         System.out.printf("Kein Treffer: \"%s\"%n", candidate);
29     }
30 }
32 if (match != null){
33     System.out.printf("Treffer: \"%s\"%n", match);
34 }else{
35     System.out.printf("Leider nichts gefunden.%n");
36 }
37 }
```

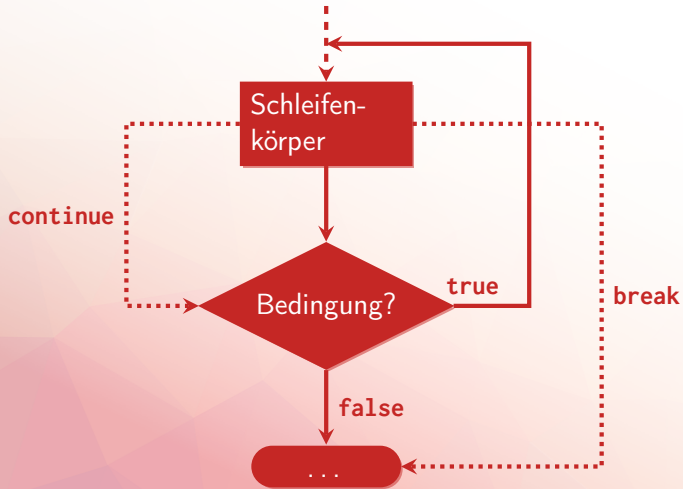
While.java

do-while-Schleife


```
do  
    Schleifenkörper  
while (Bedingung);
```

- ▶ Unterschied zu **while**-Schleife
 - ▶ Prüfung der Bedingung am **Ende**
 - ▶ Der Schleifenkörper wird **mindestens einmal** durchlaufen
- ▶ Änderung des **Schleifen-Kontrollflusses**
 - ▶ **break** verlässt die Schleife
 - ▶ **continue** springt zur Prüfung der **Schleifenbedingung am Ende**
 - ▶ (**return** verlässt Methode — und damit Schleife)

do-while-Schleife: Flussdiagramm



do-while-Schleife: Beispiel I

```
44  runDoWhileExample
45 boolean validInput = false;
46 boolean confirmed = false;
47
48 do{
49     System.out.println("Sind die einverstanden?");
50
51     String answer = scanner.nextLine();
52
53     switch (answer.toUpperCase()){
54         case "YES": case "JA": case "OUI":
55             confirmed = true;
56             validInput = true;
57             break;
58
59         case "NO": case "NEIN": case "NON":
60             confirmed = false;
61             validInput = true;
62             break;
63
64         default:
```

do-while-Schleife: Beispiel II

```
66     System.out.println("Ich verstehe Sie nicht.");  
67 }  
69 } while (!validInput);  
71 System.out.printf("Einverstanden: %b%n", confirmed);
```

While.java

while und do-while mit Stil

► Schlechter Stil:

```
while (true){  
    /* ... */  
    if (Abbruchbedingung)  
        break;  
    /* ... */  
}
```

► Abbruch im Schleifenkörper:

- Abbruchbedingung **nicht** sofort ersichtlich
- undurchsichtiger **Kontrollfluss**

► Alternative:

```
boolean done = false; // besser: sprechender Name  
while (!done){  
    /* ... */  
    if (Abbruchbedingung)  
        done = true;  
    /* ... */  
}
```

Inhalt

while- und for-Schleifen und Schleifen-Kontrollfluss
„Klassische“ for-Schleife

„Klassische“ for-Schleife (Grundversion)

```
for (Initialisierung; Bedingung; Fortsetzung)
    Schleifenkörper
```

- ▶ **Initialisierung:** Variablendeklaration mit Initialisierung

```
for (int i = 0; ...; ...)
```

- ▶ **Bedingung:** boolescher Ausdruck

```
for (... ; i < n; ...)
```

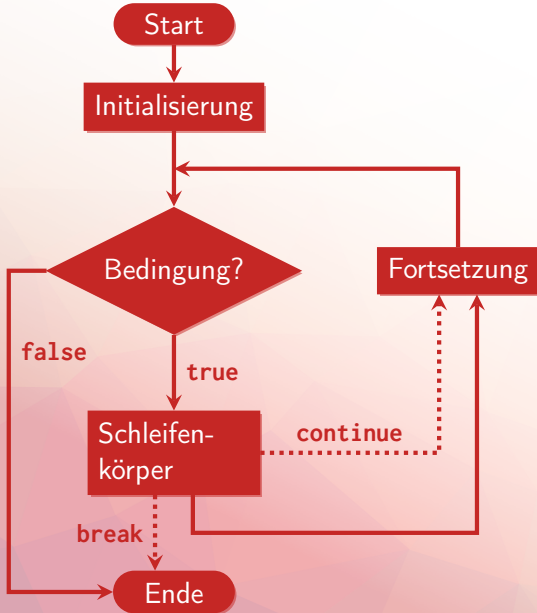
- ▶ **Fortsetzung:** Ausdrucksanweisung (s. Folie 34)

```
for (...; ...; i++)
```

- ▶ Änderung des Schleifen-Kontrollflusses

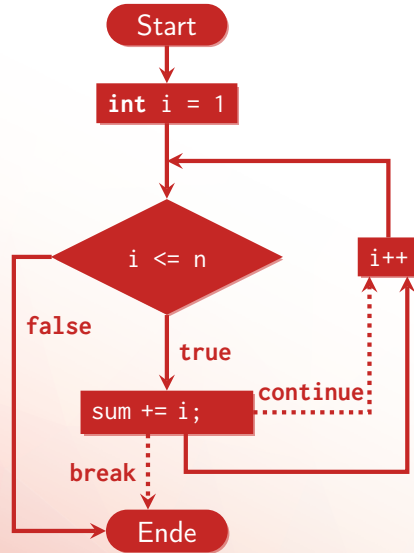
- ▶ **break** verlässt die Schleife
- ▶ **continue** springt zur Fortsetzung der Schleifen am Anfang
- ▶ **(return** verlässt Methode — und damit Schleife)

„Klassische“ for-Schleife: Flussdiagramm



„Klassische“ for-Schleife: Flussdiagramm (Beispiel)

```
int sum = 0
for (int i = 1; i <= n; i++){
    sum += i;
}
```



„Klassische“ for-Schleife: Die ganze Wahrheit

```
for (Initialisierung; Bedingung; Fortsetzung)
```

► Initialisierung

► Variablendeklarationen

```
for (int i = 0, j = 9; ...; ...) { ... }
```

► Oder: Ausdrucksanweisungen durch Kommas getrennt

```
for (logInit(), d1 = 0.0; ...; ...) { ... }
```

► Oder: leer

```
for ( ; i < 10 && j > 0; ...) { ... }
```

► Bedingung: boolescher Ausdruck oder leer (**true**)

► Fortsetzung:


► Ausdrucksanweisungen durch Kommas getrennt

```
for (...; ...; i++, j--) { ... }  
for (...; ...; logStep(), d = next(d)) { ... }
```

► Oder: leer

„Klassische“ for-Schleife: Beispiele

► Endlosschleife


```
10  runForInfiniteLoopExample  
11 for ( ; ; ){  
12     System.out.println("All work and no play makes Jack a dull boy");  
13 }
```

 For.java

```
All work and no play makes Jack a dull boy  
All work and no play makes Jack a dull boy  
All work and no play makes Jack a dull boy  
...
```

„Klassische“ for-Schleife: Beispiele

► Multiplikationstabelle [Insel]


```
20  runForMultiplicationTable  
21 for (int i = 1, j = 9; i < 10; i++, j--){  
22     System.out.printf("%d * %d = %d%n", i, j, i*j);  
23 }
```

 For.java

```
1 * 9 = 9  
2 * 8 = 16  
3 * 7 = 21  
4 * 6 = 24  
5 * 5 = 25  
6 * 4 = 24  
7 * 3 = 21  
8 * 2 = 16  
9 * 1 = 9
```

„Klassische“ for-Schleife: Beispiele I

► Ausdrucksanweisungen in for-Schleife

```
29  runForExpressionStatementsExample  
30 public static void forExpressionStatementsExample() {  
31     int i, sum;  
33     for (i = 0, sum = 0, logInit(i, sum); // Initialisierung  
34         i < 100; // Bedingung  
35         i++, logStep(i, sum)) { // Fortsetzung  
36         sum += i;  
37     }  
39 }  
41 private static void logInit(int i, int sum){  
42     System.out.printf(  
43         "Initialisierung: i == %d, sum == %d%n", i, sum);  
44 }  
46 private static void logStep(int i, int sum){  
47     System.out.printf(  
48         "Fortsetzung: i == %d, sum == %d%n", i, sum);
```

„Klassische“ for-Schleife: Beispiele II

49 | }

For.java

Initialisierung: $i == 0$, $sum == 0$

Fortsetzung: $i == 1$, $sum == 0$

Fortsetzung: $i == 2$, $sum == 1$

Fortsetzung: $i == 3$, $sum == 3$

...

Fortsetzung: $i == 99$, $sum == 4851$

Fortsetzung: $i == 100$, $sum == 4950$

Frage: Warum ist die letzte Fortsetzung bei $i==100$ obwohl die Bedingung doch $i<100$ verlangt?

„Klassische“ for-Schleife: Beispiele

- ▶ **Achtung bei Initialisierung:** Entweder Ausdrucksanweisung **oder** Variablendeklaration

```
for (int i = 0, logInit(i); ...; ... ) // FEHLER
```

- ▶ **KISS-Prinzip:** „keep it stupid simple“

- ▶ Unübersichtlich und fehleranfällig

```
for (i = 0, sum = 0 ,logInit(i, sum); i < 100; i++, logStep(i, sum))
```

- ▶ **Alternative:** länger aber verständlicher

```
int sum = 0;  
logInit(i, sum);  
for (int i = 0; i < 100; i++){  
    sum += i;  
    logStep(i, sum);  
}  
logStep(i, sum);
```

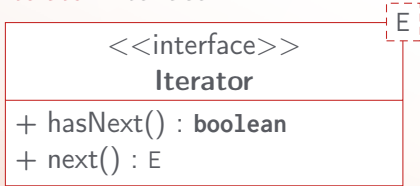
Warum ist das letzte logStep nötig für die gleiche Ausgabe?

Inhalt

while- und for-Schleifen und Schleifen-Kontrollfluss
for-each-Schleife

for-each-Schleife: Das Iterator-Interface

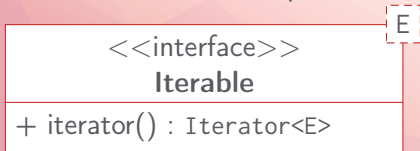
▶ Iterator-Interface



▶ Ermöglicht **schrittweises Durchlaufen** von Elemente („iterieren“)

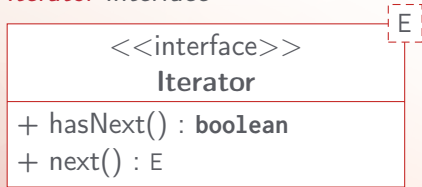
- ▶ **Flache Datenstrukturen**: Listen, Arrays, allg. **Collections**
- ▶ **nicht-flache Datenstrukturen**: Bäume, Graphen
- ▶ **Allgemein**: aufzählbare Objekte

▶ „Iterierbare“ Klassen implementieren das `Iterable`-Interface



for-each-Schleife: Das Iterator-Interface

Iterator-Interface



- ▶ `hasNext()`: liefert **true** wenn Iterator noch ein Element zur Aufzählung hat, sonst **false**
- ▶ `next`:
 - ▶ liefert das **nächste Element**
 - ▶ bewegt Iterator-Position um eins weiter


for-each-Schleife: Iterator-Interface Beispiel

🔗 `LinkedList` implementiert das 🔗 `Iterable`-Interface

```
53 public static LinkedList<CelestialBody> planets() {  
54     LinkedList<CelestialBody> planets =  
55         new LinkedList<CelestialBody>();  
56     planets.add(new CelestialBody("Mercury", 0.330e24));  
57     planets.add(new CelestialBody("Venus", 4.87e24));  
58     planets.add(new CelestialBody("Earth", 5.97e24));  
59     planets.add(new CelestialBody("Moon", 0.073e24));  
60     planets.add(new CelestialBody("Mars", 0.642e24));  
61     planets.add(new CelestialBody("Jupiter", 1898e24));  
62     planets.add(new CelestialBody("Saturn", 568e24));  
63     planets.add(new CelestialBody("Uranus", 86.8e24));  
64     planets.add(new CelestialBody("Neptune", 102e24));  
65     planets.add(new CelestialBody("Pluto", 0.0146e24));  
66     return planets;  
67 }
```

📄 For.java

for-each-Schleife: Iterator-Interface Beispiel

```
72  runIteratorExample  
73 LinkedList<CelestialBody> planets = planets();  
75 // iterator erstellen (Iterable-Interface)  
76 Iterator<CelestialBody> planetsIterator = planets.iterator();  
77 double massSum = 0d;  
79 // solange noch Elemente aufzulisten sind  
80 while (planetsIterator.hasNext()){  
81     // hole nächstes Element  
82     CelestialBody planet = planetsIterator.next();  
83     massSum += planet.getMass();  
84 }  
86 System.out.printf("Masse aller Planeten: %e\n", massSum);
```

 For.java

for-each-Schleife


```
Iterator<ElementType> iterator = elements.iterator();  
while (iterator.hasNext()){  
    ElementType element = iterator.next();  
    /* ... */  
}
```

- ▶ „Boilerplate Code“: kommt sehr häufig vor
- ▶ Enter **for**-each-Loop

```
for (ElementType element : elements){  
    /* ... */  
}
```

- ▶ Änderung des **Schleifen-Kontrollflusses** (wie gehabt)
 - ▶ **break** verlässt die Schleife
 - ▶ **continue** springt zur Prüfung der **Schleifenbedingung** (hasNext)
 - ▶ (**return** verlässt Methode — und damit Schleife)

for-each-Schleife: Beispiel

```
92  runForEachExample  
93 LinkedList<CelestialBody> planets = planets();  
94 double massSum = 0d;  
96 for (CelestialBody planet : planets){  
97     massSum += planet.getMass();  
98 }  
100 System.out.printf("Masse aller Planeten: %e%n", massSum);
```

 For.java

for-each-Schleife: Unter der Haube

```
Iterator<Typ> iterator =  
    elements.iterator();  
while (iterator.hasNext()){  
    Typ element =  
        iterator.next();  
}
```


```
0: aload elements  
1: invoke LinkedList.iterator()  
2: astore iterator  
3: goto 7  
4: aload iterator  
5: invokeinterface Iterator.next()  
6: astore element  
7: aload iterator  
8: invoke hasNext()  
9: ifne 4 // springt wenn true
```

```
for (Typ element : elements){  
    /* ... */  
}
```

```
0: aload elements  
1: invoke LinkedList.iterator()  
2: astore iterator  
3: goto 7  
4: aload iterator  
5: invokeinterface Iterator.next()  
6: astore element  
7: aload iterator  
8: invoke hasNext()  
9: ifne 4 // springt wenn true
```

for-each-Schleife: Beispiel (Array)

for-each funktioniert auch auf **Arrays**

```
106  runForEachArrayExample  
107 int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
108 int sum = 0;  
110 for (int number: numbers){  
111     sum += number;  
112 }  
114 System.out.printf("Summe: %d\n", sum);
```

 For.java

for-each-Schleife: Unter der Haube (Array)

```
for (int number : numbers){  
    /* ... */  
}
```

- ▶ Arrays implementieren das [Iterable](#)-Interface nicht
- ▶ Compiler übersetzt **for**-each in „klassische“ **for**-Schleife:


```
for (int i = 0; i < numbers.length; i++){  
    /* ... */  
}
```

- ▶ Freiwillige Übung: Bytecode vergleichen

Inhalt

while- und for-Schleifen und Schleifen-Kontrollfluss
Fehlerquelle Abbruchbedingung

Fehlerquelle Abbruchbedingung


```
101  runBadLoopExample
102 int lower = scanner.nextInt();
103 scanner.nextLine();
104 int upper = scanner.nextInt();
105
106 for (int i = lower; i != (upper+1); i++){
107     System.out.printf("%d^2 = %d\n", i, i*i);
108 }
```

 Loops.java

```
1
4
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
```

```
5
1
5^2 = 25
6^2 = 36
7^2 = 49
...
256^2 = 65536
...
```

Fehlerquelle Abbruchbedingung — Verbesserung

```
115  runImprovedLoopExample
116 int lower = scanner.nextInt();
117 scanner.nextLine();
118 int upper = scanner.nextInt();
120 for (int i = lower; i <= upper; i++){
121     System.out.printf("%d^2 = %d%n", i, i*i);
122 }
```

 Loops.java

▶ Allgemeine Konvention bei Intervallen

- ▶ Untere Schranke **einschließen**
- ▶ Obere Schranke **ausschließen**

▶ In Beispiel von oben

```
for (int i = lower; i < (upper+1); i++)
```

▶ Bei Arrays


```
for (int i = 0; i < array.length; i++)
```

Inhalt

while- und for-Schleifen und Schleifen-Kontrollfluss

Geschachtelte Schleifen

Geschachtelte Schleifen: Beispiel


```
9   runNestedLoopsExample  
10 for (int p = 2; p < 10; p++) {  
11     for (int q = p; q < 10; q++) {  
12         System.out.printf("%d * %d = %d\n", p, q, p*q);  
13     }  
14 }
```

 Loops.java

```
2 * 2 = 4  
2 * 3 = 6  
2 * 4 = 8  
...  
8 * 9 = 72  
9 * 9 = 81
```

Geschachtelte Schleifen: Noch ein Beispiel

Bubble Sort zum Sortieren

```
25  runBubbleSort  
26 public static void bubbleSort(int[] numbers) {  
27     int n = numbers.length;  
28     for (int i = 0; i < n-1; i++) {  
29         for (int j = 0; j < n-i-1; j++) {  
30             if (numbers[j] > numbers[j+1]) {  
31                 swap(numbers, j, j+1);  
32             }  
33         }  
34     }  
35 }
```


 Loops.java

Eingabe: [5, 1, 3, 4, 2, 6, 7, 9, 8]

Ergebnis: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Geschachtelte Schleifen


- ▶ **Schachtelungstiefe** von mehr als zwei vermeiden
 - ▶ Verständlichkeit
 - ▶ Nicht sofort ersichtliches Verhalten bei **break** und **continue** (siehe nächste Folien)
 - ▶ Performance: Müssen die Schleifen geschachtelt sein?
- ▶ Alternativen
 - ▶ Auslagern in Methoden
 - ▶ Redundante Berechnungen vor die Schleifen ziehen
- ▶ Beispiel: (Quadrieren einer quadratischen Matrix)

```
55  runSquareMatrix
56 for (int i = 0; i < n; i++){
57     for (int j = 0; j < n; j++){
58         result[i][j] = 0;
59         for (int k = 0; k < n; k++){
60             result[i][j] += matrix[i][k] * matrix[k][j];
61         }
62     }
63 }
```

 Loops.java

Geschachtelte Schleife

- Auslagern der innersten Schleife in Methode

```
86  runImprovedSquareMatrix  
87 for (int i = 0; i < n; i++){  
88     for (int j = 0; j < n; j++){  
89         result[i][j] = innerProduct(matrix, i, j);  
90     }  
91 }
```

 Loops.java

- Inneres Produkt von Zeilen- und Spaltenvektor der Matrix:

```
70 public static int innerProduct(int[][] x, int i, int j){  
71     int result = 0;  
72     for (int k = 0; k < x.length; k++){  
73         result += x[i][k] * x[k][j];  
74     }  
75     return result;  
76 }
```


 Loops.java

Inhalt

while- und for-Schleifen und Schleifen-Kontrollfluss
Schleifen-Marken

Schleifen-Marken

Frage: Welche Ausgabe macht folgendes Programm?

```
128  runSimpleBreakExample
129 for (int i = 0; i < 3; i++) {
130     for (int j = 0; j < 3; j++) {
131         System.out.printf("i = %d, j = %d\n", i, j);
132         break;
133     }
134 }
```


 Loops.java

```
i = 0, j = 0
i = 1, j = 0
i = 2, j = 0
```

- ▶ Grund: **break** bricht **nur innere** Schleife ab
- ▶ Aber: Was ist wenn man **beide Schleifen** abbrechen will?
- ▶ Und: Das gleiche Problem ergibt sich auch mit **continue**

Schleifen-Marken I

Findet heraus ob String s den String searchString beinhaltet

```
140  runBreakLoopExample
141 String s = "I used to be an adventurer like you, then I took an arrow in the knee";
142 String searchString = "arrow";
143 boolean found = false;
144
145 // teste jede Position für searchString in s
146 for (int i = 0; i < s.length()-searchString.length(); i++){
147     int j = 0;
148     found = false;
149
150     // vergleiche Zeichen für Zeichen
151     while (searchString.charAt(j) == s.charAt(i+j)){
152         j++;
153
154         // alle Zeichen von searchString stimmen überein
155         if (j >= searchString.length()){
156             found = true;
157             break;
```

Schleifen-Marken II

```
158     }  
159 }  
160 }  
161 System.out.printf("Gefunden: %b%n", found);
```

Loops.java

Gefunden: false

- ▶ **Problem:** **break** verlässt die innere Schleife
- ▶ **Aber** **break** muss **beide** Schleifen verlassen

Schleifen-Marken

`schleifenMarke:`
`Schleife`

- ▶ **schleifenMarke**: Bezeichner, der Schleife identifiziert
- ▶ **Schleife**: **while**-, **do-while** oder **for**-Schleife
- ▶ **continue** und **break** mit Marken in der Schleife:
 - ▶ **break** `schleifenMarke`; bricht Ausführung Schleife mit Marke „`schleifenMarke`“ ab
 - ▶ **continue** `schleifenMarke`; springt zu Schleifenbedingung von Schleife mit Marke „`schleifenMarke`“

Schleifen-Marken: Beispiel I

```
outerLoop:
while ( ... ) {
  innerLoop:
  for ( ... ) {
    // bricht beide Schleifen ab
    break outerLoop;

    // springt zu Bedingung von äußerer Schleife
    continue outerLoop;

    // äquivalent zu break/continue ohne Marke (nur innere Schleife)
    break innerLoop;
    continue innerLoop;
  }

  secondInnerLoop:
  do {
    // FEHLER: nur für aktive Schleifen erlaubt
    break innerLoop;
```


Schleifen-Marken: Beispiel II

```
// FEHLER: nur für aktive Schleifen erlaubt  
continue innerLoop;  
} while ( ... )  
}
```

- ▶ **break** oder **continue** mit Marken sind nur für **aktive Schleifen** erlaubt

Schleifen-Marken I

Korrektur: „break“ wurde durch „break searchLoop“ ersetzt

```
168  runBreakLoopWithLabelExample
169 String s = "I used to be an adventurer like you, then I took an arrow in the knee";
170 String searchString = "arrow";
171 boolean found = false;
172
173 searchLoop: // NEU: Marke für äußere Schleife
174 for (int i = 0; i < s.length()-searchString.length(); i++){
175     int j = 0;
176     found = false;
177
178     while (searchString.charAt(j) == s.charAt(i+j)){
179         j++;
180
181         if (j >= searchString.length()){
182             found = true;
183             break searchLoop; // NEU: bricht beide Schleifen ab
184         }
185     }
186 }
```

Schleifen-Marken II

```
187 }  
188 System.out.printf("Gefunden: %b%n", found);
```

Loops.java

Korrektes Ergebnis:

Gefunden: true

Methoden, Signaturen, Rekursion

Sichtbarkeit

Modifizierer

Rückgabewerte

Parameter

varargs

Überladen von Methoden

Anwendung von Überladung: Default-Parameterwerte

Call-by-Value in Java

Mehrere Resultate

main-Methode

Beispiel für Methoden einer Klasse

Methodenaufrufe

Rekursion

Methoden einer Klasse

- ▶ Methoden existieren im **Kontext einer Klasse**
- ▶ Methoden...
 - ▶ implementieren das Verhalten der **Instanzen (Objekte)** von Klassen (**Instanzmethoden**)
 - ▶ implementieren Instanz-unabhängige Funktionalität (**statische Methoden**)
 - ▶ dienen zur **Modularisierung** von Programmcode (Auslagerung von wiederkehrenden Programmteilen in Methoden)
- ▶ **Bestandteile** einer Methode

```
public double getMass() {  
    return this.mass;  
}
```

- ▶ `public double getMass():` **Signatur**
- ▶ `{ return this.mass }`: **Methodenrumpf**

Signatur einer Methode (Grundversion)

```
public static void main(String[] args)
```

Sichtbarkeit*	Modifizierer*	Rückgabotyp†	Bezeichner	Parameter
public	static	void	main	(String[] ↵ args)
private	final	Primitiv		()
protected	abstract	Referenz		(int ... xs)
	synchronized			
	strictfp			
	(native)			

* Optional

† Leer für Konstruktor

Inhalt

Methoden, Signaturen, Rekursion Sichtbarkeit

Sichtbarkeit

```
public class Sichtbarkeit{  
    public void jederDarf();  
    private void nurDieseKlasse();  
    protected void fuerAbleitungen();  
    void nurImPaket();  
}
```

Sichtbarkeit

```
+ jederDarf(): void  
# fuerAbleitungen(): void  
- nurDieseKlasse(): void  
~ nurImPaket(): void
```

Schlüsselwort	UML	Sichtbarkeit	Verwendung
public	+	Jeder	öffentliche Schnittstelle
private	–	Klasse	Hilfsmethoden
protected	#	Hierarchie	Schnittstelle zu Basisklassen
	~	Paket	interne Schnittstelle für Paket

Sichtbarkeit

```
public class Sichtbarkeit{  
    public void jederDarf();  
    private void nurDieseKlasse();  
    protected void fuerAbleitungen();  
    void nurImPaket();  
}
```

- ▶ **Sichtbarkeit** definiert einen „Vertrag“ für die Verwendung
 - ▶ Auf welche Bestandteile darf **zugegriffen** werden?
 - ▶ Welche Bestandteile sind **nur intern** relevant?
- ▶ Sichtbarkeit ist **kein** Mittel um Code vor unerlaubten Zugriffen zu schützen („security“)
- ▶ **private**, **protected** und Paket-sichtbare Methoden können über **Reflection** aufgerufen werden

Inhalt

Methoden, Signaturen, Rekursion
Modifizierer



Modifizierer

Schlüsselwort	UML	Bedeutung
static	<u>unterstrichen</u>	Klassenmethode (statisch)
abstract *	<i>kursiv</i>	ohne Implementierung
final *		nicht überschreibbar
synchronized [†]		Zugriff unter gegenseitigem Ausschluss
strictfp [†]		plattformunabh. Gleitkommaoperationen
native [†]		native Implementierung (in C/C++)

* wird später näher behandelt; † in diesem Kurs nicht näher behandelt

- ▶ Modifizierer können **miteinander kombiniert** werden

```
public static final synchronized doSomething() { /* ... */ }
```

- ▶ Nicht alle Kombinationen sind **erlaubt**

```
public abstract final doSomething() { /* ... */ }
```

Inhalt

Methoden, Signaturen, Rekursion Rückgabewerte

Rückgabewerte

► Primitiver Typ

```
public double getMass() {  
    return this.mass;  
}
```

► Referenztyp

```
public CelestialBody getPluto(){  
    return new CelestialBody("Pluto", 1.31e22);  
}
```

► **void** für Methoden ohne Rückgabewert

```
public void printCelestialBody(CelestialBody body){  
    System.out.println("%s (%e)%n",  
        body.getName(), body.getMass());  
    return;  
}
```

Rückgabewerte

- ▶ **return** **bricht** die Methodenausführung **ab**
- ▶ Bei Rückgabe „wert“ **void** ist **return** **optional**
- ▶ Ist ein **Rückgabewert** **definiert**...
 - ▶ So **muss jeder Ausführungspfad** einen Wert **zurückgeben**
 - ▶ Wir der Rückgabewert mit dem Schlüsselwort **return** zurückgegeben
- ▶ **Fehlerhaftes** Beispiel

```
public int sign(int x){  
    if (x < 0)  
        return -1;  
    else if (x > 0)  
        return +1;  
}
```

- ▶ **Fehler:** „Method must return int“
- ▶ Fall $x == 0$ **fehlt**

Rückgabewerte

Noch ein Versuch:

```
public int sign(int x){  
    if (x < 0)  
        return -1;  
    else if (x > 0)  
        return +1;  
    else if (x == 0)  
        return 0;  
}
```

- ▶ **Wieder Fehler:** „Method must return int“
- ▶ Compiler kann nicht „wissen“...
 - ▶ dass es eine **vollständige Fallunterscheidung** ist
 - ▶ der Code unterhalb des letzten Falls **nie erreicht** wird

Rückgabewerte

Korrekte Version(en):

```
public int sign(int x){  
    if (x < 0)  
        return -1;  
    else if (x > 0)  
        return +1;  
    else // x == 0  
        return 0;  
}
```

```
public int sign(int x){  
    if (x < 0)  
        return -1;  
    else if (x > 0)  
        return +1;  
    return 0;  
}
```

```
public int sign(int x){  
    int sign = 0;  
    if (x < 0)  
        sign = -1;  
    else if (x > 0)  
        sign = +1;  
    return sign;  
}
```

← **Sauberste** Version, weil...

- ▶ **Ein** return am Ende
- ▶ **Kontrollfluss** wird nicht durch Rücksprung unterbrochen

Inhalt

Methoden, Signaturen, Rekursion Parameter

Parameter

- ▶ Keine Parameter

```
public void println()
```

- ▶ Durch Komma getrennte Auflistung von Parametern

```
public void println(String s)
public String substring(int beginIndex, int endIndex)
// javax.sql.RowSet:
public void setDate(String name, Date x, Calendar cal)
```

- ▶ Auflistung von Parametern mit **varargs** am Ende

```
public int sum(int... xs)
public void printf(String format, Object... args);
```

Inhalt

Methoden, Signaturen, Rekursion varargs



varargs

```
void example(Typ1 arg1, Typ2 arg2, Typ3... args3)
```

- ▶ varargs werden durch ... nach dem Typ gekennzeichnet
- ▶ Einschränkungen
 - ▶ nur ein varargs erlaubt

```
void example(int... numbers, int... more) // FEHLER
```

- ▶ varargs müssen am Ende stehen


```
void example(int... numbers, int i) // FEHLER
```

- ▶ varargs werden auf Arrays abgebildet

varargs: Beispiel

```
30 public static int max(int... numbers) {  
31     int maxValue = Integer.MIN_VALUE;  
32     for (int number : numbers){  
33         maxValue = (number > maxValue ? number : maxValue);  
34     }  
35     return maxValue;  
36 }
```

Methods.java

```
41  runVarargsExample2  
42 System.out.printf("max() = %d\n", max());  
43 System.out.printf("max(0) = %d\n", max(0));  
44 System.out.printf("max(5,1,8,10) = %d\n", max(5,1,8,10));
```

Methods.java


```
max() = -2147483648  
max(0) = 0  
max(5,1,8,10) = 10
```

varargs sind wirklich Arrays

```
8 public static void varargsIntrospection(int... numbers) {  
9     System.out.println("Type: " +  
10         numbers.getClass().getSimpleName());  
11     System.out.println("Length: " + numbers.length);  
13     for (int number : numbers)  
14         System.out.print(number + " ");  
15     System.out.println();  
16 }
```

Methods.java

Hinweis: varargs können auch direkt als **Arrays** übergeben werden

```
21  runVarargsIntrospectionExample  
22 int[] numberArray = new int[] {1,2,3,4,5};  
23 varargsIntrospection();  
24 varargsIntrospection(1,2,3);  
25 varargsIntrospection(numberArray);
```

Methods.java

varargs sind wirklich Arrays

Ausgabe des vorherigen Beispiels

```
Type: int[]
```

```
Length: 0
```

```
Type: int[]
```

```
Length: 3
```

```
1 2 3
```

```
Type: int[]
```

```
Length: 5
```

```
1 2 3 4 5
```

Inhalt

Methoden, Signaturen, Rekursion Überladen von Methoden

Überladen von Methoden

```
public void println()  
public void println(String x)  
public void println(double x)  
...
```

- ▶ Überladene Methoden haben...
 - ▶ gleichen Namen
 - ▶ aber unterschiedliche Parameter
- ▶ Unterschiedliche Rückgabewerte reichen nicht

```
public int add(int i, int j) {}  
public long add(int i, int j) {}
```

Fehler: Duplicate method

- ▶ Compiler entscheidet zur Übersetzungszeit welche Methode aufgerufen wird
- ▶ Aber nach welchen Regeln?

Überladen von Methoden: Beispiel

```
86 public static void overload(String s) {  
87     System.out.println("overload(String)");  
88 }
```

Methods.java

```
98 public static void overload(String s1, String s2) {
```

Methods.java

```
92 public static void overload(int i) {
```

Methods.java

```
104 public static void overload(String s1, int i) {
```

Methods.java

```
110 public static void overload(int i, String s) {
```

Methods.java

Überladen von Methoden: Regeln

- ▶ Der Compiler entscheidet welche Methode aufgerufen wird...

- ▶ nach der **Anzahl** der Parameter

```
overload("Hello"); // overload(String)
overload("Hello", "World"); // overload(String, String)
```


- ▶ nach dem **Typ** des Parameters

```
overload("Hello"); // overload(String)
overload(123); // overload(int)
```

- ▶ nach der **Reihenfolge** der Parameter

```
overload("Hello", 123); // overload(String, int)
overload(123, "Hello"); // overload(int, String)
```

- ▶ Siehe auch

```
117  runSimpleOverloadExample
118 overload("Hello");
119 overload("Hello", "World");
120 // ...
```

 Methods.java

Überladen von Methoden: Hierarchien

- ▶ Noch eine **Überladung**

```
128 public static void overload(Object obj) {  
129     System.out.print("overload(Object)");  
130 }
```

Methods.java

- ▶ **Hinweis:** alle Klassen leiten von [Object](#) ab
- ▶ **Welche** Methoden werden aufgerufen?

```
overload("Hello"); // overload(String)  
overload(  
    new CelestialBody("rock", 140)); // overload(Object)
```

- ▶ **Regel:** Es wird immer die **spezifischste, mögliche** Methode aufgerufen

Überladen von Methoden: Allgemeine Regel

- ▶ Seien `m1` und `m2` zwei Methoden mit...
 - ▶ gleichem Bezeichner
 - ▶ unterschiedlichen Parametern
- ▶ Dann ist `m1` **spezifischer** als `m2` wenn man einen Parametersatz, der für `m1` möglich ist,
 - ▶ ohne **Veränderung** (insbesondere Cast)
 - ▶ und ohne **Compiler-Fehler**für `m2` verwenden kann
- ▶ **Beispiele:**
 - ▶ `overload(String s)` ist spezifischer als `overload(Object s)`
 - ▶ `overload(int i)` ist spezifischer als `overload(long l)`
 - ▶ `overload(int i, int j)` ist spezifischer als `overload(int... is)`

Überladen von Methoden: Beispiel

Reihenfolge entspricht „ist spezifischer“-Relation:

1. `overload()`
2. `overload(int i, int j)`
3. `overload(int... is)`
4. `overload(long i, long j)`

Beispiele

- ▶ `overload(1) → overload(int...)`
- ▶ `overload(1,2) → overload(int,int)`
- ▶ `overload(1,2,3) → overload(int...)`
- ▶ `overload(1L) → Fehler`, kein gültiger Aufruf vorhanden
- ▶ `overload(1L,2L) → overload(long, long)`
- ▶ `overload(1L,2) → overload(long, long)`

Hinweis: der zweite Parameter wird zu `long` promotet

- ▶ `overload() → overload()`

Hinweis: es wird **nicht** `overload(int...)` aufgerufen, da `overload()` **spezifischer** ist

Inhalt

Methoden, Signaturen, Rekursion

Anwendung von Überladung: Default-Parameterwerte

Default-Werte von Methoden

- ▶ Java unterstützt **keine** Default-Parameter
- ▶ Beispiel aus C#

```
public void greeting(  
    string greeting = "Hello",  
    string target = "World"){ ... }
```

```
greeting() // Hello World!  
greeting(greeting: "Servus") // Servus World!  
greeting(target: "Landshut") // Hello Landshut!  
greeting("Servus", "Landshut") // Servus Landshut!
```

- ▶ Wie kann man das in **Java** abbilden?


Default-Parameter mit Hilfe von Überladung

Durch **Überladung** können Default-Parameter abgebildet werden

```
220 public static void greeting(String greeting, String target) {  
221     System.out.printf("%s %s!\n", greeting, target);  
222 }  
  
224 public static void greeting(String greeting) {  
225     greeting(greeting, "World");  
226 }  
  
228 public static void greeting() {  
229     greeting("Hello");  
230 }  
  
232 // der "Trick" hat seine Grenzen...  
233 public static void greetingWithTarget(String target) {  
234     greeting("Hello", target);  
235 }
```

Methods.java

Default-Parameter mit Hilfe von Überladung

```
240  runDefaultParameterExample  
241 greeting();  
242 greeting("Servus");  
243 greetingWithTarget("Landshut");  
244 greeting("Servus", "Landshut");
```

 Methods.java

```
Hello World!  
Servus World!  
Hello Landshut!  
Servus Landshut!
```

- ▶ Der Compiler kann **nicht** zwischen `greeting(String greeting)` und `greeting(String target)` **unterscheiden**
- ▶ Daher muss die Methode **`greetingWithTarget`** implementiert werden
- ▶ **Hinweis:** Dieses „Pattern“ funktioniert auch bei **Konstruktoren**

Inhalt

Methoden, Signaturen, Rekursion Call-by-Value in Java

Call-by-Value

- ▶ Java unterstützt nur **call-by-value**
- ▶ Vergleich zu C/C++ **call-by-reference**


```
void swap(int* x, int* y){  
    int temp;  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

- ▶ Es gibt keinen derartigen ***-Operator** in Java
- ▶ Auch **Referenzen** werden als **call-by-value** übergeben
 - ▶ **Instanzen** von Objekten
 - ▶ **Arrays**

Call-by-Value: Beispiel

```
50 public static void replaceByPlanet(CelestialBody body) {  
51     body = new CelestialBody("Pluto", 1.31e22);  
52 }
```

Methods.java

```
58  runCallByValueExample  
59 var body = new CelestialBody("some rock", 140);  
60 System.out.printf("%s (%e kg)%n",  
61     body.getName(), body.getMass());  
62 replaceByPlanet(body);  
63 System.out.printf("%s (%e kg)%n",  
64     body.getName(), body.getMass());
```

Methods.java


```
some rock (1,400000e+02 kg)  
some rock (1,400000e+02 kg)
```

Call-by-Value: Noch ein Beispiel

Das **referenzierte Objekt** kann aber durch Methodenaufrufe **verändert** werden

```
69 public static void addRandomInt(LinkedList<Integer> xs) {  
70     xs.add((int) (Math.random()*100));  
71 }
```

Methods.java

```
76  runCallByValueExample2  
77 var numbers = new LinkedList<Integer>();  
78 numbers.add(1);  
79 System.out.println(numbers);  
80 addRandomInt(numbers);  
81 System.out.println(numbers);
```

Methods.java

```
[1]  
[1, 30]
```

Inhalt

Methoden, Signaturen, Rekursion Mehrere Resultate

Wie gibt man mehrere Resultate zurück?

- ▶ Java...
 - ▶ unterstützt **kein** call-by-reference
 - ▶ unterstützt **kein** mehreren Rückgabewerte
- ▶ Wie kann man **mehrere Resultate** zurückgeben?
- ▶ **Unschöne** Lösung:

```
187 public int[] minAndMax(int... numbers) {  
188     int minValue = min(numbers);  
189     int maxValue = max(numbers);  
191     return new int[] {minValue, maxValue};  
192 }
```

Methods.java

- ▶ **Ähnlich unschön**: Über Collection-Klassen (Listen, Hash-Tabellen, etc.)
- ▶ Wenn überhaupt, dann nur für **private Methoden**

Mehrere Resultate in Java I

Die Lösung in Java:

- **Klasse** für Resultat erstellen

```
205 public class MinMaxResult{
206     private final int min;
207     private final int max;
209     public MinMaxResult(int min, int max){
210         this.min = min;
211         this.max = max;
212     }
214     public int getMax() { return max; }
215     public int getMin() { return min; }
216 }
```

Methods.java

- Als **Rückgabewert** Instanz der Klasse erstellen

Mehrere Resultate in Java II

```
196 public MinMaxResult minAndMax2(int... numbers) {  
197     int minValue = min(numbers);  
198     int maxValue = max(numbers);  
200     return new MinMaxResult(minValue, maxValue);  
201 }
```

Methods.java

Inhalt

Methoden, Signaturen, Rekursion

main-Methode

Alle Methoden sind gleich — und main ist gleicher

- ▶ main-Methode: **Einstiegspunkt** in das Programm

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

- ▶ Signatur muss **genauso aussehen**
- ▶ (args kann prinzipiell anders heißen)
- ▶ args beinhaltet **Kommandozeilen-Parameter**
- ▶ Klasse in der main deklariert ist heißt **main-Klasse**
- ▶ Beim **Aufruf** über Konsole mit java

```
java MainKlasse arg1 arg2 ...
```

Ausführbare jar-Datei erstellen

- ▶ Erstellen von **ausführbarer jar-Datei** für main-Klasse „de.hawla.FancyProgram“

- ▶ **Manifest**-Datei FancyProgram.mf erstellen

```
Manifest-Version: 1.0  
Main-Class: de.hawla.FancyProgram
```

- ▶ jar-Datei **erstellen**

```
jar cmf FancyProgram.mf \  
    FancyProgram.jar <.class-Dateien>
```

- ▶ jar-Datei **ausführen**:

```
java -jar FancyProgram.jar
```

- ▶ Oder: IDE/Build-Tool nutzen...

Inhalt

Methoden, Signaturen, Rekursion

Beispiel für Methoden einer Klasse

Beispiel: Die Klasse Rectangle

Die Klasse **Rectangle** modelliert Rechtecke

Rectangle

- width : **double**
- height : **double**
- area : **double**

- + Rectangle(width : **double**, height : **double**)
- + setWidth(width : **double**): **void**
- + getWidth(): **double**
- + setHeight(height : **double**): **void**
- + getHeight(): **double**
- + area(): **double**
- + isSquare(**double** error): **boolean**
- + canContain(Rectangle other): **boolean**
- + scale(**double** s): **void**
- # updateArea(): **void**
- approxEqual(**double** x, **double** y, **double** error): **boolean**
- + getEnclosing(rectangles : ...) : Rectangle

Rectangle: Konstruktor

- ▶ Initialisiert und erstellt das Objekt
- ▶ Hat keinen Rückgabewert
- ▶ Beispiel: initialisiert Länge und Breite des Rechtecks

```
12 public Rectangle(final double width, final double height) {  
13     this.width = width;  
14     this.height = height;  
15     updateArea();  
16 }
```

Rectangle.java

Rectangle: Getter/Setter I

- ▶ Einfacher lesender und (eventuell) schreibender Zugriff auf Attribute
- ▶ **Beispiel:** lesender und schreibender Zugriff auf die Länge und Breite

```
40 public double getHeight() {  
41     return height;  
42 }  
44 public void setHeight(final double height) {  
45     if (height <= 0)  
46         throw new IllegalArgumentException("height must positive");  
47     this.height = height;  
48     updateArea();  
49 }  
51 public double getWidth() {  
52     return width;  
53 }  
55 public void setWidth(final double width) {  
56     if (width <= 0)
```


Rectangle: Getter/Setter II

```
57     throw new IllegalArgumentException("width must positive");  
58     this.width = width;  
59     updateArea();  
60 }
```

Rectangle.java

Rectangle: Abfragemethoden I

- ▶ Liefern Informationen zum Objekt
- ▶ Beispiel:
 - ▶ area liefert die Fläche
 - ▶ canContain prüft ob das Rechteck ein anderes beinhalten kann
 - ▶ isSquare prüft ob das Rechteck (annähernd) quadratisch ist

```
77 public double area(){
78     return area;
79 }
81 public boolean canContain(Rectangle other){
82     if (other == null)
83         throw new IllegalArgumentException("other rectangle must not be null");
84     return other.getWidth() < width && other.getHeight() < height;
85 }
87 public boolean isSquare(double error){
88     return approxEqual(width, height, error);
89 }
```

Rectangle: Abfragemethoden II

Rectangle.java

Rectangle: Modifizierende Methoden I

- ▶ Verändern den Zustand des Objekts
- ▶ **Beispiel:** skaliert das Rechteck um einen Faktor

```
93 public void scale(double s){  
94     if (s <= 0)  
95         throw new IllegalArgumentException("scale factor must be positive");  
96     width *= s;  
97     height *= s;  
98     updateArea();  
99 }
```

Rectangle.java

Rectangle: Hilfsmethoden I

- ▶ Zur Auslagerung von sich wiederholendem Code und Nebenrechnungen
- ▶ **private**
- ▶ **Beispiel:** prüft ob zwei **double**-Werte annähernd gleich sind

```
70 private boolean approxEqual(double x, double y, double error){  
71     return Math.abs(x-y) <= Math.abs(error);  
72 }
```

Rectangle.java

Rectangle: protected-Methoden I

- ▶ Methoden, die von ableitenden Klassen aufgerufen werden können sollen
- ▶ **Beispiel:** aktualisiert die Fläche des Rechtecks nach der Änderungen von Werten

```
64 protected void updateArea(){  
65     area = width * height;  
66 }
```

Rectangle.java

Rectangle: Klassenmethoden I

- ▶ Auch **statische Methoden** genannt
- ▶ Modifizierer **static**
- ▶ Werden der **Klasse** und nicht einer Instanz zugeordnet
- ▶ Können **ohne eine Instanz** der Klasse aufgerufen werden
- ▶ **Beispiel**: Factory-Methode, erstellt neues Rechteck, das die übergebenen Rechtecke umschließt

```
20 public static Rectangle getEnclosing(Rectangle... rectangles){
21     if (rectangles.length == 0)
22         throw new IllegalArgumentException("at least one rectangle must be given");
24     double maxWidth = Double.NEGATIVE_INFINITY;
25     double maxHeight = Double.NEGATIVE_INFINITY;
27     for (Rectangle rectangle : rectangles) {
28         if (rectangle.getWidth() > maxWidth)
29             maxWidth = rectangle.getWidth();
```

Rectangle: Klassenmethoden II

```
31     if (rectangle.getHeight() > maxHeight)
32         maxHeight = rectangle.getHeight();
33     }
34
35     return new Rectangle(maxWidth, maxHeight);
36 }
```

Rectangle.java

Inhalt

Methoden, Signaturen, Rekursion Methodenaufrufe

Methodenaufrufe

► Form:

```
methodenName(argumente);  
// oder:  
referenz.methodenName(argumente);
```

► ()-Operator:

1. **Berechnung** der Parameter (Parameterwerte liegen auf **Aufruf-Stack**)
2. **Unterbrechung Kontrollfluss** der aktuellen Methode
3. **Ausführung** des Methodenrumpfes
4. Eventuell **Rückgabewerte** auf Stack legen
5. Wiederaufnahme Kontrollfluss von **Aufrufer**
6. **Ergebnis** von Stack verwenden (oder verwerfen)

Methodenaufruf: Unter der Haube

```
public static int add(int a,  
    int b) {  
    int result = a + b;  
    return result;  
}
```

MethodCalls.java

- ▶ Zeilen 0–2: Parameterwerte addieren
- ▶ Zeilen 3: Speichern des Ergebnisses in result
- ▶ Zeile 4: Wert von result auf Stack legen
- ▶ Zeile 5: Rücksprung

```
// int result = a+b;  
0: iload a  
1: iload b  
2: iadd  
3: istore result  
  
// return result;  
4: iload result  
5: ireturn
```

Methodenaufruf: Unter der Haube

runMethodCallExample

```
int i = 2, j = 5;  
add(2*i, j*j);
```

 MethodCalls.java

- ▶ Zeilen 5 und 6: $2*i$ auf Stack legen
- ▶ Zeilen 7–9: $j*j$ auf Stack legen
- ▶ Zeile 10: Methodenaufruf, Stack:
 - ▶ Oben: $j*j$
 - ▶ Darunter: $2*i$
- ▶ Zeile 11: Rückgabewert liegt auf Stack und wird mit pop verworfen

```
// int i = 2, j = 5;  
0: iconst 2 // 2 laden  
1: istore i // in i speichern  
2: iconst 5 // 5 laden  
3: istore j // in j speichern  
  
// add(2*i, j*j);  
4: iconst 2 // 2 laden  
5: iload i // i laden  
6: imul // multiplizieren  
7: iload j // j laden  
8: iload j // j laden  
9: imul // multiplizieren  
10: invoke add // Aufruf  
11: pop // Rückgabewert werfen
```


Auswertungsreihenfolge von Parametern

- ▶ Parameter werden von **links nach rechts** ausgewertet

- ▶ Beispiel:

```
22 public static int id(int i) {  
23     System.out.printf("id(%d)%n", i);  
24     return i;  
25 }
```

MethodCalls.java

```
30  runParameterEvaluationExample  
31 System.out.printf("%d, %d oder %d%n", id(1), id(2), id(3));
```

MethodCalls.java

- ▶ Ergebnis:


```
id(1)  
id(2)  
id(3)  
1, 2 oder 3
```

Aneinanderhängen von Methodenaufrufen

- ▶ Methodenaufrufe können **aneinander gehängt** werden, wenn der Rückgabewert eine Referenz ist:

```
referenz.methode1().methode2();
```

- ▶ Beispiel:

```
37  runMethodCallChain  
38 String s1 = "It's Mario-Time!";  
39 String s2 =  
40     s1.substring(0,10).replace(" ", "-a me, ").concat("!");  
41 System.out.printf("s1 = %s\ns2 = %s\n", s1, s2);
```

 MethodCalls.java

- ▶ Ergebnis:

```
s1 = It's Mario-Time!  
s2 = It's-a me, Mario!
```

- ▶ Aber zur **Übersichtlichkeit** Zwischenwerte verwenden

Inhalt

Methoden, Signaturen, Rekursion Rekursion



Rekursion


- ▶ Rekursive Methoden **rufen sich selbst** auf
- ▶ Rekursion kann verwendet werden für...
 - ▶ **Algorithmische Probleme**, z.B. Divide & Conquer-Verfahren wie Merge-Sort, vollständige kombinatorische Aufzählung, etc. (siehe Vorlesung „Algorithmen und Datenstrukturen“)
 - ▶ **Manche Design-Pattern** aus der objektorientierten Programmierung, z.B. das Visitor-Pattern
 - ▶ **Berechnung mathematischer (rekursiver) Funktionen**
- ▶ **(Standard-)Beispiel**: Fibonacci-Folge $F : \mathbb{N}_0 \rightarrow \mathbb{N}$

$$F(n) = \begin{cases} 1, & \text{für } n \leq 1 \\ F(n-1) + F(n-2), & \text{sonst.} \end{cases}$$

Fibonacci-Folge

```
7 public static long fib(int n) {  
8     if (n <= 1)  
9         return 1;  
10    else  
11        return fib(n-1) + fib(n-2);  
12 }
```

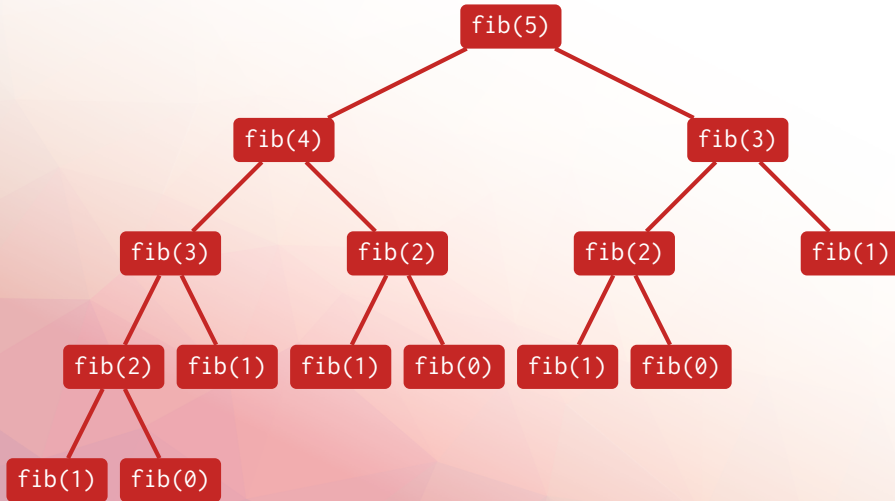
Recursion.java

```
17  runFibonacciExample  
18 System.out.printf("fib(0) = %d\n", fib(0)); // 1  
19 System.out.printf("fib(1) = %d\n", fib(1)); // 1  
20 System.out.printf("fib(2) = %d\n", fib(2)); // 2  
21 System.out.printf("fib(3) = %d\n", fib(3)); // 3  
22 System.out.printf("fib(10) = %d\n", fib(10)); // 89  
23 System.out.printf("fib(30) = %d\n", fib(30)); // 1346269  
24 System.out.printf("fib(45) = %d\n", fib(45)); // ...
```

Recursion.java

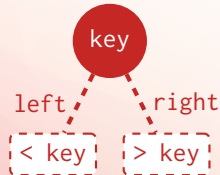
Fibonacci-Folge: Rekursionsbaum

Ein rekursiver Aufruf lässt sich als **Rekursionsbaum** darstellen



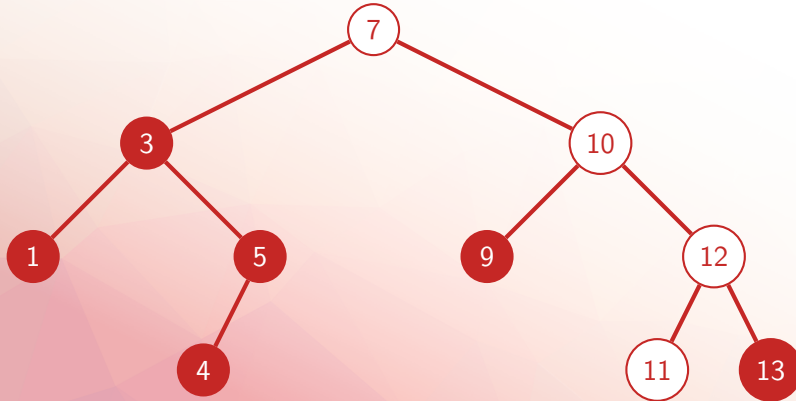
Ein praktischeres Beispiel: Binärer Suchbaum

- ▶ Binäre Suchbäume zur **sortierten Speicherung von Schlüsseln**
- ▶ **Knoten-Klasse** `BinaryNode` für einen binären Suchbaum:
 - ▶ **Schlüssel:** `int key`
 - ▶ **Linkes Kind:** `BinaryNode left` (kann `null` sein)
 - ▶ **Rechtes Kind:** `BinaryNode right` (kann `null` sein)
- ▶ Schlüssel im **linken Teilbaum** sind $< \text{key}$
- ▶ Schlüssel im **rechten Teilbaum** sind $> \text{key}$



Ein praktischeres Beispiel: Binärer Suchbaum

Auffinden des Wertes 11



Die Klasse `BinaryNode`

4 `public class BinaryNode`

`BinaryNode.java`

► Felder

```
8 private final int key;  
9 private BinaryNode left;  
10 private BinaryNode right;
```

`BinaryNode.java`

► Konstruktor

```
14 public BinaryNode(int key) {  
15     this.key = key;  
16 }
```

`BinaryNode.java`

Die Klasse `BinaryNode`

- ▶ Die Methode `BinaryNode find(int searchKey)`
 - ▶ `searchKey == this.key` ⇒ Knoten gefunden
 - ▶ `searchKey < this.key` ⇒ suche im linken Teilbaum weiter
 - ▶ `searchKey > this.key` ⇒ suche im rechten Teilbaum weiter


```
37 public BinaryNode find(int searchKey){
38     if (key == searchKey)
39         return this; // gefunden!
41     if (searchKey < key && left != null)
42         return left.find(searchKey); // rekursiver Aufruf
43     else if (searchKey > key && right != null)
44         return right.find(searchKey); // rekursiver Aufruf
45     else
46         return null;
47 }
```

BinaryNode.java

StackOverflow ist nicht nur eine Internet-Plattform

► Aufpassen bei der Rekursionstiefe:

- Gibt die **Rekursionstiefe** aus und macht einen rekursiven Aufruf:

```
29  runStackOverflowExample  
30 public static void recursion(int depth){  
31     System.out.printf("Tiefe %d\n", depth);  
32     if (depth < 100000)  
33         recursion(depth+1);  
34 }
```

 Recursion.java

► Fehler beim Ausführen:

```
1  
2  
...  
9715  
StackOverflowError
```

StackOverflows vermeiden

- ▶ Parameter werden bei Methodenaufrufen auf einen **Stack** gelegt (Stapelspeicher)
- ▶ Stack hat eine **begrenzte Kapazität**
- ▶ Wird diese überschritten: [↗ StackOverflowError](#)
- ▶ Lösungsansätze
 - ▶ **Stackgröße erhöhen** mit `java -Xss1M` (oder mehr)
 - ▶ Rekursion **auflösen**

```
52 public BinaryNode find2(int searchKey){  
53     BinaryNode currentNode = this;  
54     while (currentNode != null  
55         && currentNode.key != searchKey){  
56         currentNode =  
57             searchKey < currentNode.key ?  
58             currentNode.left : currentNode.right;  
59     }  
60     return currentNode;  
61 }  
62 }
```

BinaryNode.java