Programmieren in Java 2.Jahrgang

Dr. Heinz Schiffermüller

HTBLA-Kaindorf
Abteilung EDVO

Erstellung: Aug 2018
Letzte Überarbeitung: Sep 2018

Inhaltsverzeichnis

1 Einführung 85
1.1 Inhalte POS 2.Jahrgang - Programmieren mit Java 85
1.2 Software zur Programmentwicklung mit Java 87
1.3 Sprachmerkmale von Java 87
2 Grundzüge der Objekt-Orientierung 88
2.1 Klassen und Objekte 88
2.1.1 Attribute und Methoden 88
2.1.2 Die `main()` Methode 89
2.1.3 Die Elemente einer Klasse 90
2.1.4 Kommentare 91
2.1.5 Konstruktoren 91
2.2 Variablen 93
2.2.1 Scope von Variablen 93
2.2.2 Identifier 94
2.3 Methoden 95
2.3.1 Parameter-Übergabe 95
2.3.2 Getter-und setter-Methoden 96
2.3.3 Methoden überladen 97
2.3.4 Methoden mit variablen Argumenten: Var-Args 98
2.4 Klassenattribute und Klassenmethoden 99
2.4.1 Statischen Variablen 99
2.4.2 Statische Methoden 101
2.5 Sichtbarkeit und Datenkapselung 101
2.6 Initialisierungsblöcke 102
3 Datentypen, Operatoren und Anweisungen 104
3.1 Datentypen 104
3.2 Operatoren 106
3.2.1 Operatoren für numerische Datentypen 106
3.2.2 Vergleichsoperatoren 106
3.2.3 Logische Operatoren 106
3.2.4 Bit-Shift-Operatoren 107
3.2.5 Der Fragezeichen-Operator 107
3.3 Arrays 107
4 Exception-Handling 109
4.1 Das Abfangen von Exceptions mit `try-catch`-Blöcken 109
4.2 Das Weiterleiten von Exceptions mit `throws` 110
4.3 Das Erzeugen neuer Exceptions mit `throw` 110
4.4 Die `finally`-Anweisung 111
4.5 Wichtige Java-Exceptions 113
5 Wichtige Klassen der Java-API 114
5.1 Die Klasse `Scanner` 114
5.1.1 Methoden der Klasse `Scanner` 114
5.2 Die Klasse `Math` 115
5.2.1 Methoden und Variablen der Klasse `Math` 115
5.3 Die Klasse `Random` 115
5.3.1 Methoden der Klasse `Random` 115
5.4 Die Klasse `String` 116
5.4.1 Wichtige Methoden der Klasse `String` 116
5.4.2 Verkettung von `Strings` 118
5.4.3 Verkettung von `String`-Methoden 119
5.5 Die Klassen `StringBuffer` und `StringBuilder` 120
5.5.1 Wichtige Methoden der Klassen `StringBuffer` und `StringBuilder` 120
5.6 Die Klasse `Object` 122
5.6.1 Die Methode `toString()` 122
5.6.2 Die Methode `equals()` 123
5.7 Klassen für Datum und Uhrzeit 125
5.7.1 Datum und Uhrzeit 125
5.7.2 Ändern von Datum und Uhrzeit 126
5.7.3 Die Klasse `Period` 127
5.7.4 Die Klasse `DateTimeFormatter` 129
5.8 Die Klasse `Arrays` 131
Quellenverzeichnis 133

1 Einführung

1.1 Inhalte POS 2.Jahrgang - Programmieren mit Java

3.Semester:

Grundzüge der Objekt-orientierung	kann für Objekte des realen Lebens Zustand und Verhalten benennen und für Objektorientierung abstrahieren	kann reale Objekte objektorientiert implementieren	
	kann die Unterschiede zwischen einer Klasse u. einem Objekt benennen, kann Merkmale von Methoden u. Attributen aufzählen		
	kann eine Klasse mit Konstruktor, Attributen und Methoden implementieren	kann Konstruktoren überladen	kann Initialisierungsblöcke gezielt einsetzen
	kann Instanzen anlegen		
	kann Methoden mit versch. Parametern aufrufen, kann Methoden überladen	kann Methoden mit variablen Argumenten implementieren	
	kann Klassenattribute und Klassenmethoden implementieren und anwenden	kann Klassenattribute einsetzen und gezielt Klassenmethoden definieren	
	kennt den Begriff Datenkapselung und kann die Vorzüge erklären	kann die Bedeutung v. Sichtbarkeiten gezielt einsetzen	
Verarbeitung von Kommandozeilen-parameter	kennt Kommandozeilenparameter und kann Kommandozeilenparameter auslesen		
einfache Fehlerbehandlung	Kann auf elementare Laufzeitfehler angemessen reagieren		
	kann durch Definieren von Testfällen logische Programmfehler erkennen und mit Hilfe des Debuggers lokalisieren	kann auch Tests für Sonderfälle definieren	
reguläre Ausdrücke	kann Zeichenketten mit einfachen regulären Audrücken prüfen	kann mit Hilfe der Klassenbibliothek Zeichenketten überprüfen	
rekursive Funktionen	kann einfache Problemstellungen rekursiv lösen		
Verwendung ein-facher Container Datentypen	kann Objekte in Listen speichern		

4.Semester:

Objektdiagramme	kann einfache Objektdiagramme kontextorientiert darstellen	
einfache Klassendiagramme	kann Klassendiagramme inklusive Vererbung und Assoziationen interpretieren und implementieren	kann verschiedene Arten von Assoziationen situationsgerecht umsetzen
Klassen	kann Klassen und ihre Beziehungen für konkrete Problemstellungen entwerfen und in Programmcode umsetzen	kann Systeme mit lose gekoppelteten Klassen entwerfen
Schnittsstellen und Vererbung	erkennt Vererbungshierarchien unter Verwendung von abstrakten Klassen und Schnittstellen und kann diese implementieren	kennt Typverträglichkeiten in Vererbungshierarchien
Fehlerbehandlung mit Exceptions	kann Laufzeitfehler gesichert behandeln und entsprechend darauf reagieren	kennt die wesentlichen Exceptions der Klassenbibliothek und kann diese situationsbedingt einsetzen. Kann domainspezifische Exceptions definieren.
Verwendung von Klassenbibliotheken	kennt ausgewählte Klassen der Klassenbibliothek	kann mit Hilfe der Dokumentation selbstständig Klassen problemorientiert auswählen und anwenden
Einfache Benutzerschnittstellen	Erstellung einfacher grafischer Benutzerschnittstellen unter Verwendung elementarer Controlls	
Text- und Binärdateien, Zugriffsmöglichkeiten	textorientiere Dateien zeilenweise lesen und schreiben	kennt den Unterschied zwischen Text- und Binärdateien und kann Binärdateien kontextspezifisch lesen und schreiben

1.2 Software zur Programmentwicklung mit Java

Als Entwicklungsumgebung für die Java-Programmierung sind die Netbeans 8.2 sowie das JDK (Java Development Kit) 8 herunterzuladen und am eigenen Laptop zu installieren:

Version: JDK 8
Netbeans 8.2
Link: http://www.oracle.com/technetwork/java/javase/downloads/jdk-netbeans-jsp-142931.html
Datei: jdk-8u171-nb-8_2-windows-x64.exe (ca. 340 MB)

Zur Erstellung von UML-Diagrammen ist das Case-Tool Astah am eigenen Laptop zu installieren:

Version: Astah UML
Link: http://astah.net/student-license-request

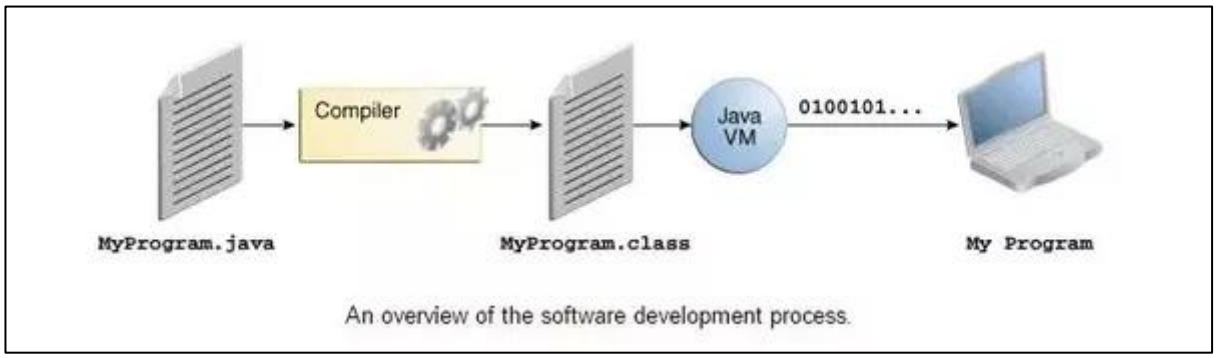
1.3 Sprachmerkmale von Java

Die Programmiersprache Java kann durch folgende Eigenschaften charakerisiert werden:

- ☐ Einfach
- ☐ Objektorientiert
- ☐ Unabhängig vom Betriebssystem
- ☐ Interpretiert
- ☐ Umfangreiche Klassenbibliothek

Entwicklung eines Java-Programms:

Der Java-Compiler übersetzt das Java-Programm aus der .java Datei in Bytecode und erzeugt eine .class Datei. Die .class-Datei ist unabhängig vom Betriebssystem (Windows, Linux, IOS,…) und kann mit Hilfer der JVM (Java Virtual Machine) interpretiert und ausgeführt werden:



An overview of the software development process.

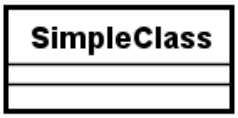
2 Grundzüge der Objekt-Orientierung

2.1 Klassen und Objekte

Klassen gehören zu den wichtigsten Elementen von objektorientierten Programmiersprachen. Eine Klassendeklaration wird durch das Schlüsselwort **`class`** eingeleitet. Die einfachste Klasse ist eine leere Klasse:

```
class SimpleClass
{
}
```

Zu jeder Klasse wird eine Datei mit dem gleiche Namen und der Extension **`.java`** erzeugt, in diesem Beispiel also die Datei **`SimpleClass.java`**. Der Compiler übersetzt die **`.java`** Quellcode Datei in eine **`.class`** Bytecode Datei.
Mit der UML (Unified Modelling Language) wird eine Klasse grafisch wie folgt dargestellt:



Objekte einer Klasse werden mit dem Schlüsselwort **`new`** erzeugt:

```
SimpleClass sc = new SimpleClass();
```

In diesem Beispiel ist **`sc`** ein Objekt der Klasse **`SimpleClass`**.

2.1.1 Attribute und Methoden

Die wichtigsten Elemente von Java-Klassen sind Attribute und Methoden.
Attribute werden auch als Instanzvariablen oder Membervariablen bezeichnet und sind Variablen die innerhalb einer Klasse definiert werden und gültig sind:

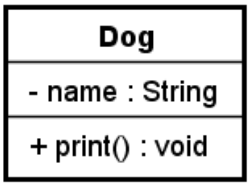
```
class Dog
{
  private String name = "Buddy";          // Instanzvariable
}
```

Methoden werdenn ebenfalls innerhalb einer Klasse definiert und enthalten Anweisungen:

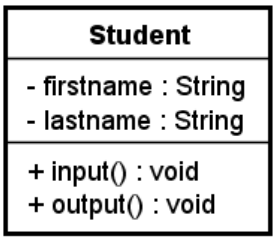
```
class Dog
{
  private String name = "Buddy";          // Instanzvariable

  public void print()                     // Methode
  {
     System.out.println("I am a dog and my name is: " + name);
  }
}
```

Instanzvariablen können in allen Methoden der Klasse verwendet werden. Die UML-Syntax für Instanzvariablen und Methoden sieht aus wie folgt:



Beispiel einer Klasse mit Attributen und Methoden:




```
public class Student {
    private String firstname;
    private String lastname;

    public void input()
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Please enter your firstname: ");
        firstname = scan.next();
        System.out.print("Please enter yout lastname: ");
        lastname = scan.next();
    }

    public void output()
    {
        System.out.println("My name is "+ firstname + " " + lastname);
    }

    public static void main(String[] args) {
        Student student1 = new Student();
        student1.input();
        student1.output();
        Student student2 = new Student();
        student2.input();
        student2.output();
    }
}
```

In der **`main()`**-Methode werden 2 Objekte der Klasse Student erzeugt und verwendet.

2.1.2 Die `main()` Methode

Jedes Java Programm beginnt seine Ausführung in einer **`main()`**-Methode. Jede Java-Klasse kann genau eine **`main()`**-Methode enthalten:

```
class SimpleClass
{
  public static void main(String [] args) {
    System.out.println("main()-method is executing");
  }
}
```

2.1.3 Die Elemente einer Klasse

Eine Klasse kann folgende Elemente enthalten:
- □ Package Deklarationen
- □ Import-Anweisungen
- □ Klassen-Deklaration
- □ Attribut-Deklaration
- □ Methoden-Deklaration
- □ Initialisierungsblöcke

Die Elemente einer Klasse müssen teilweise in einer bestimmten Reihenfolge angegeben werden:

Element	***Beispiel***	***erforderlich***	***Wo***
Package-Deklaration	`package abc;`	nein	Erste Zeile in der Datei
Import-Anweisungen	`import java.util.*;`	nein	nach package, vor class
Klassen-Deklaration	`public class C {}`	ja	nach dem import
Variablen	`int value;`	nein	innerhalb der Klasse
Methoden	`void method() {}`	nein	innerhalb der Klasse
Initialisierungsblöcke	`{}`	nein	innerhalb der Klasse

Package-Deklarationen ermöglichen es Java-Klassen nach logischen Gesichtspunkten zusammenzufassen. Die Java-API enthält tausende von Klassen die alle in packages strukturiert sind. Um eine Klasse aus der Java-API verwenden zu können muss das package, in dem sie definiert wurde, importiert werden:

```
public class ImportExample {
    public static void main(String[] args) {
        Random rand = new Random();     // Compilerfehler!
        System.out.println(rand.nextInt(10));
    }
}
```

Die Klasse Random() kann erst verwendet werden wenn das package richtig importiert wurde und der Compiler weiß wo die Klasse zu finden ist:

```
import java.util.Random;
public class ImportExample {
    public static void main(String[] args) {
        Random rand = new Random();
        System.out.println(rand.nextInt(10));  // Zahl zwischen 0 und 9
    }
}
```

Um alle Klassen eines packages zu importieren kann als Wildcard `*` verwendet werden:

```
import java.util.*;
public class ImportExample {
    public static void main(String[] args) {
        Random rand = new Random();
        System.out.println(rand.nextInt(10));  // Zahl zwischen 0 und 9
    }
}
```

Um eine Klasse einem package zuzuordnen wird ein eigenes Verzeichnis erstellt, bei Netbeans im Projects-Fenster unter Source Packages. Jede Klasse, die in diesem Verzeichnis erstellt wird enthält dann diesen Verzeichnisnamen als package-Namen:

```
package animals

import java.util.Random;

public class Dog {
    public static void main(String[] args) {
        Random rand = new Random();
        int age = rand.nextInt(10);
        System.out.println("My dog is" + age + "years old"));
    }
}
```

Die Reihenfolge der **`package`** – **`import`** – **`class`** – Anweisungen (**PIC**) darf nicht verändert werden.

Nach dem Start einer Java-Applikation läuft im Hintergrund automatisch ein Garbage Collector, der den Speicherplatz aller nicht mehr benötigten Objekte freigibt. Es ist nicht notwendig wie in C **`free()`** aufzurufen.

2.1.4 Kommentare

Folgende Kommentare sind in Java erlaubt:

```
// Kommentar bis zum Ende einer Zeile

/*
 Mehrzeiliger Kommentar
*/

/**
 * Javadoc - mehrzeiliger Kommentar
 */
```

2.1.5 Konstruktoren

Um ein Objekt (eine Instanz) einer Klasse zu erzeugen, wird der Konstruktor der Klasse aufgerufen:

```
Cat c = new Cat();
```

Bei der Deklaration eines Objekts wird zuerst der Name der Klasse (**`Cat`**) angegeben, dann der Name des Objekts (**`c`**). Mit **`new Cat()`** wird das Objekt erzeugt, wobeil **`Cat()`** der Konstruktor der Klasse **`Cat`** ist.

Konstruktoren werden innerhalb der Klasse definiert und sind vom Namen ident mit dem Namen der Klasse:

```
public class Cat {

    public Cat() {
        System.out.println("in constructor");
    }
}
```

Eine Klasse die keinen Konstruktor enthält bekommt einen sogenannten Default-Konstruktor, der keinen Übergabeparameter enthält, automatisch zugewiesen. Folgende Klassen sind daher identisch:

```
public class Bird {

    public Bird() {             // Konstruktor
    }
}
```

```
public class Bird {             // Klasse ohne Konstruktor – der Default-
                                // Konstruktor wird automatisch hinzugefügt
}
```

Eine Klasse kann auch mehrere Konstruktoren besitzen, man spricht dann vom Überladen des Konstruktors:

```
public class Bird {
    private String name;

    public Bird() {
        name = "Coco";
    }

    public Bird(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Bird b1 = new Bird();
        System.out.println(b1.name);
        Bird b2 = new Bird("Baxter");
        System.out.println(b2.name);
    }
}
```

Ausgabe:

```
Coco
Baxter
```

Konstruktoren werden oft zum Initialisieren der Instanzvariablen verwendet:

```
public class Dog {
    private String color;
    private int weight;

    public Dog(String color, int weight) {
        this.color = color;
        this.weight = weight;
    }
```

```
    public Dog(int weight) {
        this.weight = weight;
        color = "brown";
    }

    public static void main(String[] args) {
        Dog d1 = new Dog("white",12);
        Dog d2 = new Dog(6);
        Dog d3 = new Dog();          // Compilerfehler
    }
}
```

2.2 Variablen

In Java gibt es unterschiedliche Arten von Variablen, die nach ihrem Gültigkeitsbereich (Scope) unterschieden werden.

2.2.1 Scope von Variablen

Unter dem Scope einer Variable versteht man deren Gültigkeitsbereich. Folgende vier Scopes werden dabei unterschieden:

- **Lokale Variablen**
  Sie sind nur innerhalb einer Methode, ab ihrer Deklaration gültig. Sie erhalten keinen Defaultwert sondern müssen explizit initialisiert werden - sonst erfolgt ein Compiler-Fehler. Sie verlieren ihren Wert sobald die Methode verlassen wird.
- **Block-Variablen**:
  Für sie gelten die gleichen Regeln wie für lokale Variablen, ihr Gültigkeitsbereich beschränkt sich aber auf den Block `{…}` in dem sie definiert wurden. Laufvariablen in `for`-Schleifen sind ebenfalls nur innerhalb der Schleife gültig.
- **Instanzvariablen**
  Sie werden innerhalb der Klasse aber außerhalb von Methoden deklariert. Sie können in allen Methoden der Klasse verwendet werden. Sie erhalten erst mit der Instanzierung der Klasse einen Wert. Jede Instanzvariable hat eine Defaultwert. Ihre Lebensdauer ist an das dazugehörige Objekt gebunden.
- **Klassenvariablen**
  Sie werden innerhalb der Klasse aber außerhalb von Methoden mit dem Schlüsselwort `static` deklariert (sie werden deshalb auch statische Variablen genannt). Sind an die Klasse und nicht an die Objekte einer Klasse gebunden. Sie erhalten, gleich wie Instanzvariablen einen Defaultwert. Sie haben für alle Objekte der Klasse denselben Wert.

Beispiel:

```
public class VarScope
{
  private int instVar;          // Instanzvariable mit 0 initialisiert
  private static boolean classVar;  // Klassenvar. mit false initialisiert

  public void printToConsole() {
    int lokalVar;               // Lokale Variable ohne Initialisierung
```

```
    for (int i=0; i<10;i++)     // Block-Variable i
    {
      System.out.println(i);
    }
    System.out.println(i);      /* Compilerfehler! - Zugriff auf Block-
                                   Variable ausserhalb des Blocks:
                                   cannot find symbol: variable i  */
    System.out.println(lokalVar); /* Compilerfehler! - Zugriff auf nicht-
                                     Initialisierte lokale Variable:
                    variable lokalVar might not have been initialized  */
  }

  public static void main(String [] args) {
    VarScope scope = new VarScope();
    Scope.print();
  }
}
```

2.2.2 Identifier

Als Identifier bezeichnet man alle Namen die innerhalb eines Programms vergeben werden, d.h. die Namen von Variablen, Methoden, Klassen usw.
Folgende Regeln sind für die Namen von Identifiern zu beachten:

- Der Name beginnt mit einem Buchstaben oder den Symbolen **_** oder **$**
- Nach dem ersten Zeichen sind auch Ziffern erlaubt
- Java Schlüsselwörter sind nicht erlaubt

Java ist case-sensitve, d.h. unterschiedliche Groß-Kleinschreibung führt zu unterschiedlichen Identifieren.

Beispiele für gültige Identifier:

```
useCamelCase
ABC
x123
my_number
_a_valid_Name
__$$Value
```

Beispiele für ungültige Identifier, die einen Copilerfehler bewirken:

```
123X          // darf nicht mit Ziffer beginnen
not@home      // keine Sonderzeichen ausser _ und $
*coffee       // keine Sonderzeichen ausser _ und $
for           // keine Java Schlüsselwörter
char          // keine Java Schlüsselwörter
```

Coding Convention:
Alle Identifier sollten immer mit einem Kleinbuchstaben beginnen mit Ausnahme von Klassen und Interface-Namen

2.3 Methoden

Methoden enthalten die Funktionalitäten einer Klasse. Sie bestehen aus einer Folge von Anweisungen. Es gibt zwei verschiedene Arten von Methoden:

- ☐ Instanzmethoden
- ☐ Klassenmethoden

Instanzmethoden sind an ein Objekt der Klasse gebunden und arbeiten mit lokalen Variablen und Instanzvariablen. Instanzmethoden werden innerhalb der Klasse durch ihren Methodennamen aufgerufen, und ausserhalb der Klasse über die Objektreferenz in der Form:

```
objektname.methodenName()
```

Wenn eine Methode nicht auf Instanzvariablen zugreift bzw. keine anderen Instanzmethoden aufruft so sollte sie als Klassenmethode implementiert werden.

Klassenmethoden sind // ToDo

2.3.1 Parameter-Übergabe

Man unterscheidet zwischen Formal- und Aktualparameter. Die Liste der Formalparameter steht in der Methoden-Deklaration und enthält neben dem Parameternamen auch dessen Datentyp. Die Aktualparameter sind die Variablen oder Werte mit denen der Methodenaufruf erfolgt.
Die Aktualparameter müssen in Reihenfolge, Anzahl und Typ mit der Formalparameterliste übereinstimmen.

Die Übergabe von Parametern an Methoden erfolgt in Java immer durch Call by Value. D.h. der Parameter in der Methode ist eine Kopie des Aktualparameters. Eine Veränderung des Parameters in der Methode hat daher keinen Einfluss auf den Wert des Aktualparameters.
Das folgende Beispiel zeigt den Zusammenhang für primitive Datentypen:

```
public class Param1
{
  public static void main(String []args) {
    int wert = 100;             // lokal in main
    aendere(wert);
    System.out.println("Wert-Aktualparameter: " + wert);
  }

  public static void aendere(int n) {
    n++;
    System.out.println("Wert-Übergabeparameter: " + n);
  }
}
```

Ausgabe:

```
Wert-Übergabeparameter: 101
Wert-Aktualparameter: 100
```

Bei Referenztypen enthält der Parameter in der Methode eine Kopie der Referenz, d.h. wird der Wert einer Instanzvariable geändert, so wirkt sich die Änderung sowohl auf Parameter als auch auf den Aktualparameter aus, da beide Variablen auf die gleiche Referenz verweisen. Wird dem

Parameter allerdings eine neue Instanz zugeordnet, so bleibt das Aktualparameter-Objekt unverändert, wie das nachfolgende Beispiel zeigt:

```
public class Param2
{
  public static void main(String []args) {
    JButton bt= new JButton(); // JButtonreferenz erzeugen
    bt.setText("ROT");         // Beschriftung "ROT" setzen
    changeText(bt);
    System.out.println("Farbe: " + bt.getText());
  }

   public static void changeText(JButton ref) {
     ref.setText("GRUEN");      // ref und bt referenzieren dasselbe Objekt
     ref = new JButton();       // ref wird ein anderes Objekt zugewiesen
     ref. setText("BLAU");
  }
}
```

Ausgabe:

```
Farbe: GRUEN
```

2.3.2 Getter-und setter-Methoden

Da Instanzvariablen nur innerhalb der Klasse verwendet werden dürfen (Prinzip der Datenkapselung) gibt es verschiedene Möglichkeiten um den Wert einer Instanzvariable aus einer andern Klasse heraus zu verändern oder auf ihren Wert aus einer anderen Klasse zuzugreifen.
Zum Initialisieren von Instanzvariablen wird meist der Konstruktor verwendet. Kommen die gewünschten Werte für die Initialisierung aus einer anderen Klasse, können sie als Parameter an den Konstruktor übergeben werden und auf die Instanzvariable gespeichert werden. Bei Namesngleichheit von Instanzvariable und Übergabeparameter wird das Schlüsselwort **`this`** zur Unterscheidung der Variablen verwendet:

```
public class Student {
    private String firstname;
    private String lastname;

    public Student(String firstname, String lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }
}
```

```
public class UseStudent
{

    public void generateStudent()
    {
        Student student1 = new Student("Daniel","Lewis");
        Student student2 = new Student("Sarah","Parker");
    }
}
```

Getter-Methoden ermöglichen den Zugriff auf Instanzvariablen von einer anderen Klasse. Setter-Methoden ermöglichen es den Wert einer Instanzvariable aus einer anderen Klasse zu verändern.

Getter-Methoden haben denselben return-Typ wie der Datentyp der Instanzvariable. Entsprechend den Java Coding Conventions beginnt ihr Name mit **`get`** bzw. **`is`** bei boolean-Variablen. Die CamelCase-Schreibweise wird eingehalten. Der Method-Body besteht nur aus einer return-Anweisung, gibt also den Wert der Instanzvariable zurück.

Setter-Methoden sind **`void`** und haben einen Übergabeparameter, der dem Datentyp der Instanzvaribale entspricht. Entsprechend den Java Coding Conventions beginnt ihr Name mit **`set`**. Die CamelCase-Schreibweise wird eingehalten. In der Methode wird der Wert des Übergabeparameters auf die Instanzvariable geschrieben. Bei Namensgleichheit wird das Schlüsselwort **`this`** zur Unterscheidung der Variablen verwendet.

```
public class Student {
    private String name;

    public Student() {
    }

    public Student(String name) {
        this.name = name;
        this.lastname = lastname;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```
public void generateStudent()
{
   Student student1 = new Student();
   student1.setName("Daniel Lewis");
   System.out.println(student1.getName());

   Student student2 = new Student("Sarah Parker");
   System.out.println(student2.getName());
}
```

Ausgabe:

```
Daniel Lewis
Sarah Parker
```

2.3.3 Methoden überladen

Man spricht vom Überladen von Methoden wenn mehreren Methoden innerhalb einer Klasse den gleichen Namen haben, sich aber eindeutig durch ihre Übergabeparameter unterscheiden. Die Methoden müssen sich dabei zumindest durch eines der folgenden Merkmale unterscheiden:

- ☐ Anzahl der Übergabeparameter
- ☐ Datentyp von zumindest einem Parameter
- ☐ Reihenfolge der Parameter

Der Compiler muss bei jedem Methodenaufruf eindeutig erkennen können welche Methode verwendet wird. Unterscheiden sich zwei Methoden sich nur durch ihren Rückgabedatentyp so sind sie nicht überladen. Der Compiler liefert in diesem Fall eine Fehlermeldung.
Das folgende Beispiel zeigt eine Klasse mit einer mehrfach überladenen Methode **`test()`**:

```
public class Overloading
{
  public static int test() { return 1; }
  public static int test(int z) { return 2; }
  public static int test(float z) { return 3; }
  public static int test(int z1, int z2) { return 4; }
  public static int test(int... z) { return 5; }

  public static void main(String[] args)
  {
    System.out.println(test());
    System.out.println(test(2));
    System.out.println(test(2f));
    System.out.println(test(1,2));
    System.out.println(test(1,2,3));
  }
}
```

Ausgabe:

```
1
2
3
4
5
```

Wie das Beispiel zeigt, haben Funktionen mit exakt definierter Parameterliste immer Vorrang gegenüber von Funktionen mit einer VarArg-Parameterliste: Der Aufruf von **`test()`** ruft die parameterlose Methode und nicht die VarArg-Methode auf.

2.3.4 Methoden mit variablen Argumenten: Var-Args

Mit Hilfe von Var-Arg Parametern kann eine beliebig lange Liste von Parametern des gleichen Typs an eine Methode übergeben werden. Innerhalb der Methode werden die Argumente als Array behandelt.

Folgende Regeln müssen dabei beachtet werden:

- Syntax: Datentyp, gefolgt von 3 Punkten, Leerzeichen und Name des Feldes, das die Werte aufnimmt
- Der Var-Arg Typ kann ein primitiver Datentyp oder ein Referenztyp sein
- Neben einem Var-Arg Argument dürfen beliebig viele andere Parameter übergeben werden
- Es darf in einem Methodenaufruf nur einen Var-Arg Parameter geben
- Der Var-Arg Parameter muss an letzter Stelle in der Parameterliste stehen

Das nachfolgende Beispiel zeigt die Verwendung von Var-Arg Parametern anhand der Methode **`summe()`**:

```
public class VarArgs
```

```
{
  public static int summe(int... werte)
  {
    int sum = 0;
    for (int i=0; i<werte.length;i++)
    {
      sum += werte[i];
    }
    return sum;
  }

  public static void main(String[] args)
  {
    int x=10;
    System.out.format("Summe: %4d\n",summe());            // Kein Parameter
    System.out.format("Summe: %4d\n",summe(x,x*2));       // 2 Parameter
    System.out.format("Summe: %4d\n",summe(1,2,3,4));     // 4 Parameter
  }
}
```

Ausgabe:

```
Summe:    0
Summe:   30
Summe:   10
```

2.4 Klassenattribute und Klassenmethoden

Für jede Klasse können Klassenattribute und Klassenmethoden definiert werden. Sie sind durch das Schlüsselwort **`static`** gekennzeichnet und werden auch als statische Variablen oder statische Methoden bezeichnet.

2.4.1 Statischen Variablen

Statische Variablen werden mit dem Klassennamen verwendet, ohne dass eine Instanz der Klasse erzeugt werden muss.

```
public class Lion
{
  private static int count;

  public static void main(String[] args)
  {
     System.out.println(Lion.count);
  }
}
```

Beispiele für statische Variablen und Methoden finden sich z.B. in den Klassen **`Math`**, **`Double`** und **`System.out`**:

```
double radius = Double.parseDouble("1.23");
double CircleArea = Math.PI * Math.pow(radius, 2.0);
System.out.println(Lion.count);
```

Verwendung von statischen Variablen:
Statische Variablen werden dazu verwendet um Daten zu speichern die für alle Objekte einer Klasse den gleichen Wert haben sollen. Sie können für folgende Aufgaben verwendet werden:

- Definition von Konstanten Werten, wie z.B. die Konstanten **`PI`** und **`E`** der Klasse **`Math`**:

```
public static final double E = 2.7182818284590452354;
public static final double PI = 3.14159265358979323846;
```

  Mit dem Schlüsselwort **`final`** wird eine Variable als Konstante definiert, d.h. der Wert darf nach der Initialisierung nicht mehr verändert werden.

- Zählen der Instanzen einer Klasse. Als Zähler wird eine statische Variabel verwendet, die im Konstruktor inkrementiert und im Destruktor dekrementiert wird.

```
public class SomeClass
{
  private static int objCounter;
  public static int getObjCounter() { return objCounter; }

  public SomeClass()
  {
    objCounter++;
  }
}
```

- Wenn eine Methode ein Ergebnis liefern soll, ohne dass dabei Werte in der Klasse gespeichert werden müssen. Die Methoden der Klasse **`Math`**, wie z.B. **`sqrt()`** oder **`pow()`**, berechnen einen gewünschten Wert, ohne dass irgendwelche Daten in der Klasse **`Math`** dafür gespeichert werden müssen

```
System.out.println("Circle-area: "+ (Math.PI * Math.pow(2.5, 2.0)));
```

- Statische Variablen können in allen Klassen einer Applikation verwendet werden. Damit ist es möglich Werte innerhalb aller Klassen einer Applikation zu verwenden, ohne dass Objekte erzeugt und als Parameter übergeben werden müssen:

```
class A
{
  public static int someValue = 123;
}
```

```
class B
{
  public void changeValue()
  {
    A.someValue = 5;
  }
}
```

```
class C
{
  public void useValue()
  {
```

```
    System.out.println(A.someValue);
  }
}
```

2.4.2 Statische Methoden

Klassenmethoden sind gleich wie Klassenvariabeln an die Klasse und nicht an Objekte der Klasse gebunden. Sie werden mit Hilfe des Modifiers static definiert. Ihr Aufruf erfolgt mit Hilfe des Klassen- und nicht des Objektnamens:

```
KlassenName.methodenName()
```

Klassenmethoden können auch mit einem Objektnamen aufgerufen werden. Der Compiler ersetzt den Objektnamen aber automatisch durch den deklarierten Datentyp des Objekts. Da der deklarierte Datentyp aber nicht mit dem tatsächlichen Typ des Objekts übereinstimmen muss, kann es dabei zu Problemen kommen. Daher sollten Klassenmethoden ausschliesslich mit dem Klassennamen aufgerufen werden!
Innerhalb von Klassenmethoden dürfen keine Instanzvariablen verwendet oder Instanzmethoden aufgerufen werden. Der Compiler liefert sonst eine Fehlermeldung der Form:

```
non-static variable xy cannot be referenced from a static context
```

Das folgende Beispiel zeigt die Verwendung von verschiedenen Elementen einer Klasse:

```
public class Methods
{
  private int instVar = 5;             // Instanzvariable
  private static int classVar = 10;    // Klassenvariable

  public void printInstVar()           // Instanzmethode
  {
    System.out.println(instVar);
  }

  public static void printClassVar()   // Klassenmethode
  {
    System.out.println(classVar);
    System.out.println(instVar);       // Compilerfehler!
  }

  public static void main(String[] args)
  {
    Methods.printClassVar();           // Aufruf einer Klassenmethode
    Methods methods = new Methods();
    methods.printInstVar();            // Aufruf einer Instanzmethode
  }
}
```

2.5 Sichtbarkeit und Datenkapselung

Für jede Instanzvariable und Methode wird bei der Deklaration zusätzlich ihre Sichtbarkeit definiert. Mögliche Werte sind:

Sichtbarkeit	***Bedeutung***
`private`	auf die Variable kann nur innerhalb der Klasse zugegriffen werden. Instanzvariablen werden meistens als **`private`** deklariert. Der Zugriff von anderen Klassen erfolgt über getter- und setter-Methoden.
default	wenn keine Sichtbarkeit vor der Variable steht. Auf die Variable kann innerhalb der Klasse und innerhalb des gleichen packages zugegriffen werden
`protected`	Auf die Variable kann innerhalb der Klasse und innerhalb des gleichen packages zugegriffen werden, sowie in Klassen anderer packeges wenn es sich um eine abgeleitete Klasse handelt.
`public`	auf die Variable kann aus jeder Klasse zugegriffen werden

In Java gibt es vier verschiedene Möglichkeiten den Zugriff auf Elemente (Attribute oder Methoden) einer Klasse zu definieren:

- **`private`**
  Auf Elemente mit **`private`**-Access darf nur innerhalb der Klasse zugegriffen werden.
  UML-Symbol: **-**
- default
  Auf Elemente mit default-Access darf in allen Klassen und Methoden desselben packages zugegriffen werden. Wird definiert indem kein Access-Modifier angegeben wird.
  UML-Symbol: ~
- **`protected`**
  Auf Elemente mit **`protected`**-Access kann innerhalb desselben packages zugegriffen werden, bzw. auch in anderen packages, wenn der Zugriff innerhalb einer abgeleiteten Klasse erfolgt.
  UML-Symbol: **#**
- **`public`**
  Auf Elemente mit **`public`**-Access kann überall, uneingeschränkt zugegriffen werden.
  UML-Symbol: **+**

Nachfolgendes Beispiel zeigt die Syntax der vier Access-Modifier für Attribute:

```
public class Access
{
  private int var1;         // private-Access: class-level
  int var2;                 // default-Access: package-level
  protected int var3;       // protected-Access: package + inherited classes
  public int var4;          // public-Access: general
}
```

2.6 Initialisierungsblöcke

Genau wie bei Variablen wird zwischen Instanz- und statischen Initialisierungsblöcken unterschieden.

Instanz-Initialisierungsblöcke werden durch einen Block innerhalb der Klasse, aber außerhalb einer Methode definiert. Sie werden jedesmal beim Instanzieren eines Objekts durchlaufen, und zwar am Beginn des Konstruktors nachdem alle Instanzvariablen erzeugt wurden. Wenn mehrere Blöcke existieren so werden sie von oben nach unten durchlaufen. Das folgende Beispiel zeigt eine Klasse mit zwei Initialisierungsblöcken.

```
public class InitBlocks
{
  {                             // Erster Instanz-Initialisierungsblock:
    System.out.println("Init-Block-1");
  }

  public InitBlocks() {         // Erster Konstruktor:
    System.out.println("Konstruktor-1");
  }

  public InitBlocks(int wert) {// Zweiter Konstruktor:
   System.out.println("Konstruktor-2");
  }

  {                             // Zweiter Instanz-Initialisierungsblock:
    System.out.println("Init-Block-2");
  }

  public static void main(String[] args) {
        InitBlocks ib1 = new InitBlocks();
        InitBlocks ib2 = new InitBlocks(1234);
      }
}
```

Ausgabe:

```
Init-Block-1
Init-Block-2
Konstruktor-1
Init-Block-1
Init-Block-2
Konstruktor-2
```

Die Hauptaufgabe von Initialisierungsblöcken ist es Code aufzunehmen der bei mehreren überladenen Konstruktoren gleich ist. Dadurch wird das Duplizieren von Quellcode vermieden.

Statische-Initialisierungsblöcke werden nur einmal beim Laden der Klasse aufgerufen. Exisitieren mehrere Blöcke so werden sie ebenfalls von oben nach unten durchlaufen. Innerhalb der Blöcke kann nur auf Klassenvariablen zugegriffen werden. Das folgende Beispiel zeigt die Zusammenhänge:

```
public class InitBlocks
{
  {                             // Instanz-Initialisierungsblock
    System.out.println("Instanz-Init-Block");
  }

  static
  {                             // 1. Statischer Initialisierungsblock
    System.out.println("Static-Init-Block 1");
  }
```

```
  public InitBlocks()           // Konstruktor
  {
    System.out.println("Konstruktor");
  }

  static
  {                             // 2. Statischer Initialisierungsblock
    System.out.println("Static-Init-Block 2");
  }

  public static void main(String[] args)
  {
    InitBlocks ib1 = new InitBlocks();
    InitBlocks ib2 = new InitBlocks();
  }
}
```

Ausgabe:

```
Static-Init-Block 1
Static-Init-Block 2
Instanz-Init-Block
Konstruktor
Instanz-Init-Block
Konstruktor
```

3 Datentypen, Operatoren und Anweisungen

3.1 Datentypen

In Java müssen alle Variablen mit einem Datentyp deklariert werden. Es gibt acht primitive Datentypen, weiters kann jede Java-Klasse als Datentyp (Referenztyp) verwendet werden.

Datentyp	*Größe in Bytes*	*Wertebereich*	*Default-Wert*
byte	1	-128 ... +127	0
short	2	-32768 ... 32767	0
int	4	$-2^{31}…2^{31}-1$	0
long	8	$-2^{63}…2^{63}-1$	0
float	4	$+/-3.40282347 * 10^{38}$	0.0
double	8	$+/-1.79769313486231570 * 10^{308}$	0.0
char	2	Alle Unicode Zeichen	**\u0000**
boolean	1	**true, false**	**false**

Die Default-Werte gelten nur für Instanz- und Klassenvariablen.

Literale (die dirkete Darstellung von Werten) sind ganzzahlig immer vom Typ **int** und bei Gleitkommawerten immer vom Typ **double**:

```
byte b1 = 127;                  // Compilerfehler!
byte b2 = 128;                  // Compilerfehler!
int i1 = 1;
int i2 = 2;
byte b3 = i1 + i2;              // Compilerfehler -> Addition zweier int!
```

```
float f1 = 1.0;                // Compilerfehler!
float f2 = 1.0f;
double d1 = 1.0;
```

Underscores können verwendet werden um Zahlen lesbarer zu machen, sie dürfen aber nicht am Anfang oder am Ende der Zahl bzw. vor oder nach dem Komma stehen:

```
int million = 1_000_000;
double million1 = 1_000_000.1;
double thousand1 = 1_000_.1            // Compilerfehler
```

Eine implizite, automatische Typenkonvertierung erfolgt immer vom kleineren zum größeren Datentyp:

byte → short → int → long → float → double

Folgende Zuweisungen sind damit gültig/ungültig:

```
byte b1 = 1;
int i1 = 2;
long l1 = 4;
float f1 = 1.0f;
double d1 = 2.0;
d1 = f1;                       // float nach double -> ok
f1 = b1;                       // long nach float -> ok
f1 = b1;                       // byte nach float -> ok
l1 = i1;                       // int nach long -> ok
i1 = b1;                       // byte nach int -> ok

f1 = d1;                       // double nach float -> Compilerfehler!
b1 = i1;                       // int nach byte -> Compilerfehler!
l1 = f1;                       // float nach long -> Compilerfehler!
i1 = l1;                       // long nach int -> Compilerfehler!
```

Mit einem Typecast kann eine explizite Typenumwandlung erfolgen, wobei eventuell vorhandene Kommastellen abgeschnitten werden, oder ganze Teile einer Zahl verloren gehen können:

```
int i1 = 0;
long l1 = 4;
float f1 = 123.99f;
double d1 = 2.123;
i1 = (int)f1;                  // Wert von i1 = 123
f1 = (float)d1;                // WWrt von f1 = 2.123
i1 = (int)d1;                  // wert von i1 = 2
```

Variablen vom Typ **char** können jedes beliebige Unicode-Zeichen aufnehmen. Es dürfen nur einfache Hochkomma verwendet werden. Weiters gelten dieselben Escape-Sequenzen wie in C:

```
char c1 = 'A';
char c2 = '\t';          // Tabulator-Zeichen
char c3 = '\u263A';      // Smilie: ☺
```

Variablen vom Typ **String** sind Objekte, müssen aber nicht mit dem Schlüsselwort **new** erzeugt werden. Bei der Zuweisung müssen doppelte Hochkomma verwendet werden:

```
String s1 = "Hello";
String s2 = new String("Hello");
```

Die **String**-Klasse enthält eine Reihe von Methoden zur Manipualtion von Strings.

3.2 Operatoren

Die Operatoren für die primitiven Datentypen unterscheiden sich zwischen Java und C nur geringfügig.

3.2.1 Operatoren für numerische Datentypen

Für die numerischen Datentypen **byte**, **short**, **int**, **long**, **float** und **double** stehen folgende Operatoren zur Verfügung:

Operator	*Bedeutung*
+	Addition
-	Subtraktion
*****	Multiplikation
/	ganzzahlige Division
%	Modulo-Division
++	pre- und post-Inkrement
--	pre- und post-Dekrement
+=, -=, …	Zusammengesetzte Operatoren

Im Unterschied zu C können die Operatoren **%**, **++**, **--** auch für Gleitkommezahlen verwendet werden.

3.2.2 Vergleichsoperatoren

Sie dienen zum Vergleich von 2 Werten (Zahlen oder Buchstaben) und sind identisch zu C:

Operator	*Bedeutung*
==	gleich
!=	ungleich
<	kleiner
>=	kleiner-gleich
>	größer
>=	größer-gleich

3.2.3 Logische Operatoren

Für die numerischen Datentypen **byte**, **short**, **int**, **long**, **float** und **double** stehen folgende Operatoren zur Verfügung:

Operator	*Bedeutung*
`!`	negation: ändern des Wahrheitswertes
`&&, &`	logische AND mit und ohne Short-circuit evaluation
`||, |`	logische OR mit und ohne Short-circuit evaluation
`^`	logische XOR

Im Unterschied zu C kann mit den Operatoren `&`, `|` die short-circuit evaluation ausgeschaltet werden, d.h jeder logische Ausdruck wird vollständig bis zum Ende ausgewertet, auch wenn das Ergebnis schon früher feststeht.

3.2.4 Bit-Shift-Operatoren

Dienen zum Verschieben der Bits einer Zahl:

Operator	*Bedeutung*
`<<`	links-shift
`>>`	rechts-shift
`>>`	rechts –shift mit nachrückender Null
`&, |, ^`	Bitweise AND, OR, XOR Verknüpfung

3.2.5 Der Fragezeichen-Operator

Operator	*Bedeutung*
`?`	Ternärer Fragezeichen Operator

Der `?`-Operator ersetzt, gleich wie in C, eine einfache if-Anweisung:

boolean expression **`?`** *expression1* **`:`** *expression2*

Beispiel:

```
int num = new Random().nextInt();
String evenOrOdd = num % 2 == 0 ? "even" : "odd";
```

3.3 Arrays

Felder werden in Java mit **`new`** erzeugt:

```
int[] numbers = new int[3];
```

Datentyp des Feldes — Array Symbol — Größe des Feldes

Alle Elemente eines Feldes werden automatisch mit einem Default-Wert initialisiert, d.h. Zahlen mit 0, boolean mit **`false`**, alle Objekte mit **`null`**.
Felder sind auch als Return-Type bei Methoden erlaubt.

Felder können beim Erzeugen auch gleich initialisiert werden:

```
int[] num1 = new int[]{10,11,12};
int[] num2 = {10,11,12}                  // Kurzform der Initialisierung
```

Der Zugriff auf Elemente eines Feldes erfolgt über den Index, der gleich wie in C bei 0 begint.

```
String[] names = new String[2];
names[0] = "Homer";
names[1] = "Bart";
```

Wird auf ein Element außerhalb des gültigen Bereichs zugegriffen, so wird zur Laufzeit eine **ArrayIndexOutOfBounds**-Exception geworfen:

```
String[] hexStrings = { "Cafe", "FF"};
System.out.println(hexStrings[2]);
```

Ausgabe:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
```

Auf die Größe eines Feldes kann mit der **length** Eigenschaft zugegriffen werden:

```
String[] hexStrings = { "Cafe", "FF"};
for (int i = 0; i < hexStrings.length; i++)
{
  System.out.println(hexStrings[i]);
}
```

Ausgabe:

```
Cafe
FF
```

Für den Zugriff auf alle Elemente eines Feldes kann auch die foreach-Schleife verwendet werden:

```
String seasons[] = { "Spring", "Summmer", "Fall", "Winter"};
for (String season : seasons)
{
  System.out.println(season);
}
```

Ausgabe:

```
Spring
Summmer
Fall
Winter
```

Mehrdimensionale Felder:

4 Exception-Handling

Exceptions sind Laufzeitfehler die während der Ausführung eines Programms auftreten. Tritt eine Exception auf so sagt man eine Exception wird geworfen. Typische Beispiele für Exceptions sind z.B. der Zugriff auf nicht vorhandene Feld-Elemente (**`ArrayIndexOutOfBoundsException`**) oder der Versuch einen nicht-numerischen Wert mit Hilfe einer Wrapper-Klasse in eine Zahl umzuwandeln (**`NumberFormatException`**).
Exceptions können mit Hilfe von **`try-catch`** Anweisungen abgefangen werden.

Grundsätzlich läuft das Exception-Handling in Java wie folgt ab:

- Innerhalb einer Anweisung wird eine Exception ausgelöst
- Die Exception kann innerhalb derselben Methode mit einem try-catch-Block abgefangen werden.
- Die Exception kann an die aufrufende Methode weitergegeben werden. Dort kann sie wiederum abgefangen oder weitergegeben werden.
- Die Exception wird nie abgefangen und führt zu einem Programmabsturz

4.1 Das Abfangen von Exceptions mit `try-catch`-Blöcken

Die **`try-catch`** Anweisung ist wie folgt aufgebaut:

```
try
{
  // In diesem Block sind Anweisungen in denen
  // ein Laufzeitfehler auftreten kann
}
catch (FirstException ex)
{
  // In diesem Block sind Anweisungen
  // die zur Fehlerbehandlung von FirstException dienen
}
catch (SecondException ex)
{
  // In diesem Block sind Anweisungen
  // die zur Fehlerbehandlung von SecondException dienen
}
```

Tritt im **`try`**-Block eine Exception auf, so werden alle weiteren Anweisungen in diesem Block übersprungen und das Programm setzt mit der ersten Anweisung in demjenigen **`catch`**-Block fort, der den passenden Exception-Typ hat. Im **`catch`**-Block sollte die Fehlerbehandlung durchgeführt werden, also z.B. die Ausgabe einer Fehlermeldung mit einer JOptionPane.
Zu jedem **`try`**-Block kann es einen oder mehrere **`catch`**-Blöcke geben.
Sowohl im **`try`**- als auch im **`catch`**-Block können beliebig viele Anweisungen stehen.

Achtung: Bleibt der catch-Block leer so wird das Programm ohne irgendeine Fehlermeldung fortgesetzt. Das führt meist dazu dass fehlerhafte Programme weiterlaufen. Exceptions sollten daher niemals mit einem leeren **`catch`**-Block abfangen werden!!!

4.2 Das Weiterleiten von Exceptions mit `throws`

Wird eine Exception nicht mit einem **`try-catch`**-Block abgefangen so kann sie mit **`throws`** an die aufrufende Methode weitergegeben werden. In dieser Methode stehen wieder beide Möglichkeiten zur Verfügung: entweder die Exception abfangen oder weitergeben.
Das folgende Beispiel zeigt diesen Mechanismus:

```
public class TolleBLKlasse
{
  public void parseZahl(String strWert) throws NumberFormatException
  {
    int wert = Integer.parseInt(strWert);
  }
}
```

```
public class AufrufendeKlasse
{
  public void test()
  {
    try
    {
      new TolleBLKlasse().parseZahl("1.5");
    }
    catch (NumberFormatException ex)
    {
      System.out.println("Bitte nur ganze Zahlen verwenden!!!")
    }
  }
}
```

Ausgabe:

```
Bitte nur ganze Zahlen verwenden!!!
```

Anstatt die Anweisung in einen **`try-catch`**-Block einzuhüllen wird die Exception mit **`throws`** an die aufrufende Methode weitergeleitet.
Der späteste Zeitpunkt zu dem eine Exception abgefangen werden kann ist in der **`main`**-Methode. Wird sie dort nicht abgefangen so erfolgt die Weitergabe an die JVM und das Programm bricht mit einer Fehlermeldung ab.

In Java wird zwischen sogenannten checked-und unchecked-Exceptions unterschieden. Der Unterschied besteht darin, dass der Compiler bei checked-Exceptions überprüft ob die Exception entweder mit einem **`try-catch`**-Block abgefangen wird oder mit **`throws`** weitergeleitet wird. Ist beides nicht der Fall so erfolgt eine Fehlermeldung durch den Compiler. Bei unchecked-Exceptions erfolgt keine Überprüfung durch den Compiler, d.h.es bleibt dem Entwickler überlassen einen **`try-catch`**-Block zu implementieren oder die Exception weiterzuleiten.

4.3 Das Erzeugen neuer Exceptions mit `throw`

Java bietet auch die Möglichkeit eigene Exceptions zu erzeugen bzw. eigene Exception-Klassen zu definieren. Zum Werfen einer neuen Exception wird das Schlüsselwort **`throw`** verwendet. Im folgenden Beispiel wird eine Exception geworfen um eine Division durch 0 zu verhindern:

```
public class BLKlasse
{
  public int dividiere(int dividend, int divisor) throws ArithmeticException
  {
    if (divisor == 0)
    {
      throw new ArithmeticException("Division durch 0!!!");
    }
    return dividend / divisor;
  }

  public static void main(String[] args)
  {
    BLKlasse blk = new BLKlasse();
    try
    {
      System.out.println(blk.dividiere(10,0));
    }
    catch (ArithmeticException e)
    {
      System.out.println(e.toString());
    }
  }
}
```

Ausgabe:

```
java.lang.ArithmeticException: Division durch 0!!!
```

Wird eine Exception mit **`throw`** geworfen so kann sie entweder wieder in einem **`try-catch`**-Block abgefangen werden oder mit **`throws`** weitergeleitet werden.

Einige nützliche Methoden der Exception-Klassen zur Ausgabe von Informationen sind:

`public String ex.toString()`
Liefert eine Beschreibung des Fehlers in der Form: *Exception-Typ, Kurzbeschreibung*

`public String ex.getMessage()`
Liefert eine Beschreibung des Fehlers in der Form: *Kurzbeschreibung*

`public void ex.printStackTrace()`
Schreibt eine Stacktrace, d.h. alle Klassennamen und Methodennamen von der Zeile in der die Exception aufgetreten ist bis hinauf zur main()-Methode, auf den standard Error-Stream (unter NetBeans ist das die Konsole).

4.4 Die `finally`-Anweisung

Ein **`try-catch`**-Block kann um die **`finally`**-Anweisung erweitert werden. Die Anweisungen Im finally-Block werden immer (!) durchlaufen, gleichgültig ob der **`try`**-Block teilweise oder vollständig durchlaufen wird, ob der **`catch`**-Block durchlaufen wird, oder die Exception mit **`throw`** weitergeworfen wird. Selbst wenn der **`try`**-Block mit **`break`**, **`continue`** oder **`return`** verlassen wird, so wird vorher der **`finally`**-Block durchlaufen. Einzige Ausnahme ist das Beenden der Applikation mit **`System.exit(0).`**
Der **`finally`**-Block ist daher ideal für alle Arten von Aufräumarbeiten, wie z.B. das Schließen von Dateien etc.

Das folgende Beispiel zeigt die Verwendung eines **finally**-Blocks:

```
public class BLKlasse
{
  public int dividiere(int dividend, int divisor) throws ArithmeticException
  {
    try
    {
      if (divisor == 0)
      {
        throw new ArithmeticException("Division durch 0!!!");
      }
    }
    catch (ArithmeticException e)
    {
      throw e;
    }
    finally
    {
      System.out.println("Gruesse aus dem finally-Block! ");
    }
    return dividend / divisor;
  }

  public static void main(String[] args)
  {
    BLKlasse blk = new BLKlasse();
    try
    {
      System.out.println(blk.dividiere(10,2));
      System.out.println(blk.dividiere(10,0));
    }
    catch (ArithmeticException e)
    {
      System.out.println(e.toString());
      e.printStackTrace();
    }
  }
}
```

Ausgabe:

```
Gruesse aus dem finally-Block!
5
Gruesse aus dem finally-Block!
java.lang.ArithmeticException: Division durch 0!!!
    at exceptions.BLKlasse.dividiere(BLKlasse.java:20)
    at exceptions.BLKlasse.main(BLKlasse.java:40)
```

Es gibt auch die Möglichkeit **try-finally**-Blöcke ohne einen **catch**-Block zu implementieren.

4.5 Wichtige Java-Exceptions

Die folgende Tabelle gibt eine Übersicht über einige wichtige Java-Exceptions:

Exception-Typ	*Bedeutung*	*Ausgelöst von*
`ArrayIndexOutOfBoundsException`	Beim Zugriff auf ein nicht-dimensioniertes Feldelement	JVM
`IndexOutOfBoundsException`	Beim Zugriff auf ein nicht-dimensioniertes Collection-Element	JVM
`NumberFormatException`	Beim Versuch einen String in eine Zahl umzuwandeln	API
`ArthmeticException`	Gleitkommazahl	API
`ClassCastException`	Beim Versuch eine Klasse in einen illegalen Klassentyp umzuwandeln	JVM
`NullPointerException`	Beim Versuch ein Element anzusprechen, wenn das dazugehörige Objekt noch nicht mit new erzeugt wurde	JVM

5 Wichtige Klassen der Java-API

In diesem Abschnitt werden einige wichtige Java-Klassen mit häufig verwendeten Methoden vorgestellt. Die vollständige Beschreibung dieser Klasse ist in der Java API-Documatation nachzulesen.

5.1 Die Klasse `Scanner`

Dient zum Einlesen von der Konsole. Der Default-Delimiter ist ein Whitespace, d.h. Leerzeichen, Tabulator oder Zeilenumbruch.

Beispiel:

```
Scanner scan = new Scanner(System.in);

System.out.print("Ganze Zahl eingeben: ");
int i1 = scan.nextInt();
System.out.print("Kommazahl eingeben: ");
float f1 = scan.nextFloat();
System.out.print("Text eingeben: ");
String s1 = scan.next();

System.out.format("int: %d\nfloat: %f\nString: %s\n",i1,f1,s1);
```

Ausgabe:

```
Ganze Zahl eingeben: 123
Kommazahl eingeben: 1,9
Text eingeben: Test
int: 123
float: 1,900000
String: Test
```

5.1.1 Methoden der Klasse `Scanner`

`int nextInt()`
Liest die nächste int-Zahl ein

`long nextLong()`
Liest die nächste long-Zahl ein

`float nextFloat()`
Liest die nächste float-Zahl ein

`double nextDouble()`
Liest die nächste double-Zahl ein

`String next()`
Liest den nächsten String ein

`String nextLine()`
Liest die gesamte nächste Zeile als String ein

`void UseDelimiter(String delim)`
Zum Ändern des Delimiters

5.2 Die Klasse `Math`

Dient zur Berechnung mathematischer Ausdrücke:

5.2.1 Methoden und Variablen der Klasse `Math`

`static double PI`
Variable die den Wert von π als **`double`** liefert

`static double pow(double a, double b)`
liefert den Wert von a hoch b

`static double sqrt(double a)`
liefert den Wert der Quadratwurzel von a

`static double sin(double a)`
liefert den Wert von Sinus von a

`static double cos(double a)`
liefert den Wert von Cosinus von a

`static double tan(double a)`
liefert den Wert von Tangens von a

`static E abs(E a)`
liefert den Absolutbetrag von a. Die Methode ist überladen für **`int`**, **`long`**, **`float`** und **`double`**

5.3 Die Klasse `Random`

Dient zum Erzeugen von Zufallszahlen.

5.3.1 Methoden der Klasse `Random`

`int nextInt()`
Zufallszahl aus dem int-Bereich

`int nextInt(int bound)`
Zufallszahl zz aus dem Bereich 0 <= zz < bound

`long nextLong()`
Zufallszahl aus dem long-Bereich

`float nextFloat()`
float Zufallszahl zz aus dem 0 <= zz < 1

`double nextDouble()`
double Zufallszahl zz aus dem 0 <= zz < 1

`boolean nextBoolean()`
zufälliger boolean: **`true`** oder **`false`**

5.4 Die Klasse `String`

Zeichenketten können in Java mit Hilfe der Klasse **`String`** (package **`java.lang`**) implementiert werden. Jedes Zeichen ist vom Typ **`char`** und wird im Unicode-Zeichensatz kodiert. Die **`String`** Klasse ist **`final`**, d.h. es kann keine Klasse von ihr abgeleitet werden. **`String`**-Objekte sind immutable, d.h sie sind in Inhalt und Länge konstant - ein **`String`** kann nicht verändert werden, es können nur neue **`String`**-Objekte erzeugt werden.
Die Zeichen innerhalb von **`String`**-Objekten sind durchnummeriert - die Nummerierung beginnt bei Null.

5.4.1 Wichtige Methoden der Klasse `String`

Konstruktoren:

`String()`
Erzeugt ein neues String-Objekt

`String(String value)`
Erzeugt ein neues String-Objekt mit dem Wert von value

Methoden zur Stringmanipulation:

`String toLowerCase()`
Liefert einen neuen String in dem alle Großbuchstaben durch Kleinbuchstaben ersetzt werden.

`String toUpperCase()`
Liefert einen neuen String in dem alle Kleinbuchstaben durch Großbuchstaben ersetzt werden.

`String replace(char oldChar, char newChar)`
Liefert einen neuen String in dem alle Vorkommen von **`oldChar`** durch **`newChar`** ersetzt werden.

`String replace(CharSequence target, CharSequence replacement)`
Liefert einen neuen String in dem alle Vorkommen von **`target`** durch **`replacement`** ersetzt werden. **`CharSequenge`** ist eine Klasse die eine lesbare Zeichenfolge enthält (ähnlich wie **`String`**).

`String concat(String str)`
Liefert einen neuen **`String`** der den Inhalt von **`str`** an das **`this`**-Objekt anhängt.

`String subString(int begin)`
Liefert einen Teilstring der an der Position **`begin`** anfängt und bis zum Ende des Strings geht. Bei Übergabe eines ungültigen Index wird eine **`IndexOutOfBoundsException`** geworfen.

`String subString(int begin, int end)`

Liefert einen Teilstring der an der Position **begin** anfängt und an der Position end-1 endet. Bei Übergabe eines ungültigen Index wird eine **IndexOutOfBoundsException** geworfen.

String trim()
Liefert einen Teilstring der alle Whitespace-Zeichen (Leerzeichen, Tab, CR) am Beginn und Ende des Strings entfernt.

String[] split(String regex)
Liefert einen String-Array zurück, der alle Teilstrings enthält die durch Aufteilen des String in Teile um die übergebene RegularExpression entsteht. Sonderzeichen, die eine spezielle Bedeutung innerhalb einer RegularExpression haben, muss ein `'\\'` vorangestellt werden. Solche Zeichen sind z.B. `'.'`, `'+'`, `'*'` und `'?'`, die durch `'\\.'`, `'\\+'`, `'\\*'` und `'\\?'` ersetzt werden müssen.

static String format(String format, Object... args)
Liefert einen formatierten String entsprechend dem format-String zurück.

Methoden zum Suchen:

char charAt(int index)
Liefert das Zeichen an der Position **index** zurück. Bei einem ungültigen Index wird eine **IndexOutOfBoundsException** geworfen.

int indexOf(char c)
Liefert den Index des ersten Vorkommens von c im **String** bzw. -1 wenn c nicht vorkommt.

int indexOf(char c, int fromIndex)
Liefert den Index des ersten Vorkommens von c im **String** beginnend beim spezifizierten Index.

int indexOf(String str)
Liefert den Index des ersten Vorkommens von **String str** im **String** bzw. -1 wenn **str** nicht vorkommt.

int indexOf(String str, int fromIndex)
Liefert den Index des ersten Vorkommens von **str** im **String** beginnend beim spezifizierten Index.

int lastIndexOf(char c)
Liefert den Index des letzten Vorkommens von c im **String** bzw. -1 wenn c nicht vorkommt.

Methoden zum Vergleichen:

Beim Vergleichen von Strings mit dem **==** Operator wird auf referenzielle Gleichheit geprüft, d.h. ob beide String-Referenzen auf das gleiche String-Objekt verweisen. Um zwei String-Objekte auf inhaltliche Gleichheit zu überprüfen müssen die nachfolgenden Methoden (wie z.B.: **equals()**) verwendet werden. Zu beachten ist auch, dass Stringobjekte mit unterschiedlichem Inhalt in einen eigenen String-Pool kommen. Wenn daher zwei Strings der gleiche, konstante Wert zugewiesen wird, so können sie auf die gleiche Referenz im String-

Pool verweisen. Deshalb kann ein Vergleich der beiden Strings mit dem **==** Operator den Wert **true** liefern.

boolean equals(Object obj)
Vergleicht inhaltlich das **this**-Objekt mit dem übergebenen **String**. Liefert **false** wenn der Übergabeparameter kein **String**-Objekt ist.

boolean equalsIgnoreCase(Object obj)
Vergleicht inhaltlich das **this**-Objekt mit dem übergebenen **String** ohne die Groß/Kleinschreibung zu berücksichtigen.

int compareTo(String str)
Vergleicht inhaltlich das **this**-Objekt mit dem übergebenen **String** und liefert:
<0 wenn **this** < **str**
= 0 wenn **this** = **str**
>0 wenn **this** > **str**

int compareToIgnoreCase(String str)
Gleich wie **compareTo()** nur ohne die Groß/Kleinschreibung zu berücksichtigen.

boolean contains(String str)
Liefert **true** wenn das **this**-Objekt den **String str** enthält.

boolean startsWith(String str)
Liefert **true** wenn das **this**-Objekt mit **str** beginnt.

boolean endsWith(String str)
Liefert **true** wenn das **this**-Objekt mit **str** endet.

Weitere Methoden:

int length()
Liefert die Länge des Strings zurück.

boolean isEmpty()
Liefert **true** wenn die Länge des Strings gleich Null ist.

5.4.2 Verkettung von `Strings`

Zur Verkettung von **String** Objekten kann der Operator **+** verwendet werden. Ist bei der Verkettung zumindest einer der Operanden vom Typ **String** so wird der zweite Operand ebenfalls in einen **String** konvertiert. Für Objekte wird dabei die **toString()**-Methode aufgerufen.
Das folgende Beispiel zeigt einige **String**-Verkettungen:

```
System.out.println("Hallo " + 2 + "AHIF");
System.out.println("Hallo " + 1 + 1 + "AHIF");
System.out.println("Hallo " + (1+1) + "AHIF");
Integer int_obj = new Integer(2);
System.out.println("Hallo " + int_obj + "AHIF");
```

Ausgabe:

```
Hallo 2AHIF
```

```
Hallo 11AHIF
Hallo 2AHIF
Hallo 2AHIF
```

5.4.3 Verkettung von `String`-Methoden

Da die meisten **`String`**-Methoden wieder ein **`String`**-Objekt zurückgeben können die Methodenaufrufe verkettet werden. Im Sinne einer guten Lesbarkeit des Programms sollte die Verkettung aber auf drei Aufrufe beschränkt werden.
Das folgende Beispiel zeigt die Verwendung einiger **`String`**-Methoden:

```
String str = new String("    Heute ist Freitag der 11.11.2011   ");

System.out.println(str.trim());        // Veraendert str nicht
System.out.println(str);
str = str.trim();
String[] teile = str.split(" ");       // Splittet str nach dem Leerzeichen
for (String s : teile)
{
  System.out.println(s);
}
teile = str.split("\\.");              // Achtung beim Splitten nach dem "."
for (String s : teile)
{
  System.out.println(s);
}
str = str.replace("11.", "elfte ").substring(10);
System.out.println(str);
```

Ausgabe:

```
Heute ist Freitag der 11.11.2011
   Heute ist Freitag der 11.11.2011
Heute
ist
Freitag
der
11.11.2011
Heute ist Freitag der 11
11
2011
Freitag der elfte elfte 2011
```

5.5 Die Klassen `StringBuffer` und `StringBuilder`

Für die Bearbeitung von Strings stehen in Java die beiden Klassen **`StringBuffer`** und **`StringBuilder`** zur Verfügung. Im Gegensatz zur Klasse **`String`** sind diese beiden Klassen nicht immutable und ermöglichen das Bearbeiten eines Strings ohne jedesmal ein neues Objekt erzeugen zu müssen.
Die Methoden der beiden Klassen sind identisch - sie unterscheiden sich dadurch, dass Objekte der Klasse **`StringBuffer`** Thread-safe sind (d.h. in Multithread-Applikationen eingesetzt werden können) während das bei Objekten der Klasse **`StringBuilder`** nicht der Fall ist. Der Vorteil von **`StringBuilder`**-Objekten liegt in der besseren Performance.

5.5.1 Wichtige Methoden der Klassen `StringBuffer` und `StringBuilder`

Die folgenden Methoden sind der Klasse **`StringBuilder`** entnommen. Die exakt gleichen Methoden existieren für die Klasse **`StringBuffer`** (der return-Type ist entsprechend auszutauschen).

Konstruktoren:

`StringBuilder()`
Erzeugt einen leeren **`StringBuilder`**.

`StringBuilder(String str)`
Erzeugt einen neuen **`StringBuilder`**, der eine Kopie von str enthält.

`StringBuilder(int capacity)`
Erzeugt einen neuen **`StringBuilder`** mit der angegebenen Größe.

Methoden zur Stringmanipulation:

`StringBuilder append(String str)`
Hängt **`str`** an das bestehende **`StringBuilder`**-Objekt an, und gibt die darauf verweisende Referenz zurück. Es gibt eine Reihe weiterer, überladener **`append()`**-Methoden die dazu dienen Variablen der unterschiedlichsten Datentypen (**`int`**, **`float`**, …) an das **`StringBuilder`**-Objekt anzuhängen.

```
StringBuilder sb = new StringBuilder("guten ");
sb.append("tag!");
System.out.println(sb);         // Ausgabe ist "guten tag!"
```

`StringBuilder delete(int start, int end)`
Entfernt alle Zeichen von der Position **`start`** bis zur Position **`end`**-1 aus dem **`StringBuilder`**-Objekt.

```
StringBuilder sb = new StringBuilder("0123456789");
sb.delete(4,6);
System.out.println(sb);         // Ausgabe ist "01236789"
```

`StringBuilder insert(int offset, String str)`

Fügt den Teilstring **`str`** an der Position **`offset`** in das **`StringBuilder`**-Objekt ein. Alle weiteren Zeichen werden nach hinten verschoben.

```
StringBuilder sb = new StringBuilder("0123456");
sb.insert(4,"---");
System.out.println(sb);         // Ausgabe ist "0123---456"
```

`StringBuilder reverse()`

Dreht die Reihenfolge der Zeichen im **`StringBuilder`**-Objekt um.

```
StringBuilder sb = new StringBuilder("This is Java");
sb.reverse();
System.out.println(sb);         // Ausgabe ist "avaJ si sihT"
```

`StringBuilder deleteCharAt(int index)`

Löscht das Zeichen an der Position **`index`**.

```
StringBuilder sb = new StringBuilder("cheat");
sb.deleteCharAt(2);
System.out.println(sb);         // Ausgabe ist "chat"
```

`StringBuilder setCharAt(int index, char c)`

Ersetzt das Zeichen an der Position **`index`** durch das Zeichen **`c`**.

```
StringBuilder sb = new StringBuilder("cheat");
sb.setCharAt(4,'p');
System.out.println(sb);         // Ausgabe ist "cheap"
```

`StringBuffer replace(int start, int end, String str)`

Ersetzt die Zeichen von Position **`index`** bis Position **`end`**-1 durch den **`String str`**.

```
StringBuilder sb = new StringBuilder("10.Mrz.2011");
sb.replace(3,6,"Apr");
System.out.println(sb);         // Ausgabe ist "10.Apr.2011"
```

Weitere Methoden:

`String toString()`

Gibt den Wert des **`StringBuilder`**-Objekts als **`String`** zurück.

`int length()`

Liefert die Länge des **`StringBuilder`**-Objekts zurück.

5.6 Die Klasse `Object`

Die Klasse **`java.lang.Object`** ist die oberste Klasse der Java-Ableitungshierarchie. Alle weiteren Java-Klassen sind von **`Object`** abgeleitet, und erben damit alle Methoden dieser Klasse. Zwei relativ häufig benötigte Methoden sind **`toString()`** und **`equals()`**.

5.6.1 Die Methode `toString()`

Die **`toString()`**-Methode gibt die **`String`**-Repräsentation des jeweiligen Objekts zurück. Die von der Klasse **`Object`** vererbte **`toString()`**-Methode gibt den voll qualifizierten Klassennamen gefolgt vom Hashcode zurück:

```
public String toString()
{
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Wird die **`toString()`**-Methode in einer selbst erstellten Klasse nicht überschrieben so wird die soeben gezeigte **`toString()`**-Methode der Klassen **`Object`** aufgerufen:

```
package daten;

public class Person
{
  private String vorname;
  private String nachname;

  public Person(String vorname, String nachname)
  {
    this.vorname = vorname;
    this.nachname = nachname;
  }

  public static void main(String[] args)
  {
    Person p = new Person("James","Gosling");
    System.out.println(p);
  }
}
```

Ausgabe:

```
daten.Person@3e25a5
```

Wenn notwendig wird daher die **`toString()`**-Methode überschrieben, wie das folgende Beispiel zeigt:

```
public class Person
{
  private String vorname;
  private String nachname;

  public Person(String vorname, String nachname)
  {
    this.vorname = vorname;
```

```
    this.nachname = nachname;
  }

  @Override
  public String toString()
  {
    return "Person{"+"vorname="+vorname+", nachname="+nachname+'}';
  }

  public static void main(String[] args)
  {
    Person p = new Person("James","Gosling");
    System.out.println(p);
  }
}
```

Ausgabe:

```
Person{vorname=James, nachname=Gosling}
```

5.6.2 Die Methode `equals()`

Die Methode **`equals()`** wird dazu verwendet zwei Objekte der gleichen Klasse inhaltlich zu vergleichen (mit dem == Operator wird nur auf referenzielle Gleichheit geprüft!). Die von **`Object`** geerbte Methode überprüft allerdings nur die referenzielle Gleichheit:

```
public boolean equals(Object obj)
{
  return (this == obj);
}
```

Das Überschreiben der **`equals()`**-Methode könnte wie folgt aussehen:

```
public class Zahlen
{
  private int a;
  private int b;

  public Zahlen(int a, int b)
  {
    this.a = a;
    this.b = b;
  }

  @Override
  public boolean equals(Object obj)
  {
    if (obj == null)
    {
      return false;            // return false wenn obj null-Pointer
    }
    if (getClass() != obj.getClass())
    {
      return false;            // return false wenn Objekte verschiedener
    }                          // Klassen verglichen werden
    final Zahlen other = (Zahlen)obj;
```

```
    if (this.a != other.a)
    {
      return false;            // return false wenn Werte von a verschieden
    }
    if (this.b != other.b)
    {
      return false;            // return false wenn Werte von b verschieden
    }
    return true;               // return true wenn Werte von a und b gleich
  }

  public static void main(String[] args)
  {
    Zahlen z1 = new Zahlen(1,2);
    Zahlen z2 = new Zahlen(1,2);
    System.out.println("Referenziell gleich: " + (z1 == z2));
    System.out.println("Inhaltlich gleich:   " + (z1.equals(z2)));
  }
}
```

Ausgabe:

```
Referenziell gleich: false
Inhaltlich gleich:   true
```

5.7 Klassen für Datum und Uhrzeit

Seit Java 8 gibt es eine neue API für Datum und Uhrzeit in den packages **`java.time`** und **`java.time.format`**. Die wichtigsten davon sind:

- **`java.time.LocalTime`**: Klasse zur Abbildung einer Uhrzeit
- **`java.time.LocalDate`**: Klasse zur Abbildung eines Datums
- **`java.time.LocalDateTime`**: Klasse zur Abbildung von Datum und Uhrzeit
- **`java.time.format.DateTimeFormatter`**: Klasse zur Formattierung von Datum und Uhrzeit.

5.7.1 Datum und Uhrzeit

Unter Java 8 wurden völllig neue Klassen für Datum und Uhrzeit eingeführt. Die früheren Klassen **`Date`**, **`Calendar`** und **`SimpleDateFormat`** waren teilweise umständlich zu verwenden und haben einige Eigenheiten gehabt, die in der neuen Date-Time-API beseitigt wurden.
Wichtiges Merkmal der neuen, umfangreichen, API ist es, dass viele Methoden statisch verwendet werden, und viele Klassen keine Informationen über die Zeitzone enthalten, wie z.B. die Klassen **`LocalTime`**, **`LocalDate`** und **`LocalTimeDate`** zum Speichern einer Zeit, eines Datums bzw. von Zeit und Datum. Zum Erzeugen von Objekten dieser Klassen werden die statischen Methoden **`now()`** und **`of()`** verwendet:

`public static now()`
Erzeugt ein Objekt mit den aktuellen Systemzeit-Werten: für die Klasse **`LocalTime`** die Uhrzeit, für **`LocalDate`** das Datum und für **`LocalDateTime`** das Datum und die Uhrzeit.

Beispiel:

```
import java.time.*;
public static void main(String[] args)
{
  System.out.println(LocalTime.now());
  System.out.println(LocalDate.now());
  System.out.println(LocalDateTime.now());
}
```

Ausgabe wenn das Programm am 20.September 2015 läuft:

```
19:30:10.125
2015-09-20
2015-09-20T19:30:10.295
```

Die erste Ausgabe enthält die Uhrzeit in Stunden, Minuten, Sekunden und Millisekunden.
Die zweite Ausgabe enthält das Datum in der Form Jahr, Monat und Tag.
Die dritte Ausgabe enthält das Datum und die Uhrzeit, getrennt durch ein T.
Objekten dieser 3 Klassen können nicht mit **`new`** erzeugt werden!

Um Objekte mit einem bestimmten Datum und/oder Uhrzeit zu erzeugen, wird die statische Methode **`of()`** verwendet:

```
LocalTime time1 = LocalTime.of(8, 15);           // Stunden, Minuten
LocalTime time2 = LocalTime.of(8, 15, 30);       // + Sekunden
LocalTime time3 = LocalTime.of(8, 15, 30, 125);  // + Nanosekunden
```

Erzeugen von Objekten mit einem bestimmten Datum:

```
LocalDate date1 = LocalDate.of(2015, 1, 15);             // Jahr, Monat, Tag
LocalDate date2 = LocalDate.of(2015, Month.JANUARY, 15); // Jahr, Monat, Tag
```

Ein wichtiger Unterschied zu früheren Datumsklassen ist, dass die Nummerierung der Monate bei 1 (für Jänner) und nicht bei 0 beginnt.
Bei falschen Werten für Datum oder Uhrzeit wird eine Exception geworfen:

```
LocalDate date1 = LocalDate.of(2015,2,29);
```

Ausgabe:

```
Exception in thread "main" java.time.DateTimeException:
Invalid date 'February 29' as '2015' is not a leap year
```

Hier wird erkannt, dass 2015 kein Schaltjahr ist und es daher keinen 29. Februar gibt.

Beim Erzeugen von **`LocalDateTime`**-Objekten sind alle 6 Kombinationen der zuvor gezeigten **`of()`**-Methoden von Uhrzeit und Datum erlaubt:

```
LocalDateTime ldt1 = LocalDateTime.of(2015, 4, 1, 10, 45); // 1.4.2015 10:45
LocalDateTime ldt2 = LocalDateTime.of(2015, Month.APRIL, 1, 10, 45);
```

Weiters gibt es eine **`of()`**-Methoden an die ein **`LocalDate`** und ein **`LocalTime`** Objekt übergeben werden:

```
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.NOON;
LocalDateTime dt1 = LocalDateTime.of(date, time); // Heute zu Mittag (12:00)
```

5.7.2 Ändern von Datum und Uhrzeit

Zum Ändern von Datum und Uhrzeit gibt es für die drei Klassen eine Reihe von **`plusXxx()`** und **`minusXxx()`**-Methoden:

`plusDays(long days)`
`minusDays(long days)`
Gibt ein **`LocalDateTime`**- oder ein **`LocalDate`**-Objekt zurück, mit den angegebenen Tagen dazugezählt oder abgezogen

`plusHours(long hours)`
`minusHours(long hours)`
Gibt ein **`LocalDateTime`**- oder ein **`LocalTime`**-Objekt zurück mit den angegbenen Stunden dazugezählt oder abgezogen

`plusMinutes(long minutes)`
`minusMinutes(long minutes)`

Gibt ein **LocalDateTime**- oder ein **LocalTime**-Objekt zurück mit den angegbenen Minuten dazugezählt oder abgezogen

plusMonths(long months)
minusMonths(long months)
Gibt ein **LocalDateTime**- oder ein **LocalDate**-Objekt zurück, mit den angegebenen Monaten dazugezählt oder abgezogen

plusSeconds(long seconds)
minusSeconds(long seconds)
Gibt ein **LocalDateTime**- oder ein **LocalTime**-Objekt zurück mit den angegbenen Sekunden dazugezählt oder abgezogen

plusWeeks(long weeks)
minusWeeks(long weeks)
Gibt ein **LocalDateTime**- oder ein **LocalDate**-Objekt zurück, mit den angegebenen Wochen dazugezählt oder abgezogen

plusYears(long years)
minusYears(long years)
Gibt ein **LocalDateTime**- oder ein **LocalDate**-Objekt zurück, mit den angegebenen Jahren dazugezählt oder abgezogen

Für die Klassen **LocalTime** und **LocalDate** können nur jene Methoden verwendet werden, die die entsprechenden Werte verändern. Zum Beispiel können bei **LocalTime** keine Jahre geändert werden, damit ist die Methode **plusYears()** für diese Klasse nicht geeignet.

Die Methoden können auch verkettet werden, wie im folgenden Beispiel:

```
LocalDate ld = LocalDate.of(2000, Month.JANUARY, 1);
ld = ld.plusYears(2).plusMonths(3).minusDays(1);
System.out.println(ld);
```

Ausgabe:
```
2002-03-31
```

5.7.3 Die Klasse Period

Mit der Klasse **Period** können Zeitabschnitte, bestehend aus Tagen, Monaten und Jahren, festgelegt werden. Dazu werden die **ofXxx()** und **withXxx()**-Methoden verwendet:

static Period of(int years, int months, int days)
Gibt ein **Period**-Objekt, mit den angegebenen Jahren, Monaten und Tagen, zurück

static Period ofDays(int days)
Gibt ein **Period**-Objekt, mit den angegebenen Tagen, zurück

static Period ofMonths(int months)
Gibt ein **Period**-Objekt, mit den angegebenen Monaten, zurück

static Period ofWeeks(int weeks)
Gibt ein **Period**-Objekt, mit den angegebenen Wochen, zurück

static Period ofYears(int years)

Gibt ein **Period**-Objekt, mit den angegebenen Jahren, zurück

Beispiel: Ausgabe aller ersten Tage der Monate des Jahres 2000:

```
LocalDate ld = LocalDate.of(2000, Month.JANUARY, 1);
Period p = Period.ofMonths(1);
do
{
  System.out.println(ld + " " + ld.getDayOfWeek());
  ld = ld.plus(p);
} while (ld.isBefore(LocalDate.of(2001, 1, 1)));
```

Ausgabe:

```
2000-01-01 SATURDAY
2000-02-01 TUESDAY
2000-03-01 WEDNESDAY
2000-04-01 SATURDAY
2000-05-01 MONDAY
2000-06-01 THURSDAY
2000-07-01 SATURDAY
2000-08-01 TUESDAY
2000-09-01 FRIDAY
2000-10-01 SUNDAY
2000-11-01 WEDNESDAY
2000-12-01 FRIDAY
```

Wie das Beispiel zeigt gibt es **plus()** und **minus()**-Methoden der Klassen **LocalDate** und **LocalDateTime** um einen Zeitabschnitt in form eines **Period**-Objekts zum aktuellen Datum dazuzuzählen oder abzuziehen:

plus(Period p)
minus(Perios p)

Gibt ein **LocalDateTime**- oder ein **LocalDate**-Objekt zurück, mit dem angegebenen Zeitabschnitt dazugezählt oder abgezogen

Achtung:
Die **ofXxx()** können zwar verkettet werden, es ist aber nur der jeweils letzte Methodenaufruf gültig. Im folgenden Beispiel besteht der erzeugte Zeitabschnitt nur aus 10 Tagen und nicht aus einem Monat und 10 Tagen:

```
LocalDate ld = LocalDate.of(2000, Month.JANUARY, 1);
Period p = Period.ofMonths(1).ofDays(10);
System.out.println(ld.plus(p));
```

Ausgabe:

```
2000-01-11
```

Verkettungen können hingegen mit den **withXxx()**-Methoden erzeugt werden:

Period withDays(int days)

Gibt ein **Period**-Objekt, mit den angegebenen Tagen hinzugezählt, zurück

Period withMonths(int months)

Gibt ein **Period**-Objekt, mit den angegebenen Monaten hinzugezählt, zurück

Period withYears(int years)

Gibt ein **Period**-Objekt, mit den angegebenen Jahren hinzugezählt, zurück

Somit lässt sich das vorherige Beispiel richtigstellen:

```
LocalDate ld = LocalDate.of(2000, Month.JANUARY, 1);
Period p = Period.ofMonths(1).withDays(10);
System.out.println(ld.plus(p));
```

Ausgabe:

```
2000-02-11
```

5.7.4 Die Klasse `DateTimeFormatter`

Die Klasse **`java.time.format.DateTimeFormatter`** wird dazu verwendet um Datum und/oder Uhrzeit entweder in vordefinierten oder mit selbst-definierten Formaten auszugeben. Objekte dieser Klasse werden ebenfalls nicht mit **`new`** sondern mit eigenen Methoden oder über Konstante erzeugt:

```
DateTimeFormatter dtf;
dtf = DateTimeFormatter.ISO_TIME;       // Default Uhrzeit-Format
System.out.println(dtf.format(LocalTime.now()));

dtf = DateTimeFormatter.ISO_DATE;       // Default Datums-Format
System.out.println(dtf.format(LocalDate.now()));

dtf = DateTimeFormatter.ISO_DATE_TIME;  // Default Datum-Uhrzeit-Format
System.out.println(dtf.format(LocalDateTime.now()));
```

Ausgabe:

```
18:30:20.125
2015-09-23
2015-09-23T18:30:20.975
```

Die gleiche Ausgabe erhält man, wenn die **`toString()`**-Methode der jeweiligen Klasse aufrufen wird:

```
System.out.println(LocalTime.now().toString());
System.out.println(LocalDate.now().toString());
System.out.println(LocalDateTime.now().toString());
```

Eine weitere Möglichkeit den **`DateTimeFormatter`** zu konfigurieren, ist die Verwendung von Konstanten aus der Enum **`FormatStyle`**:

```
DateTimeFormatter dtf;
dtf = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(dtf.format(LocalTime.now()));

dtf = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);
System.out.println(dtf.format(LocalDate.now()));

dtf = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL,
                                            FormatStyle.MEDIUM);
System.out.println(dtf.format(LocalDateTime.now()));
```

Ausgabe:

```
18:00
23.09.2015
Mittwoch, 23. September 2015 18:00:00
```

Die Konstanten **`LONG`** und **`FULL`** dürfen nur für ein Datum verwendet werden.

Um die Ausgabe nach eigenen Vorstellungen zu konfigurieren wird die Methode **`ofPattern()`** verwendet:

```
DateTimeFormatter dtf;
dtf = DateTimeFormatter.ofPattern("dd-MM-yyyy  HH:mm:ss");
System.out.println(dtf.format(LocalDateTime.now()));

dtf = DateTimeFormatter.ofPattern("EE dd-MMM-yyyy  hh:mm:ss");
System.out.println(dtf.format(LocalDateTime.now()));

dtf = DateTimeFormatter.ofPattern("EEEE dd. MMMM yyyy - ww");
System.out.println(dtf.format(LocalDateTime.now())+". Woche");
```

Ausgabe:

```
23-09-2015  18:30:00
Mi 23-Sep-2015  06:30:00
Mittwoch 23. September 2015 - 39. Woche
```

Folgende Buchstaben können zur Formatierung verwendet werden:

Symbol	***Inhalt (DateTime-Component)***	***Wert***
Y (KB!)	Jahr (Year)	**2010; 10**
M	Monat (Month)	**Oktober; Okt; 10**
L	Monat (Month)	**Oktober; Okt;**
Q/q	Quartal	**3; Q3; 3. Quartal**
W	Woche im Monat (Week in month)	**2**
w	Woche im Jahr (Week in year)	**39**
D	Tag des Jahres (Day in Year)	**299**
d	Tag des Monats (Day in month)	**26**
E	Wochentag (Day in week)	**Mittwoch; Mi**
H	Stunde (Hour in day) 0-23	**0**
h	Stunde (Hour in am/pm) 1-12	**12**
a	am-pm	**AM**
m	Minute (Minute in hour)	**30**
s	Sekunde (Second in minute)	**55**
S	Millisekunde (Millisecond)	**123**
n	Nanosekunden (Nanonseconds)	**987654321**

Es gibt noch einige weitere reservierte Buchstaben, z.B. zum Anzeigen der Zeitzone für Objekte der Klasse **`ZonedDateTime`**. Nicht reservierte Buchstaben lösen eine Exception aus.

Die Klasse **`DateTimeFormatter`** kann auch verwendet werden um ein Datum/eine Uhrzeit zu parsen, d.h. aus einem String ein **`LocalTime`**-, **`LocalDate`**- oder **`LocalDateTime`**-Objekt zu erzeugen:

```
DateTimeFormatter dtf;
dtf = DateTimeFormatter.ofPattern("HH:mm:ss");
LocalTime lt = LocalTime.parse("12:21:12", dtf);

dtf = DateTimeFormatter.ofPattern("dd.MM.yyyy");
LocalDate ld = LocalDate.parse("01.01.2015", dtf);

dtf = DateTimeFormatter.ofPattern("EEEE dd. MMMM yyyy HH:mm");
LocalDateTime ldt =
          LocalDateTime.parse("Mittwoch 23. September 2015 12:30", dtf);
```

5.8 Die Klasse `Arrays`

Für die Manipulation von Feldern gibt es im package **`java.util`** die Klasse **`Arrays`** mit einer Reihe von nützlichen, statischen Methoden, die für Felder beliebigen Datentyps verwendet werden können:

`public static String toString(Object[] a)`

Gibt einen String mit dem Inhalt des Feldes zurück.

Beispiel:

```
int[] feld = {3,9,1,6,4};
System.out.println(Arrays.toString(feld));
```

Ausgabe:

```
[3, 9, 1, 6, 4]
```

`public static String deepToString(Object[] a)`

Gibt einen String mit dem tiefen Inhalt des Feldes zurück, d.h. für jedes Element des Feldes wird wiederum die **`toString()`**-Methode angewendet. Die Methode kann z.B. für die Ausgabe mehrdimensionaler Felder verwendet werden.

Beispiel:

```
int[][] feld = new int[2][3];
int wert = 1;
for (int i = 0; i < feld.length; i++)
{
  for (int j = 0; j < feld[i].length; j++)
  {
    feld[i][j] = wert++;
  }
}
System.out.println(Arrays.deepToString(feld));
```

Ausgabe:

```
[[1, 2, 3], [4, 5, 6]]
```

`public static void fill(Object[] a, Object value)`

Füllt das gesamte Feld mit einem Wert.

Beispiel:

```
int[] feld = new int[10];
Arrays.fill(feld,2);
System.out.println(Arrays.toString(feld));
```

Ausgabe:

```
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

`public static boolean equals(Object[] a1, Object [] a2)`

Gibt **`true`** zurück wenn alle Elemente der beiden Felder in Inhalt und Reihenfolge gleich sind.

Beispiel:

```
int[] f1 = {1,1,1};
int[] f2 = new int[3];
Arrays.fill(f2,1);
System.out.println(Arrays.equals(f1, f2));
```

Ausgabe:

```
true
```

public static Object[] copyOf(Object[] a, int newLength)

Gibt eine Kopie des Feldes mit der Länge **newLength** zurück, wobei das neu erzeugte Feld entweder abgeschnitten oder mit Nullen aufgefüllt wird.

Beispiel:

```
int[] feld = {3,9,1,6,4};
int[] f2 = Arrays.copyOf(feld, 2);
int[] f3 = Arrays.copyOf(feld, 8);
System.out.println(Arrays.toString(feld));
System.out.println(Arrays.toString(f2));
System.out.println(Arrays.toString(f3));
```

Ausgabe:

```
[3, 9, 1, 6, 4]
[3, 9]
[3, 9, 1, 6, 4, 0, 0, 0]
```

public static Object[] copyOfRange(Object[] a, int from, int to)

Gibt eine Kopie des Feldes aus dem definierten Bereich, inklusive Startindex aber exklusive Endindex, zurück.

Beispiel:

```
int[] feld = { 3, 9, 1, 6, 4 };
int[] f2 = Arrays.copyOfRange(feld, 1, 3);
System.out.println(Arrays.toString(feld));
System.out.println(Arrays.toString(f2));
```

Ausgabe:

```
[3, 9, 1, 6, 4]
[9, 1]
```

public static void sort(Object[] a)

Sortiert das Feld in aufsteigender Reihenfolge.

Beispiel:

```
s
 Arrays.sort(feld);
 System.out.println(Arrays.toString(feld));
```

Ausgabe:

```
[1, 3, 4, 6, 9]
```

public static int binarySearch(Object[] a, Objetc value)

Liefert den Index von **value** im Feld zurück. Das Feld **muss** sortiert sein, sonst ist der Rückgabewert unbestimmt. Ist **value** nicht im Feld, so wird der um 1 verschobene negative Index der Position zurückgegeben, an der **value** eingefügt werden müsste.

Beispiel:

```
int[] feld = {3,9,1,6,4};
Arrays.sort(feld);      // 1, 3, 4, 6, 9
System.out.println(Arrays.binarySearch(feld, 6));
System.out.println(Arrays.binarySearch(feld, 5));
```

Ausgabe:

```
3
-4
```

Das Element '6' befindet sich am Index 3. Das Element '5' befindet sich nicht im Feld, müßte aber an Position 3 eingefügt werden. Der um 1 verschobene negative Index ist daher -4.

Quellenverzeichnis

[1] ***OCA Oracle Certified Associate Java SE 8 Programmer***, Jeanne Boyarsky & Scott Selikoff, Sybex-Verlag, 2015

[2] ***Java 8 – Die Neuerungen***, Michael Inden, dpunkt.verlag, 2014

[3] ***Handbuch der Java-Programmierung***, 7. Auflage, Guido Krüger & Thomas Stark, Addison-Wesley-Verlag, 2011

[4] ***Java Grundlagen und objektorientierte Programmierung***, Mag. Otto Reichel, HTL St. Pölten, 2008

[5] ***Sun Certified Programmer for Java 6***, Kathy Sierra & Bert Bates, McGraw-Hill-Verlag, 2008

[6] ***Java ist auch eine Insel***, 8.Auflage, Christian Ullenboom, Galileo Computing, 2009