Programmieren mit Java

Eine grundlegende Einführung in das Objektorientierte Programmieren mit Java

Marcus Dramburg
Programmieren 2
Hamburg, 20 November 2017

HELMHOLTZ GEMEINSCHAFT

DESY

Überblick

- Historie
- Nützliches Vorwissen
- Installation
- Die Sprache Java
  - Syntax
  - Primitive Datentypen
  - Arrays / Strings
  - Klassen und Objekte
  - Collections

Historie - Entstehung

- 1991 – 1992 ensteht Java unter dem Namen OAK (Object Application Kernel) im Auftrag von Sun Microsystems. Hauptentwickler war James Gosling.
- ab 1995 Veröffentlichung unter dem Namen Java als Alpha 1.0a2
- 1996 offizielle Veröffentlichung der Version 1.0
- ab 2006 Veröffentlichung des Quellcodes unter der GPL
- 2010 Sun Microsystems und Java gehen in den Besitz von Oracle über

DESY

Historie – Java-Versionen

Jahr	Version	Klassen/ Interfaces*	Neuheiten
1996	JDK 1.0	212	I/O, Net, AWT, Applets
1997	JDK 1.1	504	Innere Klassen, Event Listener
1998	JDK 1.2, Java 2	1520	Collections, Swing, Drag&Drop
2000	JDK 1.3	1842	HotSpot Engine, Sound API
2002	JDK 1.4	2991	Assertions
2004	J2SE 5.0 (1.5)	3279	Generics, Autoboxing, Enumerations, static import, for-each-Schleife
2006	JSE 6.0 (1.6)	3793	Einbettung von Skriptsprachen
2011	JSE 7.0 (1.7)	4024	New I/O API
2014	JSE 8.0 (1.8)	4240	Lambda-Ausdrücke, New Date API, Methodenimplementierung in Interfaces (Default Interface)

* ab Version 5.0 auch Enumerations

DESY

Java - Besonderheiten

- Java ist sowohl eine kompilierte, als auch interpretierte Sprache
- Der Compiler erzeugt Bytecode als maschinenunabhängigen Zwischencode
- Der Interpreter führt den Bytecode aus ("write once, run anywhere") und zwar in einer virtuellen Stackmaschine, der JVM (Java Virtual Machine) – im Gegensatz dazu ist die VM in Android (DVM, Dalvik) eine Registermaschine (näher an echten Prozessoren)
- Mit der Einführung von Lambda-Ausdrücken aus der funktionalen Programmierung wandelt sich der objektorientierte Ansatz von Java langsam in einen Multi-Paradigmen Ansatz

Java - Wissenswertes

- JavaScript hat NIX mit Java zu tun! JavaScript (eigentlich ECMA-Script) wollte nur auf dem Java-Hype der End-90er mitsurfen, also reines Marketing!
- Es gibt freie Java-Versionen (Standard unter Linux ist openjdk) und es gibt native Java-Compiler, die Java-Code in Maschinencode übersetzen, beispielsweise die gcc-Erweiterung gcj

Vorwissen: Programmier-Paradigmen

- Paradigma = Weltbild, Weltanschauung, Lehrmeinung
- Ersetzung eines Paradigmas durch ein anderes heißt Paradigmenwechsel
- Ein Programmierparadigma ist ein fundamentaler Programmierstil und meint die Summe der einer Programmiersprache zugrunde liegenden Konzepte der Darstellung und Verwirklichung von statischen Elementen (Variablen, Funktionen, Methoden, Klassen, etc.) und dynamischen Elementen (Operationen, Kontrollfluß, Nebenläufigkeit).
- Nach 4 Jahrzenten eifrigen Verfechtens "reiner Programmierparadigmen" setzen sich in den letzten Jahren vermehrt Sprachen mit sogenanntem Multi-Paradigmen Ansatz durch.

Vorwissen: Programmier-Paradigmen (Beispiele)

- Imperative Programmierung (Programm ist sequentielle Befehlsabfolge, Assembler) -> Strukturierte Programmierung (keine unbedingten Sprünge, Algol) -> Prozedurale Programmierung (Kapselung von Funktionalität in Prozeduren, Daten separiert, Pascal, C)
- Deklarative Programmierung (Programm ist Beschreibung der gesuchten Lösung, nicht eine Implementierung des Lösungsweges)
  - Funktionale Programmierung (LISP)
  - Logische Programmierung (Prolog)
  - Mengenbasierte Abfrageformulierung (SQL)
- Objektorientierte Programmierung
  - Klassen als instanzierbare (Objekte) Grundstruktur in denen Daten zusammen mit den auf ihnen arbeitenden Routinen (Methoden) gehalten werden
  - Gegensatz zur Prozeduralen Programmierung

Vorwissen: Objektorientierung

- Grundstrukturen sind Klassen und Methoden
  - Klassen definieren die Daten (Klassen- und Instanzvariablen) und Funktionalitäten (Methoden) von aus ihnen erzeugten Instanzen (Objekte) und die Art und Weise, wie diese Instanzen erzeugt (Konstruktor) und wieder zerstört (Destruktor) werden.
  - Methoden legen die mögliche abrufbare Funktionalität eines Objekts fest
- Dabei kann man sich Klassen wie eine Ausstechform für Kekse vorstellen, z.B. eine sternförmige Form (Klasse) mit der konkrete Kekse (Objekte/Instanzen) ausgestochen werden. Die ausgestochenen Kekse lassen sich Verzieren, z.B. mit verschiedenen Nüssen. Da jeder Keks anders verziert wird, betrifft dies nur die konkrete Instanz, man hätte also eine Instanzvariable. Sollen nun alle Kekse mit Schokolade überzogen werden hätte man eine Klassenvariable vorliegen, da alle Instanzen dieser Klasse betroffen sind. Als Methode des Keks könnte man sich z.B. aufessen(Zeit t) vorstellen (Achtung Vergleich hinkt ;-):
  - SternFormKeks keksMitMandel = neuer Keks(Mandel)
    keksMitMandel.aufessen(SOFORT)

Java – Download und API Documentation

- JDK Java – Java SE (Standard Edition)
  aktuell Java SE 8uXXX
  - www.oracle.com/technetwork/java/javase/downloads/index.html
- Java Development Kit (JDK)
- Sehr interessant sind die JDK Demos and Samples
- JDK with NetBeans als Bundle
- Java SE 8 Documentation – ganz wichtig!!!
  - vollständige Beschreibung aller vorhandenen Java Klassen und ihrer Methoden
- Für Verschlüsselungaufgaben (z.B. programmatische https-Abfragen) findet sich dort noch die Java Cryptography Extension (JCE)

Java – Installation unter Linux

- Vorinstalliert ist heutzutage meistens OpenJDK
  - Nicht kompatibel zum Standard Oracle Java!
  - Nett für privaten Krempel / Open Source Projekte
  - Nicht im professionellen Bereich zu gebrauchen
- Installation via Paketmanager unter Ubuntu 16.04
  - $ sudo add-apt-repository ppa:webupd8team/java
    $ sudo apt-get update
    $ sudo apt-get install oracle-java8-installer
    $ java -version # Test auf richtige Version
- Lokale Installation mit tar-archive
  - Archive runterladen, irgendwo im eigenen home-directory auspacken
  - JAVA_HOME Variable auf die Java Installation setzen
  - /MYHOME/$JAVA_HOME/bin in die PATH-Variable aufnehmen (1.Stelle)
  - Verzeichnis für Java-Code anlegen und den Pfad in die CLASSPATH-Variable aufnehmen

DESY

Java – Installation unter Windows

- Installer für Windows herunterladen
  - Achtung kann Crap wie Browsersuchleisten mitbringen! Aufmerksamkeit!
- Ordner für den eigenen Java Sourcecode anlegen
- Nach Installation die entsprechenden Variablen anlegen bzw. setzen:
  - JAVA_HOME zeigt auf das Java Installationsverzeichnis
  - Path Variable den Pfad zu %JAVA_HOME%\bin hinzufügen
  - CLASSPATH Variable auf den angelegten Sourcecode Ordner setzen

Java – Installation unter MacOS

- Bis Java 6.0 hat Apple die Javaversion selbst gepflegt, dann eingestellt
- Bei Apple gibt es eine aktualisierte 6er Version (zu alt)
- Ab Java 7.0 pflegt Oracle die Java Version für den Mac
- Download des Installers für OS X, auspacken, Installer starten und den Anweisungen folgen
- Java Sourcecode Directory anlegen
- Dann ggf. noch die Umgebungsvariablen JAVA_HOME, PATH und CLASSPATH setzen bzw. anlegen

Java - Umgebungsvariablen

- JAVA_HOME
  - Zeigt immer auf das Verzeichnis in dem sich die Java Installation befindet
  - Linux/OS X: setzen in der $HOME/.bashrc oder $HOME/.profile mit
    export JAVA_HOME="/path/to/java-install-dir"
  - Windows: System -> Advanced system settings -> Environment Variables
    Unter User variables -> new -> Variable name: JAVA_HOME und Variable value:
    \path\to\java-install-dir
- PATH
  - Zeigt immer auf das bin-Verzeichnis im Java Installations Verzeichnis
  - Linux/OS X: .bashrc oder .profile
    export PATH="$JAVA_HOME/bin:$PATH"
  - Windows: System ... Environment Variables -> Unter System variables suchen, auswählen, Edit... %JAVA_HOME%\bin; hinzufügen

Java - Umgebungsvariablen

> CLASSPATH

- Muß nicht unbedingt gesetzt werden, kann dem Interpreter auch als Parameter übergeben werden, Parameter überschreibt die gesetzte Variable
- Zeigt auf ein beliebiges Verzeichnis unterhalb dessen der Sourcecode bzw. die Kompilate in einer Ordnerstruktur befinden, die den package Namen widerspiegelt Beispiel: Eine Klasse **Example** befindet sich in der entsprechenden Sourcecode Datei namens **Example.java**. In der Datei befindet sich die package Anweisung **package de.desy.testcases;** ist nun die CLASSPATH Variable auf /home/user/java gesetzt muß sich das Kompilat der Datei Example.java Example.class im Pfad /home/user/java/de/desy/testcases/Example.class befinden, damit sie vom Java Interpreter gefunden wird
- Linux/OS X: .bashrc oder .profile
  export CLASSPATH="/path/to/my/java-code-repository:$CLASSPATH"
- Windows: System ... Environment Variables als User variable -> new mit name: CLASSPATH und value: \path\to\my\java-code/repository

Java - Entwicklungsumgebungen

- IntelliJ IDEA
  - Kommerzielle IDE, Community Edition kostenfrei, Ultimate Edition für professionelle Entwicklung, Basis des Android Developer Studio von Google, Ableger für verschiedene Sprachen verfügbar, Windows, Linux, OS X
  - https://www.jetbrains.com/idea/#chooseYourEdition
- Eclipse
  - Basis mal von IBM entwickelt, heute OpenSource, Name war Kriegserklärung an Sun (Ekliptik ist der Bereich am Horizont hinter dem die Sonne untergeht), Unterstützung für viele verschiedene Sprachen über plugins verfügbar (Achtung: nicht alle plugins von gleicher Qualität und viele plugins können eclipse unbenutzbar machen), Windows, Linux, OS X
  - http://www.eclipse.org/die/

Java - Entwicklungsumgebungen

- Netbeans
  - Begann mal als Studentenprojekt in der Tschechischen Republik, wurde dann die Firma Netbeans, die 1999 von Sun übernommen wurde, heute OpenSource, war mal eine reine Java IDE, heute Support für viele Sprachen, ist komplett in Java geschrieben, deshalb prinzipiell überall lauffähig wo eine Standard-JVM läuft
  - https://netbeans.org/downloads/
- Darüber hinaus existieren noch ein paar kommerzielle Nischenprodukte ohne weite Verbreitung, wie der Rational Application Developer von IBM und der Jbuilder von Borland (mittlerweile nur noch ein Eclipse plugin)

Java – Texteditor und Kommandozeile

- Für das Arbeiten mit Texteditor und Kommandozeile müssen auf jeden Fall die Java Umgebungsvariablen richtig gesetzt sein
- Prinzipiell geht jeder Texteditor, sollte aber schon Syntax-Highlighting für Java zur Verfügung stellen
- Um eine Datei Example.java zu kompilieren und auszuführen müssen nacheinander der Compiler javac und der Interpreter java bemüht werden:
  - $ cd /to/where/java-files/are
    $ javac Example.java
    $java de.desy.testcases.Example # Aufruf mit vollem Namen!

Die Sprache Java

- In weiten Teilen syntaktische Ähnlichkeit zu C
- Komplexe Sprache, Umsetzung vieler Konzepte aus dem "modernen Softwareengineering", wie Design Pattern, MVC
- Imperativ, Objektorientiert
- Lauffähigkeit abhängig vom Vorhandensein einer JVM bzw. nativen Compilers
  - JVM für Windows, OS X, Linux, Solaris
  - Native Compiler (Nischendasein)
  - Android (nicht Bytecodekompatibel, andere VM)
- Wenig Freiheitsgrade
  - Enge Vorgaben durch Sprache
  - Kein Direktzugriff auf Speicherbereiche
  - Viele Automatismen (Garbage Collection)
  - Sprache für große Projekte

DESY

Java - Programmelemente

- package-Zuweisung
- import-Anweisung(en)
- Klassendeklaration
- Methodendeklaration
- Anweisungen
- Variablen
  - Klassenvariablen
  - Instanzvariablen
  - Lokale Variablen

DESY

Java – Compiler, Interpreter, JVM und Bytecode

- Der Java Compiler $JAVA_HOME/bin/javac übersetzt den Java Source Code, der ihm übergeben wird in Java Byte Code (quasi Maschinencode für die JVM). Aus *.java Dateien werden *.class Dateien
- Der Java Interpreter $JAVA_HOME/bin/java lädt den Bytecode in die JVM und und erzeugt zur Laufzeit den Maschinencode für die Architektur auf der die JVM läuft.
- JVM – Java Virtuelle Maschine ist die Implementierung eines Stack-Automaten in den der Bytecode geladen und durch den Interpreter abgearbeitet wird.
- Bytecode ist der vom Java Compiler erzeugte Zwischencode, der lauffähig auf allen JVMs gleicher Bauart ist, unabhängig von der darunterliegenden Architektur.

Java – Hello World

```java
/*
 * a multiline comment
 */
package examples.helloworld;

public class HelloWorld{
    // a line comment
    public static void main(String[] args){
        System.out.println("Hello world!");
    }
}
```

```
[:-)] [16:09:48]: /Users/dramburg/Code/Java/examples/helloworld
--> $ echo $CLASSPATH
/Users/dramburg/Code/Java
[:-)] [16:10:15]: /Users/dramburg/Code/Java/examples/helloworld
--> $ javac HelloWorld.java
[:-)] [16:10:24]: /Users/dramburg/Code/Java/examples/helloworld
--> $ java HelloWorld
Error: Could not find or load main class HelloWorld
[:-(] [16:10:37]: /Users/dramburg/Code/Java/examples/helloworld
--> $ java examples.helloworld.HelloWorld
Hello world!
[:-)] [16:11:02]: /Users/dramburg/Code/Java/examples/helloworld
--> $
```


DESY

Java – Hello World, Besprechung

- /* und */ leiten einen mehrzeiligen Kommentar ein bzw. beenden ihn
- package examples.helloword;
  - Java Code ist in packages organisiert. Fehlt die package-Anweisung wird der Code automatisch dem default package zugeordnet. Da die Zugriffsmodifikatoren unter anderem auch die Sichtbarkeit auf ein package eingrenzen können sollte immer bewußt ein package für den eigenen Programmcode angelegt und genutzt werden. Allgemeine Konvention ist, daß package-Namen immer vollständig klein geschrieben werden. In packages können sowohl Code-Dateien liegen, als auch weitere packages. Die einzelnen Komponenten entsprechen im Dateisystem Verzeichnissen, ersetzt man in einem package-Namen den Punkt durch den Pfadtrenner des benutzten Betriebssystems erhält man den Pfad im Dateisystem unterhalb eines Verzeichnisses in der CLASSPATH-Variable.
  - Anweisungen werden in Java mit einem Semikolon abgeschlossen.

Java – Hello World, Besprechung

- public class HelloWorld{...}
  - public ist einer von mehreren möglichen Zugriffsmodifikatoren. Diese regeln die Sichtbarkeit (und damit auch die Zugriffsmöglichkeit) von Klassen, Methoden, Variablen und Blöcken ein. Dabei bezeichnet public die Sichtbarkeit von überall aus.
  - Mit der class Anweisung wird eine Klasse erzeugt (im Prinzip wird ein Code-Block eröffnet mit { und }) in der dann Daten und die auf ihnen arbeitenden Methoden ihren Platz finden. In Java kann kein Code außerhalb einer Klassenstruktur existieren.
  - HelloWorld ist der Name der Klasse, hierbei ist Konvention, daß Klassennamen mit einem Großbuchstaben beginnen und jedes weiter Wort wiederum mit einem Großbuchstaben (CamelCase). Sehr streng ist Java, was die Benennung der Dateien angeht in denen eine Klasse implementiert. Die Datei muß heißen wie die Klasse plus der Endung java (hier also: HelloWorld.java). Darüber hinaus muß jede Klasse in einer eigenen Datei liegen. Selbes gilt auch für Interfaces. Eine Ausnahme was Benennung und Anzahl pro Datei angeht sind anonyme und innere Klassen.
  - Implizit wird jede Klasse in Java von der Klasse Object abgeleitet und erbt von dieser ein paar Eigenschaften. Object befindet sich mit einigen weiteren Klassen, die für Sprache von fundamentaler Bedeutung sind im package java.lang.
- // beginnt einen einzeiligen Kommentar, der bis zum Zeilenende geht

DESY

Java – Hello World, Besprechung

- public static void main(String[ ] args){...}
  - Dies ist die Signatur der main-Methode in Java. Wie auch in C ist dies die Einsprungstelle in das Programm.
  - public ist der Zugriffsmodifikator, die main-Methode muß public sein.
  - Das Schlüsselwort static bezeichnet einen Aspekt der Speicherverwaltung in Bezug auf den Zeitpunkt des Ladens und Verfügbarmachens innerhalb des Programms. Im Prinzip meint static, daß das so ausgezeichnete Element ohne Erzeugung eines Objekts vorhanden und zugreifbar ist. So ausgezeichnete Methoden nennt man auch **Klassenmethoden**, static Variablen nennt man auch **Klassenvariablen** und mit static ausgewiesene Codeblöcke nennt man **statischen Initialisierungsblock**, statische Klassen sind nur als innere Klassen möglich und heißen dann **static nested class**. Um also die main-Methode aufzurufen muß nicht erst ein Objekt der Klasse HelloWorld erzeugt werden.
  - Das Schlüsselwort void gibt hier an das die Methode keinen Rückgabewert liefert.
  - args ist ein Array der Strings hält. Hier sind dies die dem Programm beim Start übergebenen Argumente.

Java – HelloWorld, Besprechung

- System.out.println("Hello World");
  - **System** ist eine Klasse aus dem package java.lang. Dieses package ist immer im Programmcode verfügbar auch ohne explizit per import-Anweisung verfügbar gemacht zu werden. System stellt die Java-Schnittstelle zu grundlegenden Funktionen des unterliegenden Betriebssystems zur Verfügung.
  - **out** ist ein statisches Handle auf den Standardausgabekanal des Betriebssystems und vom Typ PrintStream. Darüber hinaus verfügt System noch über zwei weitere statische Handles auf Kommunikationskanäle, einmal **err** für der Standarderrorkanal, ebenfalls vom Typ PrintStream und **in** für den Standardeingabekanal vom Typ InputStream.
  - **println()** und **print()** sind zwei Methoden der Klasse PrintStream, die mehrfach überladen (gleicher Rückgabewert, gleicher Methodenname, andere Parameterliste) vorliegen um alle Typen in Java ausgeben zu können, hier einen String. Die beiden Methoden unterscheiden sich darin, daß die erste einen Zeilenumbruch an die Ausgabe hängt und die zweite nur das Argument ausgibt.

Java – Primitive Datentypen

- byte – Ganzzahltyp, 8 Bit, Wertebereich von -128 bis 127
- char – Zahlwert repräsentiert Unicode Zeichen, 16 Bit, Wertebereich von 0 bis $2^{16} - 1$
- short – Ganzzahltyp, 16 Bit, Wertebereich von 2^{-15} bis $2^{15} - 1$
- int – Ganzzahltyp, 32 Bit, Wertebereich von 2^{-31} bis $2^{31} - 1$
- long – Ganzzahltyp, 64 Bit, Wertebereich von 2^{-63} bis $2^{63} -1$
- float – Fließkommazahl, 32 Bit, Wertebereich von $+/-1,4 * 10^{-45}$ bis $+/-3,4 * 10^{38}$
- double – Fließkommazahl, 64 Bit, Wertebereich von $+/-4,9 * 10^{-324}$ bis $+/-1,7 * 10^{308}$
- boolean – Wahrheitswert, Wertebereich ist false, true

Java – Primitive Datentypen, Fließkomma-Arithmetik

```java
/*
 * FloatingPointFailures.java
 */

package examples.primitives;

public class FloatingPointFailures{

    public static void main(String[] args){
        // first try with double
        double aDouble = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
        if (aDouble == ((1.0/3.0) + (1.0/3.0) + (1.0/3.0))){
            System.out.println("adding up 10 times 0.1 is EQUAL to add 3 times 1/3");
        }else{
            System.out.println("adding up 10 times 0.1 is NOT EQUAL to add 3 times 1/3");
        }
        // second try with float
        float aFloat = 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f + 0.1f;
        if (aFloat == 0.1f * 10.0f){
            System.out.println("adding up 10 times 0.1 IS EQUAL to ten times 0.1");
        }else{
            System.out.println("adding up 10 times 0.1 IS NOT EQUAL to ten times 0.1");
        }
        System.exit(0);
    }
}
```

```
[:-)] [11:27:35]: /Users/dramburg/Code/Java/examples/primitives
--> $ javac FloatingPointFailures.java
[:-)] [11:30:00]: /Users/dramburg/Code/Java/examples/primitives
--> $ java examples.primitives.FloatingPointFailures
adding up 10 times 0.1 is NOT EQUAL to add 3 times 1/3
adding up 10 times 0.1 IS NOT EQUAL to ten times 0.1
```


DESY

Java – Primitive Datentypen, Default Initialisierung

- Der Default Ganzzahltyp ist int, also 3 + 4 ist die Addition zweier Integer, somit auch das Ergebnis long l = 3 + 4; geht nur mit cast
- Der Default Fließkommatyp ist double, 3.0 + 4.0 ist die Addition zweier Double Werte
- byte – 0
- short – 0
- int – 0
- long – 0L
- float – 0.0f
- double – 0.0d
- char – '\u0000'
- boolean - false

Java – Primitive Datentypen, Beispiel 1

```
/*
 * PrimitiveDefaultValues.java
 */
package examples.primitives;

public class PrimitiveDefaultValues{

    public static void main(String[] args){
        byte aByte;
        boolean aBool;
        char aChar;
        float aFloat;
        double aDouble;
        int anInt;
        long aLong;
        short aShort;
        System.out.println("aByte:   " + aByte);
        System.out.println("aBool:   " + aBool);
        System.out.println("aChar:   " + aChar);
        System.out.println("aFloat:  " + aFloat);
        System.out.println("aDouble: " + aDouble);
        System.out.println("anInt:   " + anInt);
        System.out.println("aLong:   " + aLong);
        System.out.println("aShort:  " + aShort);
        System.exit(0);
    }
}
```


DESY

Java – Primitive Datentypen, Beispiel 1 Ausgabe

```
[:-)] [14:33:57]: /Users/dramburg/Code/Java/examples/primitives
--> $ javac PrimitiveDefaultValues.java
PrimitiveDefaultValues.java:17: error: variable aByte might not have been initialized
        System.out.println("aByte:   " + aByte);
                                       ^
PrimitiveDefaultValues.java:18: error: variable aBool might not have been initialized
        System.out.println("aBool:   " + aBool);
                                       ^
PrimitiveDefaultValues.java:19: error: variable aChar might not have been initialized
        System.out.println("aChar:   " + aChar);
                                       ^
PrimitiveDefaultValues.java:20: error: variable aFloat might not have been initialized
        System.out.println("aFloat:  " + aFloat);
                                       ^
PrimitiveDefaultValues.java:21: error: variable aDouble might not have been initialized
        System.out.println("aDouble: " + aDouble);
                                       ^
PrimitiveDefaultValues.java:22: error: variable anInt might not have been initialized
        System.out.println("anInt:   " + anInt);
                                       ^
PrimitiveDefaultValues.java:23: error: variable aLong might not have been initialized
        System.out.println("aLong:   " + aLong);
                                       ^
PrimitiveDefaultValues.java:24: error: variable aShort might not have been initialized
        System.out.println("aShort:  " + aShort);
                                       ^
8 errors
```

Java – Primitive Datentypen, Beispiel 2

```java
/*
 * PrimitiveDefaultValues2.java
 */
package examples.primitives;

public class PrimitiveDefaultValues2{

    private byte aByte;
    private boolean aBool;
    private char aChar;
    private float aFloat;
    private double aDouble;
    private int anInt;
    private long aLong;
    private short aShort;

    public static void main(String[] args){
        PrimitiveDefaultValues2 pdv2 = new PrimitiveDefaultValues2();
        System.out.println("aByte:   " + pdv2.aByte);
        System.out.println("aBool:   " + pdv2.aBool);
        System.out.println("aChar:   " + pdv2.aChar);
        System.out.println("aFloat:  " + pdv2.aFloat);
        System.out.println("aDouble: " + pdv2.aDouble);
        System.out.println("anInt:   " + pdv2.anInt);
        System.out.println("aLong:   " + pdv2.aLong);
        System.out.println("aShort:  " + pdv2.aShort);
        System.exit(0);
    }
}
```


DESY

Java – Primitive Datentypen, Beispiel 2 Ausgabe

```
[:-)] [15:01:49]: /Users/dramburg/Code/Java/examples/primitives
--> $ javac PrimitiveDefaultValues2.java
[:-)] [15:02:20]: /Users/dramburg/Code/Java/examples/primitives
--> $ java examples.primitives.PrimitiveDefaultValues2
aByte:   0
aBool:   false
aChar:
aFloat:  0.0
aDouble: 0.0
anInt:   0
aLong:   0
aShort:  0
[:-)] [15:02:27]: /Users/dramburg/Code/Java/examples/primitives
--> $
```


DESY

Java – Primitive Datentypen, Wrapper Klassen

- Zu allen primitiven Datentypen gibt es eine Wrapper-Klasse, die Objektrepräsentationen zum jeweiligen Datentyp zur Verfügung stellen, dies ist z.B. nötig wenn Primitive Datentzpen in Collections gespeichert werden sollen (Collections nehmen nur Objekte auf).
- Die Wrapper liegen alle im package java.lang und heißen Boolean, Byte, Character, Float, Double, Integer, Long und Short.
- Es gibt jeweils einen Konstruktor, der einen Wert des jeweiligen Typs erwartet oder eine Stringrepräsentation des Typs
- Viele Methoden (Klassen- und Instanzmethoden) zur Umwandlung und Darstellung von Primitiven
- Für die Ganzzahlen ist mit Hilfe der Wrapperklassen seit Java 8 auch eine Darstellung als unsigned Wert möglich um den Wertebereich zu vergrößern.

Java – Wrapper-Klassen, Beispiele

```java
/*
 * PrimitiveWrappers.java
 */
package examples.primitives;

public class PrimitiveWrappers{

    public static void main(String[] args){

        char c = 'a';
        int i = -1;
        boolean b = true;
        double d = 3.1;
        Character aCharacter = new Character(c);
        Character anotherCharacter = new Character('b');
        Integer anInteger = new Integer(i);
        Double aDouble = new Double(d);

        System.out.println("c: " + c + ", aCharacter: " + aCharacter);
        System.out.println("Character('c').equals(Character('C')) = "
            + aCharacter.equals(Character.valueOf('C')));
        System.out.println("aCharacter.equals(anotherCharacter) = "
            + aCharacter.equals(anotherCharacter));
        System.out.println("value of anInteger: " + anInteger.intValue());
        System.out.println("-1 parsed as int: " + Integer.parseInt("-1"));
        System.out.println("aDouble as integer: " + aDouble.intValue());
        System.out.println("Max(3.1, 3.2): " + Double.max(3.1, 3.2));

        System.exit(0);
    }
}
```


DESY

Java – Wrapper-Klassen, Beispiele Ausgabe

```
[:-)] [17:18:47]: /Users/dramburg/Code/Java/examples/primitives
--> $ javac PrimitiveWrappers.java
[:-)] [17:18:54]: /Users/dramburg/Code/Java/examples/primitives
--> $ java examples.primitives.PrimitiveWrappers
c: a, aCharacter: a
Character('c').equals(Character('C')) = false
aCharacter.equals(anotherCharacter) = false
value of anInteger: -1
-1 parsed as int: -1
aDouble as integer: 3
Max(3.1, 3.2): 3.2
[:-)] [17:18:59]: /Users/dramburg/Code/Java/examples/primitives
--> $
```


DESY

Java – Referenztypen und call-by value/reference

- Alle nicht-primitiven Datentypen (also alle Objekte) sind Referenztypen.
- Variablenbezeichner enthält einen Zeiger (Speicheradresse) auf das Objekt.
- Im Gegensatz zu C sind die Adressen nicht direkt manipulierbar.
- Primitive Datentypen werden immer per value übergeben
- Objekte werden immer referenziert, also per reference übergeben

Java – Arithmetische Operatoren

Operator	Beschreibung	Operator-Typ
+	Positives Vorzeichen	unär
-	Negatives Vorzeichen	unär
+	Addition	binär
-	Subtraktion	binär
*	Multiplikation	binär
/	Division	binär
%	Modulo	binär
++	Inkrement	unär
--	Dekrement	unär

Java – Arithmetische Operatoren, Beispiel

```
/*
 * ArithmeticOperators.java
 */

package examples.operators;

public class ArithmeticOperators{

    public static void main(String[] args){
        int anInteger = 3;
        long aLong = 7L;
        float aFloat = 0.4f;
        double aDouble = 0.6;
        // implicit type cast
        System.out.println("int + long: " + (anInteger + aLong));
        System.out.println("float + double: " + (aFloat + aDouble));
        System.out.println("3/6 = " + (3/6));
        System.out.println("3%6 = " + (3%6));
        System.out.println("3.0/6 = " + (3.0/6));
        System.out.println("++anInteger * anInteger-- = "
          + (++anInteger * anInteger--));
        System.exit(0);
    }
}
```


DESY

Java – Arithmetische Operatoren, Beispiel Ausgabe

```
[:-)] [11:50:00]: /Users/dramburg/Code/Java/examples/operators
--> $ javac ArithmeticOperators.java
[:-)] [11:50:37]: /Users/dramburg/Code/Java/examples/operators
--> $ java examples.operators.ArithmeticOperators
int + long: 10
float + double: 1.0000000059604646
3/6 = 0
3%6 = 3
3.0/6 = 0.5
++anInteger * anInteger-- = 16
[:-)] [11:50:39]: /Users/dramburg/Code/Java/examples/operators
--> $
```


DESY

Java – Relationale Operatoren

Operator	Beschreibung	Operator-Typ
==	Gleichheit	binär
!=	Ungleichheit	binär
>	Größer als	binär
<	Kleiner als	binär
>=	Größer oder gleich	binär
<=	Kleiner oder gleich	binär

Java – Relationale Operatoren, Beispiel

```java
/*
 * RelationalOperators.java
 */

package examples.operators;

public class RelationalOperators{

    public static void main(String[] args){
        String s1 = "hello";
        String s2 = "hello";
        int i = 1;
        Integer anInteger = new Integer(1);
        Integer anotherInteger = new Integer(i);
        System.out.println("is i (1) == 1: " + (i == 1));
        System.out.println("s1 (\"hello\") == \"hello\": " + (s1 == "hello"));
        System.out.println("s1 (\"hello\") == s2 (\"hello\"): " + (s1 == s2));
        System.out.println("\"hello\" == \"hello\": " + ("hello" == "hello"));
        System.out.println("anInteger == anotherInteger: "
          + (anInteger == anotherInteger));
        // Correct object test on equal
        System.out.println("s1.equals(\"hello\"): " + s1.equals("hello"));
        System.out.println("anInteger.equals(anotherInteger): "
          + anInteger.equals(anotherInteger));
        System.out.println("'a' < 'b': " + ('a' < 'b'));
        System.out.println("100 > 99: " + (100 > 99));
        System.out.println("i <= 1: " + (i <= 1) + ", i >= 1: " + (i >= 1));
        System.exit(0);
    }
}
```


DESY

Java – Relationale Operatoren, Beispiel Ausgabe

```
[:-)] [15:03:55]: /Users/dramburg/Code/Java/examples/operators
--> $ javac RelationalOperators.java
[:-)] [15:03:58]: /Users/dramburg/Code/Java/examples/operators
--> $ java examples.operators.RelationalOperators
is i (1) == 1: true
s1 ("hello") == "hello": true
s1 ("hello") == s2 ("hello"): true
"hello" == "hello": true
anInteger == anotherInteger: false
s1.equals("hello"): true
anInteger.equals(anotherInteger): true
'a' < 'b': true
100 > 99: true
i <= 1: true, i >= 1: true
[:-)] [15:04:00]: /Users/dramburg/Code/Java/examples/operators
--> $
```

Java – Relationale Operatoren, Beispiel Besprechung

- Was vergleicht der == Operator?
- Den Inhalt zweier Speicherstellen
  - Ok bei primitiven Datentypen, Speicherstelle enthält Wert
  - NICHT Ok bei Referenztypen, Speicher enthält Adresse, liefert also keinen Wertvergleich, sondern nur einen Vergleich, ob es sich um das selbe Objekt handelt
- Warum funktioniert dann der Vergleich zweier Stringobjekte?
  - Funktioniert nur scheinbar
  - Stringliterale sind Konstanten, die nur einmal im Speicher abgelegt werden
  - Somit funktioniert der Vergleich unterschiedlicher Stringobjekte, die das gleiche Literal haben, da beide auf die gleiche Stringkonstante zeigen.
- Bessere Vergleichsmethoden für Objekte sind equals() (Object) und compareTo() (Comparable Interface)

Java – Logische Operatoren

Operator	Beschreibung	Operator-Typ
&&	Logisch UND	binär
| |	Logisch ODER	binär
!	Logisch NICHT	unär
? :	Bedingungs- oder Ternärer Operator	ternär

Der ternäre Operator kann als eine Kurzform einer if-else-Anweisung betrachtet werden.
X ? Y : Z kann man lesen als if X then Y else Z

Java – Logische Operatoren, Beispiel

```java
/*
 * LogicalOperators.java
 */

package examples.operators;

public class LogicalOperators{

    public static void main(String[] args){
        boolean b = true;
        boolean c = !b;
        System.out.println("true && true = " + (true && true));
        System.out.println("true && false = " + (true && false));
        System.out.println("false && false = " + (false && false));
        System.out.println("true || true = " + (true || true));
        System.out.println("true || false = " + (true || false));
        System.out.println("false || false = " + (false || false));
        System.out.println("!b = " + (!b));

        String s = b ? "b is true" : "b is false";
        System.out.println(s);
        String t = c ? "c is true" : "c is false";
        System.out.println(t);
        System.exit(0);
    }
}
```


DESY

Java – Logische Operatoren, Beispiel Ausgabe

```
[:-)] [16:06:51]: /Users/dramburg/Code/Java/examples/operators
--> $ javac LogicalOperators.java
[:-)] [16:06:57]: /Users/dramburg/Code/Java/examples/operators
--> $ java examples.operators.LogicalOperators
true && true = true
true && false = false
false && false = false
true || true = true
true || false = true
false || false = false
!b = false
b is true
c is false
[:-)] [16:07:12]: /Users/dramburg/Code/Java/examples/operators
--> $
```

Java – Logische Operatoren, short-circuit

- Logische Operatoren in Java sind sogenannte short-circuit Operatoren
- d.h. die Auswertung eines Ausdrucks, stoppt sobald klar ist, daß der Ausdruck nicht mehr zu wahr evaluiert werden kann
  - Beispiele:
    false && true – stoppt sofort, da false mit nichts verundet true ergeben kann
    false || true – hier muß auch der zweite Ausdruck ausgewertet werden, da false mit true verodert true ergibt.

Java – Operatoren zur Bitmanipulation

Operator	Beschreibung	Operator-Typ
&	Bitweise UND	binär
|	Bitweise ODER	binär
^	Bitweise XOR	binär
~	Bitweise Negation (Einer-Komplement)	unär
<<	Bitweise Linksschieben (left-shift)	binär
>>	Bitweise Rechtsschieben (right-shift)	binär
>>>	Vorzeichenloses Rechtsschieben (unsigned right-shift)	binär

Java – Bitmanipulation, Beispiel

```
/*
 * BitwiseOperators.java
 */

package examples.operators;

public class BitwiseOperators{

    public static void main(String[] args){
        int i = 1;
        int j = -1;
        System.out.println("1 & -1 = " + (i & j));
        System.out.println("1 | -1 = " + (i | j));
        System.out.println("1 ^ -1 + " + (i ^ j));
        System.out.println("~1 + 1 = " + ((~i) + 1));
        System.out.println("1 << 3 = " + (i << 3));
        System.out.println("8 >> 3 = " + (8 >> 3));
        System.out.println("8 >>> 3 = " + (8 >>> 3));
        System.out.println("-8 >> 3 = " + (-8 >> 3));
        System.out.println("-8 >>> 3 = " + (-8 >>> 3));
        System.exit(0);
    }
}
```

Java – Bitmanipulation, Beispiel Ausgabe

```
[:-)] [17:01:26]: /Users/dramburg/Code/Java/examples/operators
--> $ javac BitwiseOperators.java
[:-)] [17:03:00]: /Users/dramburg/Code/Java/examples/operators
--> $ java examples.operators.BitwiseOperators
1 & -1 = 1
1 | -1 = -1
1 ^ -1 + -2
~1 + 1 = -1
1 << 3 = 8
8 >> 3 = 1
8 >>> 3 = 1
-8 >> 3 = -1
-8 >>> 3 = 536870911
[:-)] [17:03:02]: /Users/dramburg/Code/Java/examples/operators
--> $
```

Java – right-shift und unsigned right-shift

- Der einfache right-shift Operator >> füllt von rechts immer mit der Zahl auf, die im höherwertigsten Bit steht
  - für positive Zahlen werden rechts Nullen eingefügt
  - für positive Zahlen werden rechts Einsen aufgefüllt
- Der unsigned right-shift Operator >>> füllt von rechts immer mit Nullen auf
  - für positive Zahlen Verhalten wie normales right-shift
  - für negative Zahlen, Verhalten wie bei einem unsigned 32 Bit Integer
- Alle Shift-Operatoren werten von der Shiftdistanz nur die ersten 5 Bit aus -> $2^5 = 32$

Java - Zuweisungsoperatoren

Operator	Beschreibung	Operator-Typ
=	Zuweisung	binär
+=	Zuweisung mit Addition	binär
-=	Zuweisung mit Subtraktion	binär
*=	Zuweisung mit Multiplikation	binär
/=	Zuweisung mit Division	binär
%=	Zuweisung mit Modulo	binär
<<=	Zuweisung mit Linksschieben	binär
>>=	Zuweisung mit Rechtsschieben	binär
>>>=	Zuweisung mit vorzeichenbehaftetem Rechtsschieben	binär
&=	Zuweisung mit Bitweisem UND	binär
^=	Zuweisung mit Bitweisem XOR	binär
|=	Zuweisung mit Bitweisem OR	binär

Java – Sonstige Operatoren

Operator	Beschreibung	Operator Typ
new	Erzeugungsoperator	unär
instanceof	Klassenüberprüfung	binär
( )	Cast-Operator, Parameterliste	
[ ]	Array-Subskript	
{ }	Block	
.	Zugriffs-, Referenz- oder Scope-Operator	
+	String-Konkatenation	binär
,	Komma, Parametertrenner	
:	Doppelpunkt, Sprungmarke	
//	Einzeilenkommentar	
/*	Start Mehrzeilenkommentar	
/**	Start Javadoc Kommentar	
*/	Ende Mehrzeilen und Javadoc Kommentar	

DESY

Java – Sonstige Operatoren, Beispiele

```
/*
 * OtherOperators.java
 */

package examples.operators;

public class OtherOperators{

    public static void main(String[] args){
        String aString = new String("Hello!");
        System.out.println("Is aString an instance of class String: "
            + aString instanceof String);
        System.out.println("And also an instance of class Object: "
            + aString instanceof Object);
        // doing some casts
        double d = 3.0;
        float f = (float)d;
        Object object = new Object();
        System.out.println("Type of aString: "
            + aString.getClass().getName());
        System.out.println("Type of object: "
            + object.getClass().getName());
        int[] intArray = new int[10];
        int[] anotherIntArray = {0,1,2,3,4,5,6,7,8,9};
        System.out.println("anotherIntArray[5] = " + anotherIntArray[5]);
        System.out.println("Size of intArray: " + intArray.length
            + ", size of anotherIntArray: " + anotherIntArray.length);
        // ATTENTION for strings length is a method NOT AN ATTRIBUTE!!!
        System.out.println("Size of aString: " + aString.length());
        System.exit(0);
    }
}
```


DESY

Java – Sonstige Operatoren, Beispiele Ausgabe

```
[:-)] [10:59:28]: /Users/dramburg/Code/Java/examples/operators
--> $ javac OtherOperators.java
[:-)] [11:02:59]: /Users/dramburg/Code/Java/examples/operators
--> $ java examples.operators.OtherOperators
true
true
Type of aString: java.lang.String
Type of object: java.lang.Object
anotherIntArray[5] = 5
Size of intArray: 10, size of anotherIntArray: 10
Size of aString: 6
[:-)] [11:03:02]: /Users/dramburg/Code/Java/examples/operators
--> $
```

Java – Inkrement/Dekrement in Prä- und Postfix Notation

- Inkrement ++
- Dekrement --
- Präfix-Notation
  - Wert wird verändert und dann erst genutzt, z.B.:
  - int a = 0; System.out.print("a = " + (++a)); Ausgabe: a = 1
  - int b = 1; System.out.print("b = " + (--b)); Ausgabe: b = 0
- Postfix-Notation
  - Wert wird erst genutzt und dann erst verändert, z.B.:
  - int a = 0; System.out.print("a = " + (a++)); Ausgabe: a = 0
  - int b = 1; System.out.print("b = " + (b--)); Ausgabe: b = 1

DESY

Java – Auswertungsreihenfolge von Operatoren

- In der Regel werden Ausdrücke von links nach rechts ausgewertet (links-assoziativ) ein paar Operatoren werden jedoch von rechts nach links ausgewertet (rechts-assoziativ), diese Ausnahmen sind Präfix-Inkrement und -Dekrement, die Zuweisungen und der ternäre Operator
- Reihenfolge (von hoher zu niedriger Priorität)
  - (Postfix) ++, (Postfix) --, ( ), [ ], (Scope)., { }
  - ++ (Präfix), -- (Präfix), + (Vorzeichen), - (Vorzeichen), !, ~, (Typ-Cast),
  - *, /, %
  - + (String-Konkatenation), +, -
  - <<, >>, >>>
  - <, <=, >, >=, instanceof
  - ==, !=

Java – Auswertungsreihenfolge von Operatoren

- Reihenfolge (Fortsetzung)
  - &
  - ^
  - |
  - &&
  - | |
  - ? :
  - =, *=, /=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=
  - ,

Java – Vorsicht bei Stringkonkatenation und Addition

```java
/*
 * TestStringConcatenation.java
 */

package examples.tests;

public class TestStringConcatenation{

    public static void main(String[] args){
        System.out.println("print integer product: " + 2 * 2);
        System.out.println("print integer product: " + (2 * 2));
        System.out.println("print integer sum: " + 2 + 2);
        System.out.println("print integer sum: " + (2 + 2));
    }
}
```

```
 [:-)] [11:25:36]: /Users/dramburg/Code/Java/examples/tests
 --> $ javac TestStringConcatenation.java
 [:-)] [11:25:41]: /Users/dramburg/Code/Java/examples/tests
 --> $ java examples.tests.TestStringConcatenation
 print integer product: 4
 print integer product: 4
 print integer sum: 22
 print integer sum: 4
```


DESY

Java – Reservierte Wörter und Schlüsselwörter

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

Die reservierten Wörter können nicht als Bezeichner für Variablen, Methoden oder Klassen benutzt werden. Die Keywords const und goto sind zwar reserviert, werden zur Zeit aber in Java nicht genutzt.

Java – Modifier, Bedeutung

Modifier	Bedeutung
abstract	Nicht instanzierbar, enthält nur Signatur
final	Kann nicht erweitert/überschrieben werden, Wert nicht änderbar
keiner **(package)**	Nur im package sichtbar
native	Plattform spezifische Implementierung (kein/e Signatur/Body)
private	Nur in Klasse sichtbar in der definiert
protected	Im eigenen Package und Unterklassen sichtbar
public	Überall sichtbar
strictfp	Alle Methoden der Klasse bzw. die Methode nutzt Floating Point Implementierung der JVM nicht der Hardware
static	Klassenmethode, -variable, innere Klasse wird Top-Level
synchronized	Nebenläufigkeit, lock für Klasse/Objekt/Methode
transient	Wird nicht serialisiert
volatile	Nebenläufigkeit, hält eine variable in Threads synchron

Java – Modifier, sind erlaubt bei:

Modifier	Klasse	Interface	Enum	Konstruktor	Methode	Variable
public	ja	ja	ja	ja	ja	ja
protected	nein	nein	ja/nein	ja	ja	ja
keiner **(package/ default)**	ja	ja	ja	ja	ja	ja
private	nein	nein	ja/nein	ja	ja	ja
final	ja	nein	nein	ja	ja	ja
abstract	ja	ja,redundant	nein	nein	ja	nein
static	nein	nein	ja/nein	nein	ja	ja
native	nein	nein	nein	nein	ja	nein
transient	nein	nein	nein	nein	nein	ja
volatile	nein	nein	nein	nein	nein	ja
synchronized	nein	nein	nein	nein	ja	nein
strictfp	ja	ja	ja	nein	ja	nein

DESY

Java – Syntax: if-Statement

```
// if-statement
if(TEST){
    CODE;
}

// if-else-statement
if(TEST){
    CODE;
}else{
    CODE;
}

// if-else-if-else-statement
if(TEST){
    CODE;
}else if(TEST){
    CODE;
}else{
    CODE;
}
```


DESY

Java – Syntax: if-Statement, Beispiel

```
/*
 * IfExample.java
 */

package examples.syntax;

public class IfExample{

    public static void main(String[] args){
        if(args.length == 0){
            System.out.println(
                "You have called the program without arguments!");
        }else if(args.length > 0 && args.length < 4){
            System.out.println(
                "You have called the program with these arguments:");
            for(int i = 0; i < args.length; i++){
                System.out.println("args[" + i + "] = " + args[i]);
            }
        }else{
            System.out.println("Way to much arguments!");
        }
        System.exit(0);
    }
}
```

Java – Syntax: if-Statement, Beispiel Ausgabe

```
[:-)] [12:44:19]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac IfExample.java
[:-)] [12:44:44]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.IfExample
You have called the program without arguments!
[:-)] [12:44:50]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.IfExample eins zwei drei
You have called the program with these arguments:
args[0] = eins
args[1] = zwei
args[2] = drei
[:-)] [12:44:55]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.IfExample eins zwei drei vier
Way to much arguments!
[:-)] [12:45:01]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java – Syntax: switch-Statement

```
// A switch statement is similar to an
// if - if-else - else statement. In Java
// the used expression must return an int,
// a String or an enumerated value.
switch(EXPRESSION){
    case VALUE_1:
        CODE;
        break;
    default:
        CODE;
}
// or with more than one case
switch(EXPRESSION){
    case VALUE_1:
        CODE;
        break;
    case VALUE_2:
        CODE;
        break;
        .
        .
        .
    case VALUE_N:
        CODE;
        break;
    default:
        CODE;
}
```

Java – Syntax: switch-Statement, Beispiel

```
/*
 * SwitchExample.java
 */

package examples.syntax;

public class SwitchExample{

    public static void main(String[] args){
        if (args.length >= 1){
            System.out.println("Argument is of type: "
                + args[0].getClass().getName());
            switch(args[0]){
                case "eins":
                case "Eins":
                    System.out.println("EINS");
                    break;
                case "zwei":
                case "Zwei":
                    System.out.println("ZWEI");
                    break;
                default:
                    System.out.println(
                        "Found nothing, only the default!");
            }
        }
        System.exit(0);
    }
}
```


DESY

Java – Syntax: switch-Statement, Beispiel Ausgabe

```
[:-)] [13:41:41]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac SwitchExample.java
[:-)] [13:41:43]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.SwitchExample eins
Argument is of type: java.lang.String
EINS
[:-)] [13:41:50]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.SwitchExample Zwei
Argument is of type: java.lang.String
ZWEI
[:-)] [13:42:02]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.SwitchExample drei
Argument is of type: java.lang.String
Found nothing, only the default!
[:-)] [13:42:20]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java – Syntax: while-Schleife

```
// while loops are the most basic loops
while(TEST){
    CODE;
}

// with an inner test and continue statement
while(TEST){
    if(INNER-TEST){
        continue;
    }
    CODE;
}

// with an inner test and break statement
while(TEST){
    if(INNER-TEST){
        break;
    }
    CODE;
}

// a do-while loop, first runs the code
// then does the testing, code is executed
// at least one time
do{
    CODE;
}while(TEST);
```


DESY

Java – Syntax: while-Schleife, Beispiel

```
/*
 * WhileExample.java
 */

package examples.syntax;

public class WhileExample{

    public static void main(String[] args){
        int indexCounter = 0;
        System.out.println("First loop:");
        while(++indexCounter < args.length){
            System.out.println("args["
                + indexCounter + "] = " + args[indexCounter]);
            indexCounter++;
        }
        indexCounter = 0;
        System.out.println("Second loop:");
        while(true){
            if(indexCounter >= args.length){
                System.out.println("Within break condition");
                break;
            }
            System.out.println("args["
                + indexCounter + "] = " + args[indexCounter]);
            indexCounter++;
        }

        System.exit(0);
    }
}
```


DESY

Java – Syntax: while-Schleife, Beispiel Ausgabe

```
[:-)] [14:24:29]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac WhileExample.java
[:-)] [14:24:48]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.WhileExample
First loop:
Second loop:
Within break condition
[:-)] [14:24:50]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.WhileExample eins zwei drei
First loop:
args[0] = eins
args[1] = zwei
args[2] = drei
Second loop:
args[0] = eins
args[1] = zwei
args[2] = drei
Within break condition
[:-)] [14:24:56]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java – Syntax: while-Schleife mit break, continue, Label

```
// As goto in Java isn't used, whether it's reserved,
// the only place you can use labels is for distinct
// separation of nested loops. It's mainly used
// for clarification and better readability.
// The label has to be placed directly before the
// loop statement!
CODE;
OUTER_LABEL:
while(TEST){
    INNER_LABEL:
    while(TEST){
        if(INNER_TEST_1){
            continue INNER_LABEL;
        }
        if(INNER_TEST_2){
            break OUTER_LABEL;
        }
    }// branch to INNER_LABEL goes here
    CODE;
}// branch to OUTER_LABEL goes here
```

Java – Syntax: while-Schleife mit Label, Beispiel

```
/*
 * WhileLabelExample.java
 */

package examples.syntax;

public class WhileLabelExample{

    public static void main(String[] args){

        int anArray[][] = {{0,1,3},{5,7,9},{11,13,15},{17,19,21}};
        boolean onlyFirstTimeBranch = true;
        int i = 0;
        outer: // OUTER_LABEL
        while(i < anArray.length){
            int j = 0;
            inner: // INNER_LABEL
            while(j < anArray[i].length){
                int value = anArray[i][j];
                System.out.println("arr["
                    + i + "][" + j + "]: " +    value);
                j++;
                if(value > 7 && onlyFirstTimeBranch){
                    System.out.println("continue inner label");
                    onlyFirstTimeBranch = false;
                    continue inner;
                }
                if(value/5 == 3){
                    System.out.println("break outer label");
                    break outer;
                }
            }
            i++;
            System.out.println("Reached inner label");
        }
        System.out.println("Reached outer label");
        System.exit(0);
    }
}
```


DESY

Java – Syntax: while-Schleife, Label, Beispiel Ausgabe

```
[:-)] [16:05:44]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac WhileLabelExample.java
[:-)] [16:13:37]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.WhileLabelExample
arr[0][0]: 0
arr[0][1]: 1
arr[0][2]: 3
Reached inner label
arr[1][0]: 5
arr[1][1]: 7
arr[1][2]: 9
continue inner label
Reached inner label
arr[2][0]: 11
arr[2][1]: 13
arr[2][2]: 15
break outer label
Reached outer label
[:-)] [16:13:39]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

https://xkcd.com/292/ Maus über das letzte Bild, Tooltip!

DESY

Java – Syntax: for-Schleife

```
// the for loop does a controlled counted loop
// through a (mostly) fixed number of iterations
// using a loop counter.
// INIT is the loop counter declaration and initialization
// STEP is mostly some kind of increment or decrement,
// but could also be a statement like: i += 2

for(INIT, TEST, STEP){
    CODE;
}

// Also as an alternate version to the C-style for-loop
// you can use an Iterator object as loop variable

for(GET_ITERATOR; ITERATOR_HAS_NEXT;){
    CODE(ITERATOR_NEXT);
}
```


DESY

Java – Syntax: for-Schleife, Beispiel

```
/*
 * ForExample.java
 */

package examples.syntax;

public class ForExample{

    public static final int SIZE = 4;

    public static void main(String[] args){

        int table[][][] = new int[SIZE][SIZE][SIZE];
        for(int i = 0; i < table.length; i++){
            for(int j = 0; j < table[i].length; j++){
                for(int k = 0; k < table[i][j].length; k++){
                    table[i][j][k] = i * j * k;
                }
            }
        }
        System.out.println("Here comes the three-dimensional array:");
        for(int i = 0; i < table.length; i++){
            for(int j = 0; j < table[i].length; j++){
                System.out.print("{");
                for(int k = 0; k < table[i][j].length; k++){
                    System.out.print("" + table[i][j][k]
                            + (k < SIZE - 1 ? ", " : ""));
                }
                System.out.println("}");
            }
            System.out.println("");
        }
        System.exit(0);
    }
}
```


DESY

Java – Syntax: for-Schleife, Beispiel Ausgabe

```
[:-)] [16:59:57]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac ForExample.java
[:-)] [17:00:50]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.ForExample
Here comes the three-dimensional array:
{0, 0, 0, 0}
{0, 0, 0, 0}
{0, 0, 0, 0}
{0, 0, 0, 0}

{0, 0, 0, 0}
{0, 1, 2, 3}
{0, 2, 4, 6}
{0, 3, 6, 9}

{0, 0, 0, 0}
{0, 2, 4, 6}
{0, 4, 8, 12}
{0, 6, 12, 18}

{0, 0, 0, 0}
{0, 3, 6, 9}
{0, 6, 12, 18}
{0, 9, 18, 27}
```

Java – Syntax: for-Schleife mit Iterator, Beispiel

```java
/*
 * ForIteratorExample.java
 */

package examples.syntax;

import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class ForIteratorExample{

    public static void main(String[] args){
        List<String> names = new ArrayList<String>();
        names.add("Helga");
        names.add("Wolfgang");
        names.add("Heidi");
        names.add("Frank");

        for(Iterator<String> it = names.iterator(); it.hasNext();){
            System.out.println(it.next());
        }
    }
}
```

Java – Syntax: for-Schleife mit Iterator, Beispiel Ausgabe

```
[:-)] [10:36:30]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac ForIteratorExample.java
[:-)] [10:37:09]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.ForIteratorExample
Helga
Wolfgang
Heidi
Frank
[:-)] [10:37:15]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java – Syntax: foreach-Schleife

```
// Since Java 5.0 you can use a somehow
// simplified and clarified version of a for-loop
// the so-called for-each loop. Called so because
// the loop logic puts every element of the iterated
// array or collection exactly one time on a declared
// variable.
// TYPE means the element type collected in the
// data structure, and COLLECTION is the structure
// over which should be iterated, could be a Collection
// type or an array

for(TYPE name: COLECTION){
    CODE(name);
}
```


DESY

Java – Syntax: foreach-Schleife, Beispiel

```
/*
 * ForEachExample.java
 */

package examples.syntax;

import java.util.List;
import java.util.ArrayList;

public class ForEachExample{

    public static void main(String[] args){
        int[] numbers = {1,2,3,4};
        List<String> names = new ArrayList<String>();
        names.add("Helga");
        names.add("Wolfgang");
        names.add("Heidi");
        names.add("Frank");

        for(int aNumber: numbers){
            System.out.println(aNumber);
        }
        for(String aName: names){
            System.out.println(aName);
        }
        System.exit(0);
    }
}
```


DESY

Java – Syntax: foreach-Schleife, Beispiel Ausgabe

```
[:-)] [11:30:39]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac ForEachExample.java
[:-)] [11:35:11]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.ForEachExample
1
2
3
4
Helga
Wolfgang
Heidi
Frank
[:-)] [11:35:29]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java - Enumerations

- Java Enumerations sind wesentlich vielseitiger als in C
- Alle Elemente einer enum sind Konstanten, können also nach der ersten Initialisierung nicht mehr geändert werden.
- Alle Elemente können ihrerseits auch konstante Attribute halten.
- Außerdem erben alle Elemente die Eigenschaften der abstrakten Klasse java.lang.Enum, die von java.lang.Object abgeleitet ist, somit sind die Elemente vollwertige Objekte, die über die vererbten Interfaces Comparable<T> vergleichbar sind.
- Zusätzlich sind in java.lang.Enum noch nützliche Methoden implementiert, wie name() – liefert den Namen der Konstanten als String, ordinal() – gibt den der Konstanten zugeordneten Zahlwert als int zurück und toString() – die intern name() aufruft

Java - Enumerations

- Enumerations können wie Klassen über einen Konstruktor und Methoden verfügen, sowie Variablen halten. In einer enum kann auch eine main-Methode deklariert werden. Erhält eine enum einen Zugriffsmodifikator muß sie wie eine Klasse in einer eigene Textdatei mit dem Namen der enum definiert werden.
- Der Compiler fügt jeder enum, bei Erzeugung eine statische Methode values() hinzu, die einen Array mit allen Konstanten zurückliefert.

Java – Syntax: enums/Enumerations

```
// enums in Java are a lot more powerful
// than enumerations in C. They can be used
// in C-stylish way, but are more something
// between a C struct and a Java class. As in
// C they define constants and therefore the enum
// elements are written in uppercase. But they can
// define methods as a main and also can define a constructor,
// like a Java class, and must declared in an own
// file when a access modifier like public is used

// simple C-style variant
enum Name{ CONSTANT_1, CONSTANT_2, ..., CONSTANT_N}

// complex Java-style
ACCESS_MODIFIER enum Name{
    CONSTANT_1(constValue_1, constValue_2, ..., constValue_n),
    CONSTANT_2(constValue_1, constValue_2, ..., constValue_n),
    ...,
    CONSTANT_N(constValue_1, constValue_2, ..., constValue_n);

    INSTANCE_VARIABLES_DECLARATION

    Name(TYPE value_1, TYPE value_2, ..., TYPE value_n){
        ...// the constructor for the enum
    }

    METHODS_DECLARATIONS
}
```


DESY

Java – Syntax: simple enum, Beispiel

```java
/*
 * EnumExample.java
 */

package examples.syntax;

enum Direction{
    NORTH, SOUTH, EAST, WEST
}

public class EnumExample{

    public static void main(String[] args){

        System.out.println(Direction.NORTH);
        System.out.println(Direction.SOUTH);
        System.out.println(Direction.EAST);
        System.out.println(Direction.WEST);

        System.exit(0);
    }
}
```


DESY

Java – Syntax: simple enum, Beispiel Ausgabe

```
[:-)] [12:28:12]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac EnumExample.java
[:-)] [12:29:54]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.EnumExample
NORTH
SOUTH
EAST
WEST
[:-)] [12:29:57]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java – Syntax: Java enum, Beispiel

```java
/*
 * Day.java
 */

package examples.syntax;

public enum Day{
    MONDAY("Monday", "Montag"),
    TUESDAY("Tuesday", "Dienstag"),
    WEDNESDAY("Wednesday", "Mittwoch"),
    THURSDAY("Thursday", "Donnerstag"),
    FRIDAY("Friday", "Freitag"),
    SATURDAY("Saturday", "Samstag"),
    SUNDAY("Sunday", "Sonntag");

    public final String english;
    public final String german;

    Day(String english, String german){
        this.english = english;
        this.german = german;
    }

    public static void main(String[] args){

        for(Day aDay: Day.values()){
            System.out.println(aDay.english
                + " is called in Germany "
                + aDay.german);
        }
        System.exit(0);
    }
}
```

Java – Syntax: Java enum, Beispiel Ausgabe

```
[:-)] [12:20:24]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac Day.java
[:-)] [12:20:40]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.Day
Monday is called in Germany Montag
Tuesday is called in Germany Dienstag
Wednesday is called in Germany Mittwoch
Thursday is called in Germany Donnerstag
Friday is called in Germany Freitag
Saturday is called in Germany Samstag
Sunday is called in Germany Sonntag
[:-)] [12:20:43]: /Users/dramburg/Code/Java/examples/syntax
```

Java – Klassen und Interfaces

- Sprachen wie C++ unterstützen ein Konstrukt wie Mehrfachvererbung, um Klassen die Eigenschaften von mehreren Klassen zukommen zu lassen, oft führt das allerdings zu nicht ganz eindeutigem Code.
- Java unterstützt deshalb keine Mehrfachvererbung, Klassenhierarchien sind in Java eindeutig.
- Um Klassen in Java aber Eigenschaften ausserhalb einer "ist-ein"-Beziehung mitzugeben existieren sogenannte Interfaces
- Interfaces definieren Schnittstellenbeschreibungen, die eine Klasse, die ein Interface implementiert umsetzen und garantieren muß, ansonsten wirft der Compiler einen Fehler.
- Bis Java 8 konnten in Java nur Methodensignaturen definiert werden, es gab keine Möglichkeit der Implementation. Ab Java 8 können Methoden auch in Interfaces implementiert werden (default implementation).

Java - Klassen

- Klassen sind in Java die grundlegenden Strukturen, die Daten und die auf ihnen arbeitenden Methoden definieren.
- Klassen bilden in Java eine Hierarchie mit java.lang.Object als oberste Klasse von der alle Klassen in Java abgeleitet sind.
- Klassen brauche einen Konstruktor, damit konkrete Objekte aus ihnen erzeugt werden können. Wird keiner explizit definiert wird der implizite Default-Konstruktor benutzt, der ist allerdings nicht mehr aufrufbar, wenn ein Konstruktor definiert wurde.
- Klassen können abstract sein, dann können sie nicht instanziert werden und definieren mehr oder weniger eine vererbbare Schnittstelle

Java – Syntax: Klassen-Deklaration

```
// class syntax
MODIFIER(S) class CLASS_NAME{
    CLASS_VARIABLES_DECLARATION
    INSTANCE_VARIABLES_DECLARATION

    MODIFIER CLASS_NAME(){
        // default constructor
    }

    MODIFIER CLASS_NAME(PARAMETERS){
        // constructor with parameters
    }

    METHODS_DECLARATIONS
}
// general class declaration
public class NormalClass{...}

// final class, no further specialization possible
public final class NoMoreSpecialization{...}

// abstract class declaration, no instantiation possible
public abstract class NoObjectsPossible{...}
```


DESY

Java – Syntax: Klassen-Deklaration, Beispiel

```java
/*
 * ExampleClass.java
 */

package examples.syntax;

public class ExampleClass{
    public static final String classVariable = "ExampleClass";
    private String instanceVariable;
    // default constructor
    public ExampleClass(){}
    // constructor with parameters
    public ExampleClass(String instanceVariable){
        this.instanceVariable = instanceVariable;
    }
    // setter-method
    public void setInstanceVariable(String instanceVariable){
        this.instanceVariable = instanceVariable;
    }
    // getter-method
    public String getInstanceVariable(){
        return this.instanceVariable;
    }
    // overwrite toString from java.lang.Object
    @Override
    public String toString(){
        return ExampleClass.classVariable + ": "
                + this.instanceVariable;
    }
```


DESY

Java – Syntax: Klassen-Deklaration, Beispiel Ausgabe

```
    public static void main(String[] args){
        ExampleClass example1 = new ExampleClass();
        ExampleClass example2 = new ExampleClass("TWO");
        example1.setInstanceVariable("ONE");
        System.out.println(example2.getInstanceVariable());
        example2.setInstanceVariable("Two");
        System.out.println(example1.getInstanceVariable());
        System.out.println(example1);
        System.out.println(example2);
        System.exit(0);
    }
}
```

```
[:-)] [17:16:21]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac ExampleClass.java
[:-)] [17:16:24]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.ExampleClass
TWO
ONE
ExampleClass: ONE
ExampleClass: Two
[:-)] [17:16:26]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java – Aus Klassen Objekte erzeugen mit new

- Aus einer Klassendeklaration wird mit Hilfe des new-Operators eine Instanz/ein Objekt dieser Klasse erzeugt.
- Bei Aufruf wird der für eine Instanz dieser Klasse notwendige Speicher reserviert, dann das Objekt in diesen Bereich geladen. Anschliessend wird das Objekt über den aufgerufenen Konstruktor initialisiert und eine Referenz (Adresse) auf das initialisierte Objekt zurückgeliefert.
- Der richtige Konstruktor wird anhand der Signatur gefunden.
- Ist kein expliziter Konstruktor vorhanden, wird in evtl. vorhandenen Oberklassen nach einem Konstruktor gesucht. Ansonsten wird vom Compiler ein default-Konstruktor ohne Argumente erzeugt.

Java – static Factory-Methoden

- Ist der Konstruktor private funktioniert die Instanzierung mittels new nicht von außerhalb der Klasse bzw. aus einem static Kontext (main) heraus nicht mehr.
- Hierfür wird eine public static Methode benötigt, die den Konstruktor aus der Klasse aufruft und eine Referenz zurückliefert. Diese Methoden werden auch static Factory-Methoden genannt.
- Private Konstruktoren braucht man immer dann, wenn innerhalb einer Anwendung eine gewisse Kontrolle über Erzeugung und Anzahl bestimmter Objekte notwendig.
- Der Sonderfall, daß lediglich ein Objekt einer Klasse erzeugt werden darf ist das sogenannte Singleton(-Pattern). Hierfür wird noch eine Klassenvariable benötigt, die die Referenz des Objekts hält benötigt. Eine mögliche Anwendung wäre die Nutzung eines Konfigurationsobjekts für die Applikation.

Java – static Factory-Methoden, Beispiel

```java
/*
 * StaticFactory.java
 */

package examples.syntax;

public class StaticFactory{

    private static final int maxObjects = 2;
    private static int objectCounter = 0;
    private final int aValue;

    private StaticFactory(int aValue){
        this.aValue = aValue;
    }

    public String toString(){
        return "Object No. " + StaticFactory.objectCounter
            + " with value "+ this.aValue;
    }

    public static StaticFactory getInstance(int aValue){
        if (StaticFactory.objectCounter < StaticFactory.maxObjects){
            StaticFactory instance = new StaticFactory(aValue);
            StaticFactory.objectCounter++;
            System.out.println(instance);
            return instance;
        }else{
            System.out.println("No more objects allowed");
            return null;
        }
    }
```


DESY

Java – static Factory-Methoden, Beispiel und Ausgabe

```
    public static void main(String[] args){
        StaticFactory sf1 = StaticFactory.getInstance(5);
        StaticFactory sf2 = StaticFactory.getInstance(10);
        StaticFactory sf3 = StaticFactory.getInstance(15);
        //if (sf1, sf2, sf3 != null) {doSomethingWith(sf1,sf2,sf3)}
        System.exit(0);
    }
}
```

```
[:-)] [12:20:57]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac StaticFactory.java
[:-)] [12:21:42]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.StaticFactory
Object No. 1 with value 5
Object No. 2 with value 10
No more objects allowed
[:-)] [12:21:45]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java - Singleton

```java
/*
 * Singleton.java
 */

package examples.syntax;

public class Singleton{

    private final static Singleton instance = new Singleton();
    private int objectCounter = 0;

    private Singleton(){
        this.objectCounter++;
    }

    public static Singleton getInstance(){
        return instance;
    }

    public String toString(){
        return "Singleton #" + this.objectCounter;
    }

    public static void main(String[] args){
        Singleton single1 = Singleton.getInstance();
        Singleton single2 = Singleton.getInstance();
        Singleton single3 = Singleton.getInstance();
        System.out.println(single1);
        System.out.println(single2);
        System.out.println(single3);
    }
}
```

Java – Singleton, Ausgabe

```
[:-)] [16:11:15]: /Users/dramburg/Code/Java/examples/syntax
--> $ javac Singleton.java
[:-)] [16:11:23]: /Users/dramburg/Code/Java/examples/syntax
--> $ java examples.syntax.Singleton
Singleton #1
Singleton #1
Singleton #1
[:-)] [16:11:39]: /Users/dramburg/Code/Java/examples/syntax
--> $
```

Java - Aufgabe

- Erstellt drei Klassen, eine Klasse Person, Address und Date
- Eure Klassen und was ihr zur Aufgabe sonst noch realisiert, sollen im package exercises.one liegen
- Person hat vier Instanzvariablen für den Vornamen, den Nachnamen, die Adresse und das Geburtsdatum.
- Address hat vier Instanzvariablen für die Straße, die Hausnummer, die PLZ und die Stadt
- Date hat drei Instanzvariablen für Tag, Monat und Jahr, alle drei vom Typ int und keine weiteren Variablen
- Alle Instanzvariablen sind private, bei welchen Instanzvariablen macht es Sinn sie als private final zu deklarieren?
- Die Methode toString() aus java.lang.Object soll in allen Klassen überschrieben und zur Ausgabe genutzt werden.
- Zum Testen soll die angehängte TestDriver.java benutzt werden.

DESY

Aufgabe – TestDriver.java

```java
/*
 * TestDriver.java
 * A short test for the
 * Person, Address and Date classes
 */

package examples.exercises;

import exercises.one.Person;
import exercises.one.Address;
import exercises.one.Date;

public class TestDriver{

    public static void main(String[] args){
        Date dateOfBirth1 = new Date(13,7,1983);
        Date dateOfBirth2 = new Date(23,9,1981);
        Date dateOfBirth3 = new Date(1,5,1994);
        Date dateOfBirth4 = new Date(11,1,1973);
        Address address1 =
            new Address("Westendorfer Weg", 51, 21031, "Hamburg");
        Address address2 =
            new Address("Ebertallee", 1, 22607, "Hamburg");
        Address address3 =
            new Address("Stresemannstraße", 163, 22769, "Hamburg");
        Address address4 =
            new Address("Breite Straße", 111, 22767, "Hamburg");
        Person person1 =
            new Person("Helga", "Müller", dateOfBirth1, address1);
        Person person2 =
            new Person("Frank", "Meier", dateOfBirth2, address2);
        Person person3 =
            new Person("Eva", "Heydrich", dateOfBirth3, address3);
        Person person4 =
            new Person("Stefan", "Liebig", dateOfBirth4, address4);
        System.out.println(person1);
        System.out.println(person2);
        System.out.println(person3);
        System.out.println(person4);
    }
}
```

Aufgabe – TestDriver.java

```
// Ausgabe: BITTE ohne die Kommentarzeichen vorneweg ;-)
// Muss auch nicht in blau sein!

// Vorname: Helga
// Nachname: Müller
// Geboren am: 13. July 1983
// Westendorfer Weg 51
// 21031 Hamburg
//
// Vorname: Frank
// Nachname: Meier
// Geboren am: 23. September 1981
// Ebertallee 1
// 22607 Hamburg
//
// Vorname: Eva
// Nachname: Heydrich
// Geboren am: 1. Mai 1994
// Stresemannstraße 163
// 22769 Hamburg
//
// Vorname: Stefan
// Nachname: Liebig
// Geboren am: 11. Januar 1983
// Breite Straße 111
// 22767 Hamburg
//
```

Java – Abstrakte Klassen

- Werden definiert durch das Schlüsselwort abstract
- Abstrakte Klassen können nicht instanziert werden, kein default-Konstruktor wird angelegt.
- Kann abstrakte wie konkrete Methoden enthalten.
- Dient dazu über abstrakte Methoden instanzierbare Unterklassen zu einer Implementierung zu "zwingen"

Java – Abstrakte Klasse, Beispiel

```java
/*
 * Bird.java
 */

package examples.abstracts;

public abstract class Bird{

    protected String color;

    public abstract String fly();

    @Override
    public String toString(){
        return this.color;
    }
}

/*
 * Raven.java
 */

package examples.abstracts;

public class Raven extends Bird{

     public Raven(){
        this.color = "black";
     }

     @Override
     public String fly(){
        return "The " + this.color + " raven starts to fly.";
     }
}
```


DESY

Java – Abstrakte Klasse, Beispiel Ausgabe

```
/*
 * TestBirdAndRaven.java
 */

package examples.abstracts;

public class TestBirdAndRaven{

    public static void main(String[] args){
        //Bird bird = new Bird(); // compile time error!
        Bird raven = new Raven();
        System.out.println(raven.fly());
        System.out.println(raven);
        System.exit(0);
    }
}

// try to instantiate Bird
[:-(] [13:40:03]: /Users/dramburg/Code/Java/examples/abstracts
--> $ javac *.java
TestBirdAndRaven.java:10: error: Bird is abstract; cannot be instantiated
        Bird bird = new Bird();
                    ^
1 error

// correct output
[:-)] [13:46:09]: /Users/dramburg/Code/Java/examples/abstracts
--> $ javac *.java
[:-)] [13:46:58]: /Users/dramburg/Code/Java/examples/abstracts
--> $ java examples.abstracts.TestBirdAndRaven
The black raven starts to fly.
black
```


DESY

Java – Interfaces, Old Style

- Bis Java 8 konnten Interfaces keine Methoden implementieren und keine Variablen, waren also eindeutig in der Funktionalität von abstrakten Klassen unterscheidbar.
- Old Style Interfaces halten Methodensignaturen und "zwingen" Klassen, die diese Interfaces implementieren auch die Methoden zu realisieren. Hierbei handelt es sich um eine wirkliche Schnittstellenimplementierung.
- Zusätzlich wurden und werden Interfaces als sogenannte "Marker Interfaces" genutzt. Dies meint ein leeres Interface, das nur genutzt wird um Code eine bestimmte Verarbeitung der Objekte, die dieses Marker Interface implementieren, zu signalisieren. Dies lässt sich seit Java 1.4 meist besser mit Annotations umsetzen.
- Methoden in Interfaces sind implizit public, man kann sich das Schlüsselwortprinzipiell sparen, aber es auch der besseren Lesbarkeit wegen nutzen

Java – Syntax: Interfaces

Java – Interface, Beispiel

```java
/* Bird.java */
package examples.interfaces;

public abstract class Bird{

    protected String color;
    public abstract String fly();

}

/* NestBuilder.java */
package examples.interfaces;

public interface NestBuilder{
    public String buildANest();
}

/* Raven.java */
package examples.interfaces;

public class Raven extends Bird implements NestBuilder{

     public Raven(){
        this.color = "black";
     }

     @Override
     public String fly(){
        return "The " + this.color + " raven starts to fly.";
     }

     public String buildANest(){
         return "A nest has been built.";
     }
}
```


DESY

Java – Interface, Beispiel und Ausgabe

```
/*
 * TestBirdAndRaven.java
 */

package examples.interfaces;

public class TestBirdAndRaven{

    public static void main(String[] args){
        Raven raven = new Raven();
        System.out.println(raven.fly());
        System.out.println(raven.buildANest());
        System.exit(0);
    }
}

// Ausgabe:
[:-)] [16:47:45]: /Users/dramburg/Code/Java/examples/interfaces
--> $ javac *.java
[:-)] [16:47:49]: /Users/dramburg/Code/Java/examples/interfaces
--> $ java examples.interfaces.TestBirdAndRaven
The black raven starts to fly.
A nest has been built.
```

Java – Marker Interface, Beispiel

```java
/*
 * Testable.java
 */
package examples.interfaces;

public interface Testable{
    // a marker interface
}

/*
 * ATestableClass.java
 */
package examples.interfaces;

public class ATestableClass implements Testable{

    public boolean test;

    public ATestableClass(boolean test){
        this.test = test;
    }

    public boolean doTest(){
        if(!this.test){
            this.test = true;
        }
        return this.test;
    }
}
```

Java – Marker Interface, Beispiel und Ausgabe

```java
/*
 * MyApplication.java
 */
package examples.interfaces;

public class MyApplication{

    public static void main(String[] args){
        ATestableClass aClass = new ATestableClass(false);
        if (aClass instanceof Testable){
            System.out.println(aClass.doTest());
        }
        System.exit(0);
    }
}

// Ausgabe:
[:-)] [15:58:49]: /Users/dramburg/Code/Java/examples/interfaces
--> $ javac *.java
[:-)] [15:58:51]: /Users/dramburg/Code/Java/examples/interfaces
--> $ java examples.interfaces.MyApplication
true
```

Java – Interfaces, New Style with default implementation

- Seit Java 8, lassen sich in Interfaces auch Methoden implementieren und zwar sogenannte default-Methoden und statische Methoden
- Den default-Methoden wird das Schlüsselwort default voran gestellt
- Aufpassen muß man Fällen in denen eine Klasse mehrere Interfaces implementiert, die alle die gleiche Methodensignatur als default-Implementierung nutzen. Der Compiler wirft in solchen Fällen einen Fehler, wenn nicht explizit angegeben wird, welche Methode aufgerufen werden soll.
- Auch seit Java 8 können in Interfaces statische Methoden, als "Helfer"-Methoden für die default-Methoden implementiert werden.

Java – Fehlerbehandlung I

- In Java ist eine ordentliche Fehlerbehandlung in die Sprache eingebaut.
- Java unterscheidet zwischen den Fehlerklassen Error und Exception, sowie konzeptionell zwischen checked und unchecked Exceptions.
- Error und Exception sind als Klassen von Throwable abgeleitet, von Exception wird die Klasse RuntimeException , sowie selbst erstellte Exceptions abgeleitet.
- Error und RuntimeException, sowie alle davon abgeleiteten Fehlerklassen bilden die unchecked Exceptions, alle anderen sind konzeptionell checked Exceptions.
- Checked Exceptions stellen Fehler dar, die während der Ausführung des Programms auftreten, wie z.B. Datei nicht gefunden oder Datei nicht lesbar und nicht unmittelbar zum Absturz des Programms führen

Java – Fehlerbehandlung II

- Unchecked Exceptions bilden Fehlerzustände ab, von denen ein Programm nicht mehr in einen konsistent operierenden Zustand gelangen kann, weswegen das Programm nur noch abgebrochen werden kann. Hierbei bilden die RuntimeExceptions Fehler ab, die auf Programmierfehler zurückzuführen sind, wie der Zugriff auf Bereiche außerhalb eines Array oder eine Division durch 0. Dies sind Fehler, die vor Auslieferung eines Programms behoben sein sollten. Errors bilden hauptsächlich Fehler ab, die auf Begrenzung der Hardware oder Fehler der Hardware beruhen, wie Stacküberlauf oder kein Speicher mehr vorhanden.

Java – Exceptions

- Eigene Exceptions erzeugt man indem man eine Klasse mit dem Namen der Exception anlegt und von der Klasse Exception ableitet.
- Via Exception erben alle Exceptions die Methoden von Throwable, wie beispielsweise printStackTrace() und von Object wie toString()
- Die Klasse Exception verfügt über zwei Konstruktoren, einen parameterlosen und einen mit einem String Parameter, der der Übergabe einer kurzen, aussagekräftigen Fehlernachricht dient.
- Im Konstruktor der eigenen Exceptionklasse sollte man, mittels super den Konstruktor der Oberklasse mit initialisieren und ihm die Fehlernachricht übergeben

Java - try-catch Block

- In einem try-catch Block wird Code abgehandelt, der potentiell "schiefgehen" und eine Exception auslösen kann.
- Vor Java 7 mußte einer try-Klausel, in der der eigentliche Programmcode ausgeführt wird ein oder mehrere catch-Klauseln und/ oder eine finally-Klausel folgen
- In den catch-Klauseln findet die Ausnahmebehandlung statt, beispielsweise Logging
- In der finally-Klausel kann Programmcode untergebracht werden, der unbedingt ausgeführt werden muß. Vor Java 7 war das auf jeden Fall immer das "saubere" Schließen von Ressourcen, z.B. offenen Dateien

Java – try-catch ab Java 7

- Java 7 bringt zwei Neuerungen in Bezug auf Exception Handling, das try-with-resource Statement und die multiple-catch Klausel
- try-with-resource
  - Ressourcen, wie Dateien, Streams, Netzwerkverbindungen können in einer parameterartigen Liste hinter dem try-Schlüsselwort geöffnet werden
  - Die einzelnen Ressourcen werden mit einem Semikolon getrennt
  - Unmittelbar nach Verlassen der try-Klausel werden die Ressourcen geschlossen ...
  - Dies funktioniert nur, wenn die Ressourcen das Interface java.lang.AutoCloseable implementiert
- multiple-catch
  - In einer catch-Klausel können nun mehrere Exceptions gleichartig behandelt werden
  - Dazu werden die einzelnen Exceptions in derParametrliste durch den |-Operator getrennt
  - ACHTUNG: Von einander abgeleitete Ausnahmen können nicht in einer Klausel abgheandelt werden

DESY

Java – Syntax: try-catch-Block

```
// try catch syntax
try{
    doDangerousThings();
}catch(PossibleException e){
    doSomethingWithTheThrownException();
}catch ... //catch other Exceptions

// try catch finally syntax
// code in the finally clause will be
// executed under all circumstances
try{
    doDangerousThings();
}catch(PossibleException e){
    doSomethingWithTheThrownException();
}finally{
    thingsHaveToBeDoneInEveryCase()
}

// try finally syntax
// could be used if the thrown exception
// can't be handled here but some resource
// has to be closed
try{
    doDangerousThings();
}finally{
    thingsHaveToBeDoneInEveryCase()
}
```


DESY

Java – Syntax: try-multi-catch-Block

```
// since Java 7 multiple exceptions could be handled
// in one catch clause but you can't handle exceptions
// in one catch clause if one inherit directly or
// indirectly from the other one
try{
    doDangerousThings();
}catch(PossibleExceptionOne | PossibleExceptionTwo e){
    doSomethingWithTheThrownException();
}finally{
    thingsHaveToBeDoneInEveryCase()
}
```

Java – Syntax: try-with-resource-Block

```
// try-with-resource-block since Java 7
// resources will be closed immediately after
// code execution in the try clause
try(ResourceTypeOne varOne = new ResourceTypeOne();
    ResourceTypeTwo varTwo = new ResourceTypeTwo()){
        doSomethingWithVarOne();
        doSomethingWithVarTwo();
}// at this point the resources are already closed
// if you didn't catch thrown exceptions the method
// must be declared as a throwing one

// try-with-resource-statement can have catch
// and try clauses
try(ResourceTypeOne varOne = new ResourceTypeOne();
    ResourceTypeTwo varTwo = new ResourceTypeTwo()){
        doSomethingWithVarOne();
        doSomethingWithVarTwo();
}catch(PossibleException e){
    doSomethingWithTheThrownException();
}finally{
    // Attention resources were already closed!
    thingsHaveToBeDoneInEveryCase()
}
```

Java – Exceptions, Beispiel

```java
/*
 * SomeSpookyThingsHappenedException.java
 */

package examples.exceptions;

public class SomeSpookyThingsHappenedException extends Exception{

    private final int errorState;

    public SomeSpookyThingsHappenedException(){
        this("Really spooky here!", -1); // call the 2 param const
    }

    public SomeSpookyThingsHappenedException(String message){
        this(message, -1); // call the 2 param const
    }

    public SomeSpookyThingsHappenedException(String message, int errorState){
        super(message);
        this.errorState = errorState;
    }

    public void printErrorState(){
        System.out.println(this.errorState);
    }

}
```


DESY

Java – throws and throw

- Mit dem Schlüsselwort **throws** müssen Methodensignaturen gekennzeichnet werden, die entweder eine Exception werfen oder unbehandelt eine Methode aufrufen die ihrerseits eine Exception wirft. Die Throws-Klausel steht hinter der eigentlichen Methodensignatur. Hier können mehrere Exceptions, kommasepariert, stehen.
- Mit dem Schlüsselwort **throw** wird bei Eintreten eines Fehlers die entsprechende Exception geworfen. Die Exception selbst wird hierbei als anonymes (namenloses, keine Referenz) Objekt direkt hinter der throw-Anweisung mit new erzeugt.

Java – throws und throw, Beispiel

```
/*
 * DangerousThingsDoingClass.java
 */

package examples.exceptions;

public class DangerousThingsDoingClass{

    public void doSomethingDangerous(int aNumber)
    throws SomeSpookyThingsHappenedException{
        if(aNumber == 666){
            throw new SomeSpookyThingsHappenedException
                ("There came a spooky number", aNumber);
        }else{
            System.out.println(aNumber);
        }
    }
}
```

Java – throws und throw, ohne Fehlerbehandlung

```java
/*
 * DangerousThingsUsingClass.java
 */

package examples.exceptions;

public class DangerousThingsUsingClass{

    public void useDangerousThings(){
        DangerousThingsDoingClass aClass =
             new DangerousThingsDoingClass();
        aClass.doSomethingDangerous(666);
    }

    public static void main(String[] args){
        DangerousThingsUsingClass dtuc =
             new DangerousThingsUsingClass();
        dtuc.useDangerousThings();
    }
}

// Ausgabe Compiler:

[:-(] [14:38:42]: /Users/dramburg/Code/Java/examples/exceptions
--> $ javac DangerousThingsUsingClass.java
DangerousThingsUsingClass.java:11: error: unreported exception
SomeSpookyThingsHappenedException; must be caught or declared to be thrown
        aClass.doSomethingDangerous(666);
                                   ^
1 error
```


DESY

Java – try-catch-Blöcke

- Um auftretende Fehler geordnet zu behandlen existiert in Java der **try-catch-Block**, in den Code, der Exceptions werfen kann eingeschlossen werden kann und in dem auf die einzelnen auftretenden Ausnahmen reagiert werden kann.
- In einem try-catch-Block können beliebig viele catch-Klauseln vorhanden sein, so viele als nötig für die Fehlerbehandlung sind.
- Ein try-catch-Block kann mit einer finally-Klausel abgeschlossen werden. Code innerhalb dieser Klausel wird immer ausgeführt, sowohl im Fehlerfall als auch im regulärenFall.

Java - try-catch-Blöcke, mit Fehlerbehandlung

```java
/*
 * DangerousThingsUsingClass.java
 */

package examples.exceptions;

public class DangerousThingsUsingClass{

    public void useDangerousThings(){
        DangerousThingsDoingClass aClass =
             new DangerousThingsDoingClass();
        try{
            aClass.doSomethingDangerous(666);
        }catch(SomeSpookyThingsHappenedException e){
            e.printErrorState();
            e.printStackTrace();
        }finally{
            System.out.println("Auf jeden Fall ausgef√ºhrt");
        }
    }

    public static void main(String[] args){
        DangerousThingsUsingClass dtuc =
             new DangerousThingsUsingClass();
        dtuc.useDangerousThings();
    }
}
```

Java - try-catch-Blöcke, mit Fehlerbehandlung Ausgabe

```
[:-)] [15:00:40]: /Users/dramburg/Code/Java/examples/exceptions
--> $ javac DangerousThingsUsingClass.java
[:-)] [15:01:39]: /Users/dramburg/Code/Java/examples/exceptions
--> $ java examples.exceptions.DangerousThingsUsingClass
666
examples.exceptions.SomeSpookyThingsHappenedException: There came a spooky number
  at examples.exceptions.DangerousThingsDoingClass.doSomethingDangerous(DangerousThingsDoingClass.java:12)
  at examples.exceptions.DangerousThingsUsingClass.useDangerousThings(DangerousThingsUsingClass.java:12)
  at examples.exceptions.DangerousThingsUsingClass.main(DangerousThingsUsingClass.java:24)
Auf jeden Fall ausgeführt
[:-)] [15:01:41]: /Users/dramburg/Code/Java/examples/exceptions
--> $
```


DESY

Java - Vererbung

- Mit Vererbung wird im Prinzip ein Prozess der Spezialisierung vom Allgemeinen bezeichnet. Je weiter Klassen in einer Klassenhierarchie oben stehen, umso allgemeiner sind die Eigenschaften und Methoden, je weiter eine Klasse in der Hierarchie unten steht, umso spezieller sind die implementierten Attribute.
- Im Wesentlichen "erben" Unterklassen die Eigenschaften und Methoden aller Klassen, die in der Klassenhierarchie in einer Vererbungslinie mit ihnen stehen.
- Attribute aus Oberklassen, können in von ihnen abgeleiteten Klassen überschrieben und somit eine andere Implementierung zur Verfügung stellen.
- Sind Attribute aus der Oberklasse überschrieben worden, kann mit dem Schlüsselwort **super** zugegriffen werden
- super ist eine Referenz auf die direkte Oberklasse, analog zu this

Java – statische Initialisierungsblöcke

- Statische Initialisierungsblöcke werden erzeugt mit static{ }
- Sie dienen der Initialisierung von statischen Variablen
- Deklaration muß vorher passieren
- Ausgeführt wird der Code beim Laden der Klasse
- Es kann nur auf statischen Klassenkontext zugegriffen werden, nicht auf Instanzvariablen und –methoden.
- Es kann mehrere statische Initialisierungsblöcke geben, sie werden in der Reihenfolge im Text ausgeführt.
- Empfiehlt sich wenn für die Initialisierung von statischen Variablen statische Methoden aufgerufen werden müssen, sonst einfacher bei der Deklaration.

Java – statische Initialisierungsblöcke, Beispiel

```
/*
 * StaticInititializerBlock.java
 */

package examples.blocks;

public class StaticInititializerBlock{

    public static int aNumber; //must be declared here

    static{
        aNumber = 5; //can be initialized here
        System.out.println("Within static block");
    }

    public static void main(String[] args){
        System.out.println("Within main");
        System.out.println(aNumber);
    }
}

// Ausgabe:
[:-)] [06:35:50]: /Users/dramburg/Code/Java/examples/blocks
--> $ javac StaticInititializerBlock.java
[:-)] [06:51:35]: /Users/dramburg/Code/Java/examples/blocks
--> $ java examples.blocks.StaticInititializerBlock
Within static block
Within main
5
```


DESY

Java - Initialisierungsblock

- Initialisierungsblöcke werden mit { } erzeugt
- Der Code im Block wird bei Anlegen eines Objekts, noch vor dem Aufruf des Konstruktors aufgerufen
- Alle hier definierten Variablen und Referenzen existieren im Kontext des Objekts
- Hilfreich sind Initialisierungsblöcke wenn eine Klasse über verschiedene Konstruktoren verfügt, aber bestimmte Initialisierungen für alle Konstruktoraufrufe gleich sein müssen

Java – Initialisierungsblock, Beispiele

```java
/*
 * InititializerBlock.java
 */

package examples.blocks;

public class InititializerBlock{
    private final int aNumber; // must be declared here

    {
        aNumber = 5; // can be initialized here
        System.out.println("Within block");
    }

    public InititializerBlock(){
        System.out.println("Within constructor");
    }

    public int getANumber(){
        return this.aNumber;
    }

    public static void main(String[] args){
        System.out.println("Within main()");
        InititializerBlock aBlockClass = new InititializerBlock();
        System.out.println(aBlockClass.getANumber());
    }
}

// Ausgabe:
[:-)] [07:09:28]: /Users/dramburg/Code/Java/examples/blocks
--> $ javac InititializerBlock.java
[:-)] [07:09:31]: /Users/dramburg/Code/Java/examples/blocks
--> $ java examples.blocks.InititializerBlock
Within main()
Within block
Within constructor
5
```


DESY

Java – Aufgabe Fortbewegungsmittel I

- Baut eine Klassenhierarchie für alle Arten von Fortbewegungsmitteln (MeanOfTransportation oder MeanOfTravel). Fragestellung hier ist, welche Eigenschaften haben alle Fortbewegungsmittel gemein und was sind die wesentlichen Dinge, die man mit ihnen anstellen kann. Fortbewegungsmittel sollen nicht instanzierbar sein.
- Unter der Oberklasse sollte eine weitere Spezialisierung in Fahrzeug (Vehicle), Schiff (Ship), Flugzeuge (Plane) und Raumschiff (Spaceship) erfolgen. Alle Methoden und Variablen, die hier schon Sinn machen, sollen auch hier implementiert werden. Auch diese Ebene soll nicht instanzierbar sein.
- Unter der Klasse Vehicle sollen weitere Klassen eine Spezialisierung zu Auto, PKW, LKW, Zweirad, Fahrrad, Motorrad, Tretroller, Pferd und Kutsche umsetzen. Fragen stellen sich hier zu einer geeigneten Hierarchie und den Wesensmerkmalen der einzelnen Fahrzeugtypen.

Java – Aufgabe Fortbewegungsmittel II

- Ähnliche Spezialisierungen wie für Fahrzeug sollen auch für Schiff (Segelschiff, Motorschiff, Ruderboot, etc. – lohnt die Unterscheidung in Boot und Schiff? Was wären die Unterscheidungsmerkmale?), Flugzeug (Hubschrauber, Zeppelin, Segelflugzeug, Flugzeug, etc.) und Raumschiff (Was wären hier brauchbare Unterscheidungsmerkmale? Rakete für Orbit, Schwerefeldüberwindung, etc. – <Fantasie ein> Stellar/Interstellarreisen, Warpantrieb, etc.</Fantasie aus>) verwirklicht werden.
- Alle Fahrzeuge verfügen über einen Antrieb (Propulsion), sei es nun ein Elektro- oder Verbrennungsmotor, ein Pedal, Wind, etc.
- Ein Antrieb ist ein Objekt, nicht instanzierbar und mit verschiedenen Spezialisierungen (s.o.). Jedes Fahrzeug hält eine Referenz auf den zu ihm passenden Antrieb.

Java – Aufgabe Fortbewegungsmittel III

- Damit ein Fortbewegunsmittel in Gang gesetzt werden kann, benötigt es einen Fahrer, Piloten, Steuermann, Kapitän, Astronauten, etc., dies alles sind Rollen, die von einer Person eingenommen werden können. Was sind die Vorraussetzungen für so eine Rolle, wie lässt sich so etwas modellieren?
- Viele der Fortbewegungsmittel, können zusätzlich Personen/ Passagiere und/oder Lasten befördern, allerdings nur bis zu einer gewissen Kapazität bzw. bis zu einem zulässigen Höchstgewicht.
- Müssen Fahrer und Mitfahrer schon von Anfang an im Fortbewegungsmittel vorhanden sein?
- Soll ein Fortbewegungsmittel bewegt werden und hat keinen Fahrer an Bord soll eine DriverNotIncludedException geworfen werden, für den Fall das ein Fahrer einsteigen möchte und bereits ein Fahrer im Fahrzeug ist soll eine DriverAlreadyIncludedException geworfen werden. Analog sollen Überschreiten von Kapazität und Höchstgewicht behandelt werden.

Java – Fortbewegungsmittel IV

- In allen Objekten, die Daten halten soll toString() überschrieben werden und den aktuellen Zustand des Objekts ausgeben, Fahrer, Passagiere, Fracht, wenn vorhanden.
- Jedes Fortbewegungsmittel verfügt über die notwendigen Methoden, um Personen und Fracht aufzunehmen und abzuladen, sowie in Bewegung gesetzt und angehalten zu werden. Darüber hinaus empfehlen sich Abfragen über den Bewegungszustand des Fortbewegungsmittel um Sicherzustellen, daß Personen nur aussteigen, wenn das Fahrzeug stillsteht.
- Wie muß Person erweitert werden, um den Anforderungen des Programms zu genügen?
- Es sollen zu jedem Zweig von Fortbewegungsmitteln 4 konkrete Instanzen erstellt werden, am besten in einem statischen Initialisierungsblock in der Klasse mit der main-Methode, dort sollen auch alle benötigten Personen und Antriebe erzeugt werden.

Java – Fortbewegungsmittel V

- In der main-Methode sollen dann für die verschiedenen Fortbewegungsmittel, Personen und Lasten eingeladen, Fahrzeuge in Bewegung gesetzt und angehalten werden und der jeweilige Zustand des Fortbewegungsmittels ausgegeben werden. Jeder mögliche Fehlerzustand soll einmal ausgegeben werden.
- Ist es sinnvoll die verschiedenen Teile in verschiedene packages bzw. sub-packages zu verlagern? Warum?

Java - Strings

- Strings sind in Java Immutable ...
- ... können deshalb sehr gut als Schlüssel in Maps eingesetzt werden
- Werden mit doppelten Anführungszeichen als "Konstante" angelegt, ähnlich wie in C, aber ...
- ... werden als separates Objekt mit new String("A String") erzeugt
- Sind intern UTF-16 codiert (in der Ausgabe UTF-8!)

Java – java.lang.String

- Implementiert u.a. die Interfaces Serializable und Comparable<String>
- Ist als final class angelegt -> keine Vererbung!
- Bietet viele Konstruktoren um Strings aus Byte- und Char-Arrays, sowie aus String (Copy-Konstruktor), StringBuffer und StringBuilder zu erzeugen
- Enthält viele Methoden zur Arbeit mit Strings, wie charAt(), startsWith(), endsWith(), format(), indexOf(), length(), split(), trim() und vieles mehr
- Außerdem findet man statische Methoden zur Umwandlung von primitiven Datentypen in Strings: valueOf()

Java – Probleme bei der Konkatenation von Strings

- In der Oracle Javadoc steht: "The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. String concatenation is implemented through the StringBuilder(or StringBuffer) class and its append method."
- ...das stimmt aber nur teilweise...
- .... nämlich nur für statische Zeichenketten. Werden in der Konkatenation noch Ausdrücke ausgewertet funktioniert der Automatismus nicht (Teure Operation!). Besser: Verwendung von StringBuffer oder StringBuilder

Java – String-Konkatenation, Beispiel mit +

```java
/*
 * StringConcatenation.java
 */
package examples.strings;

public class StringConcatenation{

    public long lazyConcatMeasurement(int loopCounter){
        final long startTime = System.currentTimeMillis();
        String s = "loopCounter: " + loopCounter + "\n";
        for(int i = 0; i < loopCounter; i++){
            s += ", loop_" + i + ": " + ((loopCounter - i)/4) + "\n";
        }
        long runTime = System.currentTimeMillis() - startTime;
        //System.out.println(s);
        System.out.println("Lazy runtime: " + runTime);
        return runTime;
    }
```


DESY

Java – String-Konkatenation, Beispiel mit StringBuffer

```java
public long betterConcatMeasurement(int loopCounter){
    final long startTime = System.currentTimeMillis();
    StringBuffer sb = new StringBuffer();
    sb.append("loopCounter: ");
    sb.append(loopCounter);
    sb.append("\n");
    for(int i = 0; i < loopCounter; i++){
        sb.append(", loop_");
        sb.append(i);
        sb.append(": ");
        sb.append((loopCounter - i)/4);
        sb.append("\n");
    }
    String s = sb.toString();
    long runTime = System.currentTimeMillis() - startTime;
    //System.out.println(s);
    System.out.println("Better runtime: " + runTime);
    return runTime;
}
```

Java – String-Konkatenation, Beispiel Aufruf

```
    public static void main(String[] args){
        StringConcatenation sc = new StringConcatenation();
        int repeats = 10;
        int loops = 10000; //auch mal mit 25000, 1000000 laufen lassen!
        long meanLazy = 0;
        long meanBetter = 0;
        for(int i = 0; i < repeats; i++){
            meanLazy += sc.lazyConcatMeasurement(loops);
        }
        for(int i = 0; i < repeats; i++){
            meanBetter += sc.betterConcatMeasurement(loops);
        }
        System.out.println("Lazy mean value: " + meanLazy/repeats);
        System.out.println("Better mean value: " + meanBetter/repeats);
    }

}
```

Ausgabe für loops=10000:

```
[:-)] [14:03:41]: /Users/dramburg/Code/Java/examples/strings
--> $ java examples.strings.StringConcatenation
Lazy runtime: 1154
Lazy runtime: 803
...
Lazy runtime: 785
Lazy runtime: 773
Better runtime: 3
Better runtime: 3
...
Better runtime: 1
Better runtime: 1
Lazy mean value: 833
Better mean value: 1
```

Java – java.lang.StringBuffer

- Seit Java 1.0
- "Mutable String"
- Thread-Safe, durch Synchronisierung
- Ähnliche Schnittstelle wie String, zusätzlich append() um am Ende des StringBuffer etwas hinzuzufügen, bzw. insert() um an beliebiger Stelle etwas einzufügen.

Java - StringBuilder

- Seit Java 1.5
- "Mutable String"
- NICHT Thread-Safe, keine Synchronisierung...
- ... dadurch nochmal etwas schneller als StringBuffer
- Benutzt selbe Schnittstelle wie StringBuffer und sollte in Single-Thread-Anwendungen bevorzugt genommen werden

Java – I/O Grundlagen

- Reader/Writer/PrintWriter
- BufferedReader/BufferedWriter
- InputStream/OutputStream/PrintStream
- BufferedInputStream/BufferedOutputStream
- ByteChannel
- FileChannel
- Scanner/Console
- Path/Files
- OpenOptions/StandardOpenOptions/LinkOptions

DESY

Java – I/O, System.in und System.out

- Die Java "I/O-Klassiker" zur Ausgabe auf der Standardein- und Standardausgabe, sowie der Standardfehlerausgabe der Kommandozeile über die statischen Streams der Klasse java.lang.System:
  - System.in – statischer InputStream
  - System.out – statischer PrintStream
  - System.err – statischer PrintStream
- Auf die Streams kann über die Methoden read() bzw. print()/println() zugegriffen werden

Java – I/O: java.io.Console seit Java 1.6

- Seit Java 6 existiert die Klasse java.io.Console als Repräsentation der das Programm aufrufenden Systemkonsole.
- An die Console kommt man über einen Aufruf von System.console()
- Pro Programm existiert maximal ein Console-Objekt, und das auch nur wenn das Programm via Systemkonsole, also z.B. ein Terminal gestartet wurde -> ALSO: muß vor Benutzung des Console-Objekts geprüft werden, ob der Aufruf von console() nicht eine null-Referenz liefert.
- Via Console kann lesend (readLine()/readPassword()) und schreibend (printf()/format()) auf die Systemkonsole zugegriffen werden

Java – I/O: java.io.Console, Beispiel

```
/*
 * ConsoleExample.java
 */
package examples.io;

import java.io.Console;

public class ConsoleExample{

    public static void main(String[] args){
        Console console = System.console();

        if(console == null){
            System.err.println("Found no Console object!");
            System.exit(1);
        }
        console.printf("%s", "a string");
    }
}
```

Java – File-I/O, Grundlagen

- Dateioperationen sind immer kritisch, d.h. sie gehören in einen try-catch-Block mit sinnvoller Ausnahmebehandlung
- Die an den Dateioperationen beteiligten Objekte wie Reader, Writer, Streams, etc. sollten nach Beendigung der Operationen durch Aufruf von close() auf dem betreffenden Objekt sauber geschlossen werden (vor Einführung des try-with-ressource-Blocks!)
- close() sollte in der finally-Klausel des try-catch-Blocks ausgeführt werden und ist selbst eine kritische Dateioperation, die in einen try-catch-Block gehört. Selbes gilt auch für den Aufruf von flush() bei Schreiboperationen (vor Einführung des try-with-ressource-Blocks!)
- Für Writer/OutputStream-Klassen ruft close() laut API vorher flush() auf, bei hohem Paranoia-Level kann man das auch selber machen.

Java – I/O, Dateien lesen vor Java 5

- Mit einer Kombination aus BufferedReader und FileReader
  - BufferedReader ist eine sehr allgemeiner Reader, der aus einem Zwischenspeicher, dem Buffer Inhalte liest
  - FileReader ist eine sogenannte "Decoration" für BufferedReader, also eine Erweiterung der Funktionalität des BufferedReader um die Fähigkeit eine Datei zu öffnen und deren Inhalt auszulesen
- Mit Streams in einer Kombination von DataInputStream, BufferedInputStream und FileInputStream
  - DataInputStream verarbeitet beliebige Datenquellen
  - Mit den Decorations BufferedInputStream (Zwischenspeichern des Datenstroms) und FileInputStream (File Handling) wird dies zu einer Möglichkeit Dateien einzulesen

Java – I/O, Dateien lesen vor Java 5, Beispiel I

```
private final String fileName = "/path/to/file";

public void readAndPrintFile(){
    BufferedReader bReader = null;
    FileReader fReader = null;
    try{
        fReader = new FileReader(fileName);
        bReader = new BufferedReader(fReader);
        String line;
        while((line = bReader.readLine()) != null){
            System.out.println(line);
        }
    }catch(IOException e){
        e.printStackTrace();
    }finally{
        try{
            fReader.close();
            bReader.close();
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

Java – I/O, Dateien lesen vor Java 5, Beispiel II

```
private final String fileName = "/path/to/file";

public void readAndPrintFile(){
    DataInputStream input = null;
    StringBuffer buffer = new StringBuffer();
    try{
        input = new DataInputStream(
                    new BufferedInputStream(
                        new FileInputStream(this.fileName)));
        String line;
        while((line = input.readLine()) != null){
            buffer.append(line);
            buffer.append("\n");
        }
        System.out.println(buffer.toString());
    }catch(IOException e){
        e.printStackTrace();
    }finally{
        try{
            input.close();
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

Java – I/O, Dateien lesen ab Java 5

- Ab Java 5 kommt eine weitere Möglichkeit hinzu kleine Textdateien unkompliziert einzulesen, mit Hilfe der Klassen java.util.Scanner und java.io.File
- Scanner ist eigentlich als simple Text-Parser Klasse entworfen worden und kann auf Strings, Dateien, Streams und Channels arbeiten
- Scanner kann reguläre Ausdrücke zum Parsen verwenden ...
- ... und ganz einfach Textdateien lesen

Java – I/O, Dateien lesen ab Java 5, Beispiel

```java
/*
 * ReadFileJava5.java
 * Java File Reading with Scanner
 * Scanner closes automatically, no finally needed here!
 */
package examples.io;

import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class ReadFileJava5{
    private final String fileName = "/path/to/file";

    public void readAndPrintFile(){
        try{
            Scanner scanner = new Scanner(new File(fileName));
            while(scanner.hasNext()){
                System.out.println(scanner.nextLine());
            }
        }catch(IOException e){
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        ReadFileJava5 fileReader = new ReadFileJava5();
        fileReader.readAndPrintFile();
    }
}
```

Java – I/O, Dateien lesen ab Java 7

- Ab Java 7 kommt die Möglichkeit hinzu einem try-catch-Block Ressourcen zum automatisierten Handling zu überantworten. Dies können beispielsweise die an der I/O beteiligten Objekte sein. Sie werden ähnlich einer Methoden-Parameterliste dem Block vorangestellt.
- Oracle empfiehlt auf seinen Tutorialseiten eine geschachtelte Ressourceneinbindung (Decorator!), da die meisten Ressourcen das AutoCloseable-Interface einbinden und also auch bei Benutzung als Decorator automatisch geschlossen werden.
- Bei hohem Paranoia-Level empfiehlt sich die einzelne Einbindung der Ressourcen, da nur so für jeden Fall sicher gestellt ist, daß die Ressourcen auch wirklich automatisch geschlossen werden.

Java – I/O, Dateien lesen ab Java 7, Version Literatur

```java
/*
 * ReadFileJava7.java
 * Ressource handling with try-with-ressource
 * as in common literature
 */
package examples.io;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFileJava7{
    private final String fileName = "/path/to/file";

    public void readAndPrintFile(){
        try(BufferedReader bReader =
                new BufferedReader(new FileReader(fileName))){
            String line;
            while((line = bReader.readLine()) != null){
                System.out.println(line);
            }
        }catch(IOException e){
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        ReadFileJava7 reader = new ReadFileJava7();
        reader.readAndPrintFile();
    }
}
```


DESY

Java – I/O, Dateien lesen ab Java 7, Version hohe Paranoia

```java
/*
 * ReadFileJava7paranoid.java
 * Ressource handling with high paranoia level
 */
package examples.io;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFileJava7paranoid{
    private final String fileName = "/path/to/file";

    public void readAndPrintFile(){
        try(
            FileReader fReader = new FileReader(fileName);
            BufferedReader bReader = new BufferedReader(fReader)){

            String line;
            while((line = bReader.readLine()) != null){
                System.out.println(line);
            }
        }catch(IOException e){
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        ReadFileJava7paranoid reader = new ReadFileJava7paranoid();
        reader.readAndPrintFile();
    }
}
```

Java – I/O, Dateien lesen ab Java 8

- Ab Java8 lassen sich Dateien mit Hilfe der Klassen Files und Paths aus dem package java.nio.file einlesen
- Gleichzeitig können die Datenströme mit Hilfe der funktionalen Interfaces und den default-Implementierungen in den Interfaces Stream und Collectors aus java.util.stream vor der Ausgabe weiterverarbeitet werden. Zur Verfügung stehen Methoden zum Suchen (*findAny(), findFirst()*), Filtern (*filter()*), Veringerung der Elemente (*reduce()*), Sammeln und Speichern (*collect()*), zur Übergabe jedes Elements an eine Methode zur Weiterverarbeitung (*map()*), etc.

Java – I/O, Dateien lesen ab Java 8, Beispiele

```
private final String fileName = "/path/to/file";
private List<String> list = new ArrayList<String>();

public void readAndPrintFile(){
    try(Stream<String> stream =
            Files.lines(Paths.get(this.fileName))){
        stream.forEach(System.out::println);
    }catch(IOException e){
        e.printStackTrace();
    }
}

public void readAndPrintFileWithList(){
    try(Stream<String> stream =
            Files.lines(Paths.get(this.fileName))){
        list = stream.collect(Collectors.toList());
    }catch(IOException e){
        e.printStackTrace();
    }
    list.forEach(System.out::println);
}

public void readFilterAndPrintFileWithList(){
    try(Stream<String> stream =
            Files.lines(Paths.get(this.fileName))){
        list = stream
            .filter(line -> line.startsWith("import"))
            .collect(Collectors.toList());
    }catch(IOException e){
        e.printStackTrace();
    }
    list.forEach(System.out::println);
}
```

Java – I/O, Dateien schreiben vor Java 7

- Möglich über Writer und OutputStream, sowohl gepuffert als auch ungepuffert
- Die Buffered-Varianten sind meistens schneller, da sie erst schreiben wenn der Buffer voll ist (Schreiben in Dateien == Teure Operation!)
- Die ungepufferten Varianten schreiben sofort (oder versuchen dies zumindest)
- Bei den Buffered-Varianten muß darauf geachtet werden den Buffer vor dem Schließen der Ressource mit einem Aufruf von flush() explizit zu leeren und etwaigen noch vorhandenen Inhalt auf die Platte zu schreiben.

Java – I/O, Dateien schreiben vor Java 7, Beispiel Writer

```java
/*
 * WriteFileOldJava.java
 * Attention! FileWriter uses the system default
 * character encoding, if this is not what you want
 * use an OutputStreamWriter which can use other encodings
 */
package examples.io;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.FileNotFoundException;

public class WriteFileOldJava{
   private final String fileName =
       "/Users/dramburg/tmp/writtenFile_2.txt";

   public void writeStringToFile(String aString){
       BufferedWriter bWriter = null;
       try{
           bWriter = new BufferedWriter(new FileWriter(fileName));
           bWriter.write(aString);
       }catch(FileNotFoundException e){
           e.printStackTrace();
       }catch(IOException e){
           e.printStackTrace();
       }finally{
           try{
               bWriter.close();
           }catch(IOException e){
               e.printStackTrace();
           }
       }
   }
```

Java – I/O, Dateien schreiben vor Java 7, Beispiel Writer

```
    public void writeStringToFileParanoidVersion(String aString){
        BufferedWriter bWriter = null;
        FileWriter fWriter = null;
        try{
            fWriter = new FileWriter(fileName);
            bWriter = new BufferedWriter(fWriter);
            bWriter.write(aString);
        }catch(FileNotFoundException e){
            e.printStackTrace();
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            try{
                bWriter.flush();
                bWriter.close();
                fWriter.close();
            }catch(IOException e){
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args){
        WriteFileOldJava writer = new WriteFileOldJava();
        writer.writeStringToFile("Dont panic! It's only from Earth," +
                " so it will be mostly harmless!\n");
    }
}
```


DESY

Java – I/O, Dateien schreiben vor Java 7, Beispiel Stream

```
public class WriteFileWithStreamOldJava{
   private final String fileName =
         "/Users/dramburg/tmp/writtenFile_1.txt";

   public void writeStringToFile(String aString){
       DataOutputStream output = null;
       byte[] stringAsBytes = aString.getBytes();
       int i = 0;
       try{
           output = new DataOutputStream(
               new BufferedOutputStream(
                   new FileOutputStream(this.fileName)));

           while(i < stringAsBytes.length){
               output.writeByte(stringAsBytes[i++]);
           }
       }catch(FileNotFoundException e){
           e.printStackTrace();
       }catch(IOException e){
           e.printStackTrace();
       }finally{
           try{
               output.flush();
               output.close();
           }catch(IOException e){
               e.printStackTrace();
           }
       }
   }
```

Java – I/O, Dateien schreiben ab Java 7

- Ab Java 7 wird das Exception Handling durch das try-with-resource Statement etwas einfacher
- Ressourcen die das AutoCloseable Interface implementieren sichern zu, die Ressource automatisch beim Verlassen der try-Klausel zu schließen und gepufferte Inhalte vorher durch einen internen Call von flush zu leeren.

Java – I/O, Dateien schreiben ab Java 7, Beispiel

```
import java.io.Writer;
import java.io.BufferedWriter;
import java.io.OutputStreamWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.FileSystemException;
import java.io.UnsupportedEncodingException;

public class WriteFileJava7{
    private final String fileName =
       "/Users/dramburg/tmp/writtenFile_3.txt";

    public void writeStringToFile(String aString){
        try(Writer writer = new BufferedWriter(
                    new OutputStreamWriter(
                    new FileOutputStream(fileName), "utf-8"))){
            writer.write(aString);
        }catch(FileNotFoundException
          | FileSystemException
          | UnsupportedEncodingException e){
            e.printStackTrace();
        }catch(IOException e){ // IOException is the base class
            e.printStackTrace(); // it has to use a separate catch clause
        }
    }

    public static void main(String[] args){
        WriteFileJava7 writer = new WriteFileJava7();
        writer.writeStringToFile("So long, and thanks for all the fish!");
    }
}
```


DESY

Java – I/O, Dateien schreiben ab Java 8

- Ab Java 8 lassen sich Schreiboperationen mit Hilfe der Klassen Files und Paths aus dem Paket java.nio.file noch einfacher und flexibler gestalten (quasi Einzeiler)

Java – I/O, Dateien schreiben ab Java 8, Beispiel

```
/*
 * WriteFileJava8.java
 */
package examples.io;

import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;
import java.nio.file.StandardOpenOption;
import java.util.List;
import java.util.Arrays;

public class WriteFileJava8{
    private final String fileName =
            "/Users/dramburg/tmp/writtenFile_5.txt";
    private final Charset utf8 = StandardCharsets.UTF_8;
    private final List<String> lines =
            Arrays.asList("Hello and good bye!", "Last line.");

    public void writeStringToFile(){
        try{
            Files.write(Paths.get(fileName), lines, utf8,
                StandardOpenOption.CREATE,
                StandardOpenOption.APPEND);
        }catch(IOException e){
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        WriteFileJava8 writer = new WriteFileJava8();
        writer.writeStringToFile();
    }
}
```

Java – Kopien von Objekten und Objektreferenzen

- Während bei primitiven Datentypen die Kopie eines Werts kein Problem darstellt:
  - int i = 3;
    int j = i; // hier haben sowohl j, als auch i den Wert 3 ...
    j = 5; // ... und hier verändert nur j den Wert
- ... ist dies bei Objekten nicht ganz so einfach, da hier nur Referenzen über den Variablennamen angesprochen werden und nicht der Wert (oder besser: der Wert ist hier eine Speicheradresse!):
  - ObjectA anObject = new ObjectA(13);
    ObjectA anotherObject = anObject; // beide Variablen zeigen auf das gleiche Objekt
    anotherObject.setValue(23); // bei beiden Referenzen hat sich der Wert im
    // zugehörigen Objekt geändert
- Diese Referenzkopien eines Objekts nennt man Shallow Copy, was muß man machen, um eine "echte", tiefe Kopie, eine sogenannte Deep Copy zu erhalten?

Java – Kopien von Objektreferenzen, Beispiel

```java
/*
 * CopyTest.java
 */
package examples.deepcopy;

class ObjectA{
    private int aNumber;

    public ObjectA(int aNumber){this.aNumber = aNumber;}
    public int getANumber(){return aNumber;}
    public void setANumber(int aNumber){this.aNumber = aNumber;}
}

public class CopyTest{

    public static void main(String[] args){
        int aNumber = 13;
        int anotherNumber = aNumber;
        System.out.println("aNumber: " + aNumber + ", anotherNumber: " + anotherNumber);
        anotherNumber = 23;
        System.out.println("aNumber: " + aNumber + ", anotherNumber: " + anotherNumber);
        ObjectA anObject = new ObjectA(2);
        ObjectA anotherObject = anObject;
        System.out.println("anObject: " + anObject.getANumber()
                           + ", anotherObject: " + anotherObject.getANumber());
        anotherObject.setANumber(42);
        System.out.println("anObject: " + anObject.getANumber()
                           + ", anotherObject: " + anotherObject.getANumber());
    }
}
// Ausgabe:
--> $ java examples.deepcopy.CopyTest
aNumber: 13, anotherNumber: 13
aNumber: 13, anotherNumber: 23
anObject: 2, anotherObject: 2
anObject: 42, anotherObject: 42
```


DESY

Java – Objektkopie, Lösung 1 – Copy-Konstruktor

- Man spendiert allen Klassen von denen man eine Deep Copy zulassen will einen sogenannten Copy-Konstruktor, also einen Konstruktor dessen einziger Zweck die Anfertigung einer genauen Kopie des übergebenen Objekts ist.
- Der Copy-Konstruktor hat nur einen Parameter, ein Objekt der implementierenden Klassen.
- Aus dem übergebenen Objekt werden alle primitiven Datentypen in das neu erzeugte Objekt kopiert. Hält das übergebene Objekt Referenzen auf andere Objekte, entscheidet das weitere Vorgehen darüber, was für eine Kopie des Objekts man erhält. Kopiert man die Objektreferenzen lediglich, erhält man eine "flache" Kopie, eine sogenannte Shallow Copy. Ruft man hingegen für alle in der Tiefe vorhandenen Objektreferenzen deren Copy-Konstruktoren auf erhält man eine sogenannte Deep Copy, also eine echte, vom Original unabhängige Kopie. Dies kann im Einzelfall sehr aufwendig und teuer werden, ist aber für bestimmte Fälle unerlässlich, beispielsweise für ein Backup.

DESY

Java – Copy-Konstruktor Deep Copy, Beispiel

```
package examples.deepcopy;

class ObjectA{
    private int aNumber;
    private ObjectB anObjectB;

    public ObjectA(int aNumber, ObjectB anObjectB){
        this.aNumber = aNumber;
        this.anObjectB = anObjectB;
    }
    public ObjectA(ObjectA anotherObjectA){
        this.aNumber = anotherObjectA.getANumber();
        this.anObjectB = anotherObjectA.getAnObjectB();
    }
    public int getANumber(){return this.aNumber;}
    public void setANumber(int aNumber){this.aNumber = aNumber;}
    public ObjectB getAnObjectB(){return this.anObjectB;}
    public void setAnObjectB(ObjectB anObjectB){
        this.anObjectB = anObjectB;
    }
    @Override
    public String toString(){
        return "ObjectA: aNumber=" + this.aNumber
               + ", " + this.anObjectB;
    }
}

class ObjectB{
    private String aName;

    public ObjectB(String aName){this.aName = aName;}
    public String getAName(){return aName;}
    public void setAName(String aName){this.aName = aName;}
    @Override
    public String toString(){
        return "ObjectB: aName=" + this.aName;
    }
}
```

Java – Copy-Konstruktor Deep Copy, Beispiel

```
public class CopyCtorDeep{

    public static void main(String[] args){
        ObjectB anObjectB = new ObjectB("Gaston");
        ObjectA anObjectA = new ObjectA(23, anObjectB);
        ObjectA anotherObjectA = new ObjectA(anObjectA);
        System.out.println(anObjectA);
        System.out.println(anotherObjectA);
        anObjectA.setANumber(42);
        System.out.println(anObjectA);
        System.out.println(anotherObjectA);
        ObjectB anotherObjectB = new ObjectB("Marsupilami");
        anObjectA.setAnObjectB(anotherObjectB);
        System.out.println(anObjectA);
        System.out.println(anotherObjectA);
    }
}
```

Ausgabe:

```
[:-)] [12:21:09]: /Users/dramburg/Code/Java/examples/deepcopy
--> $ java examples.deepcopy.CopyCtorDeep
ObjectA: aNumber=23, ObjectB: aName=Gaston
ObjectA: aNumber=23, ObjectB: aName=Gaston
ObjectA: aNumber=42, ObjectB: aName=Gaston
ObjectA: aNumber=23, ObjectB: aName=Gaston
ObjectA: aNumber=42, ObjectB: aName=Marsupilami
ObjectA: aNumber=23, ObjectB: aName=Gaston
```

Java – Lösung 2, das Cloneable Interface und clone()

- Java unterstützt das Klonen, also Anfertigen von Objektkopien über die Methode clone() aus java.lang.Object und dem Implementieren des Interface java.lang.Cloneable.
- Die default-Implementierung aus Object erzeugt ein neues Objekt mit "stumpf" kopierten Attributfeldern, da es ja über die Klassen unterhalb von Object nichts "weiß". Für Klassen, die nur primitive Daten enthalten ist dies ok, für Objektattribute hingegen erhält man allerdings wieder nur eine Kopie der Referenz, also eine flache Kopie. Für eine Deep Copy muß die clone()-Methode in den Unterklassen überschrieben werden.
- Dabei sollten folgende Bedingungen eingehalten werden:
  - original != clone
  - original.getClass() == clone.getClass()
  - original.equals(clone)

Java – Cloneable und clone(), Beispiel

```
package examples.deepcopy;

class ObjectA implements Cloneable{
    private int aNumber;
    private ObjectB anObjectB;

    public ObjectA(int aNumber, ObjectB anObjectB){
        this.aNumber = aNumber;
        this.anObjectB = anObjectB;
    }
    public int getANumber(){return this.aNumber;}
    public void setANumber(int aNumber){this.aNumber = aNumber;}
    public ObjectB getAnObjectB(){return this.anObjectB;}
    public void setAnObjectB(ObjectB anObjectB){
    this.anObjectB = anObjectB;
    }
    @Override
    public String toString(){
        return "ObjectA: aNumber=" + this.aNumber
                + ", " + this.anObjectB;
    }
    @Override
    public Object clone() throws CloneNotSupportedException{
        ObjectA cloned = (ObjectA)super.clone();
        cloned.setAnObjectB((ObjectB)this.anObjectB.clone());
        return cloned;
    }
}
```


DESY

Java – Cloneable und clone(), Beispiel

```
class ObjectB implements Cloneable{
    private String aName;

    public ObjectB(String aName){this.aName = aName;}
    public String getAName(){return aName;}
    public void setAName(String aName){this.aName = aName;}
    @Override
    public String toString(){
        return "ObjectB: aName=" + this.aName;
    }
    @Override
    public Object clone() throws CloneNotSupportedException{
        ObjectB cloned = (ObjectB)super.clone();
        return cloned;
    }
}

public class CloneDeep{

    public static void main(String[] args){
        ObjectB anObjectB = new ObjectB("Gaston");
        ObjectA anObjectA = new ObjectA(23, anObjectB);
        ObjectA anotherObjectA = null;
        try{
            anotherObjectA = (ObjectA)anObjectA.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
```


DESY

Java – Cloneable und clone(), Beispiel

```
        System.out.println(anObjectA);
        System.out.println(anotherObjectA);
        anObjectA.setANumber(42);
        ObjectB anotherObjectB = new ObjectB("Marsupilami");
        anObjectA.setAnObjectB(anotherObjectB);
        System.out.println(anObjectA);
        System.out.println(anotherObjectA);
    }
}
```

Ausgabe:

```
[:-)] [13:09:20]: /Users/dramburg/Code/Java/examples/deepcopy
--> $ javac CloneDeep.java
[:-)] [13:09:46]: /Users/dramburg/Code/Java/examples/deepcopy
--> $ java examples.deepcopy.CloneDeep
ObjectA: aNumber=23, ObjectB: aName=Gaston
ObjectA: aNumber=23, ObjectB: aName=Gaston
ObjectA: aNumber=42, ObjectB: aName=Marsupilami
ObjectA: aNumber=23, ObjectB: aName=Gaston
```

Java – Cloneable und clone(), Beispiel mit Tests

- Überschreibt man in den beiden Klassen ObjectA und ObjectB noch die equals() Methode, lässt sich Überprüfen, ob die Bedingungen für das Klonen eingehalten wurden:

```
// ObjectA
@Override
public boolean equals(Object object){
    if(object == null){
        return false;
    }
    if(this.getClass() != object.getClass()){
        return false;
    }
    if(this == object){
        return true;
    }
    ObjectA other = (ObjectA)object;
    return this.aNumber == other.getANumber()
        && this.anObjectB.equals(other.getAnObjectB());
}
```

Java – Cloneable und clone(), Beispiel mit Tests

```
// ObjectB
@Override
public boolean equals(Object object){
    if(object == null){
        return false;
    }
    if(this.getClass() != object.getClass()){
        return false;
    }
    if(this == object){
        return true;
    }
    ObjectB other = (ObjectB)object;
    return this.aName.equals(other.getAName());
}
// ...in der main:
System.out.println("Testing fullfillment of cloning conditions:");
System.out.println("Original != Clone:                          "
    + (anObjectA != anotherObjectA));
System.out.println("Original.getClass() == Clone.getClass(): "
    + (anObjectA.getClass() == anotherObjectA.getClass()));
System.out.println("Original.equals(Clone):                     "
    + anObjectA.equals(anotherObjectA));

Ausgabe:
--> $ java examples.deepcopy.CloneDeepWithTests
Testing fullfillment of cloning conditions:
Original != Clone:                          true
Original.getClass() == Clone.getClass(): true
Original.equals(Clone):                     true
```

Java – Serialisierung / Deserialisierung

- Serialisierung meint im wesentlichen binäre Daten, hier initialisierte Objekte in eine textuelle Form zu überführen. Dies meistens im Zusammenhang mit dem Speichern eines Programmzustands oder Datenbestands, wie beispielweise beim Anlegen eines Backups oder Speichern eines Spielstandes. Deserialisierung meint somit die Umkehrung des Prozesses, also das Einlesen gespeicherter Daten oder Objekte.
- Java bietet hierzu im Paket java.io die Klassen ObjectOutputStream und ObjectInputStream, die neben einer writeObject() bzw. readObject() jede Menge anderer Methoden zum Lesen bzw. Schreiben von primitiven Datentypen bieten.
- Benötigen Objekte spezielle Operationen zur Serialisierung/ Deserialisierung müssen die entsprechenden Klassen die Methoden readObject(), writeObject() und readObjectNoData(), mit den genauen in der Javadoc genannten Signaturen implementieren.

Java – Serialisierung / Deserialisierung

- Alle Klassen, die Serialisiert werden sollen, müssen das Markerinterface java.io.Serializable implementieren
- Meistens ist es nützlich den serialisierbaren Klassen einen parameterlosen Konstruktor mitzugeben
- Im Beispiel wird eine Klasse Person durch ein kleines Programm erst serialisiert und dann durch ein zweites Programm wieder deserialisiert.

Java – Serializable class Person

```java
/*
 * Person.java
 */
package examples.serialize;

import java.io.Serializable;

public class Person implements Serializable{
    private final String firstName;
    private final String lastName;
    private final int age;

    public Person(String firstName, String lastName, int age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    public Person(){
        this("Max", "Mustermann", 99);
    }
    public String getFirstName(){return this.firstName;}
    public String getLastName(){return this.lastName;}
    public int getAge(){return this.age;}
}
```

Java – Objektserialisierung mit ObjectOutputStream

```
/*
 * SerializedPerson.java
 */
package examples.serialize;

import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class SerializedPerson{

    public static void main(String[] args){
        String storage = "/Users/dramburg/tmp/persons.data";
        Person p1 = new Person("Pippi", "Langstrumpf", 9);
        Person p2 = new Person("Karlsson", "vom Dach", 14);
        try(ObjectOutputStream output = new ObjectOutputStream(
            new FileOutputStream(storage))){
                output.writeObject(p1);
                output.writeObject(p2);
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

Java – Objektdeserialisierung mit ObjectInputStream

```
/*
 * DeSerializedPerson.java
 */
package examples.serialize;

import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class DeSerializedPerson{

    public static void main(String[] args){
        String storage = "/Users/dramburg/tmp/persons.data";
        Person p1 = null;
        Person p2 = null;
        try(ObjectInputStream input = new ObjectInputStream(
            new FileInputStream(storage))){
                p1 = (Person)input.readObject();
                p2 = (Person)input.readObject();
        }catch(IOException | ClassNotFoundException e){
            e.printStackTrace();
        }
        System.out.println(p1.getFirstName() + " "
                           + p1.getLastName() + ", " + p1.getAge());
        System.out.println(p2.getFirstName() + " "
                           + p2.getLastName() + ", " + p2.getAge());
    }
}
```


DESY

Java – Objektserialisierung, Beispiel Ausgabe

```
[:-)] [16:33:49]: /Users/dramburg/Code/Java/examples/serialize
--> $ java examples.serialize.SerializedPerson
[:-)] [16:33:56]: /Users/dramburg/Code/Java/examples/serialize
--> $ java examples.serialize.DeSerializedPerson
Pippi Langstrumpf, 9
Karlsson vom Dach, 14
[:-)] [16:34:11]: /Users/dramburg/Code/Java/examples/serialize
--> $ file /Users/dramburg/tmp/persons.data
/Users/dramburg/tmp/persons.data: Java serialization data, version 5
[:-)] [16:35:45]: /Users/dramburg/Code/Java/examples/serialize
--> $ xxd /Users/dramburg/tmp/persons.data
0000000: aced 0005 7372 0019 6578 616d 706c 6573  ....sr..examples
0000010: 2e73 6572 6961 6c69 7a65 2e50 6572 736f  .serialize.Perso
0000020: 6e6e 040c c6a5 5c54 a102 0003 4900 0361  nn....\T....I..a
0000030: 6765 4c00 0966 6972 7374 4e61 6d65 7400  geL..firstNamet.
0000040: 124c 6a61 7661 2f6c 616e 672f 5374 7269  .Ljava/lang/Stri
0000050: 6e67 3b4c 0008 6c61 7374 4e61 6d65 7100  ng;L..lastNameq.
0000060: 7e00 0178 7000 0000 0974 0005 5069 7070  ~..xp....t..Pipp
0000070: 6974 000b 4c61 6e67 7374 7275 6d70 6673  it..Langstrumpfs
0000080: 7100 7e00 0000 0000 0e74 0008 4b61 726c  q.~......t..Karl
0000090: 7373 6f6e 7400 0876 6f6d 2044 6163 68    ssont..vom Dach
```


DESY

Java – Die import Anweisung

- Mit der import Anweisung wird der Code der importierten Klassen bzw. statischen Inhalte dem Interpreter als Namensraum verfügbar gemacht. Es wird also möglich nach einem Import von javax.swing.JButton das Element JButton im Code mit seinem einfachen Namen, statt seinem voll qualifizierenden Namen anzusprechen. Im kompilierten Programmcode wird weiterhin der vollständige Name javax.swing.JButton benutzt. Das ganze dient also der Vereinfachung beim Programmieren.
- Seit Java 1.5 existiert zusätzlich noch die static import Anweisung mit deren Hilfe häufig gebrauchte statische Variablen und Methoden im Code über ihren einfachen Namen angesprochen werden können.
- In Java sind sogenannte Wildcard Importe (import java.swing.*;) möglich, sollten aber generell vermieden werden und zwar aus Gründen der Lesbarkeit und um im Konfliktfall (eine Klasse gleichen Namens in zwei mittels Wildcard importierten Paketen) einen Hinweis zu haben.

Java – Die assert-Anweisung

- Die assert Anweisung dient der Überprüfung notwendiger Nebenbedingungen (sog. Invarianten) von ausgeführtem Programmcode. Sie wurde mit Java 1.4 eingeführt.
- Laufen sie fehl, erzeugen Sie einen AssertionError
- Sie sollten nicht im Produktionscode ausgeführt werden, sind also ein Sprachmittel für die Testphase und müssen beim Aufruf des Interpreters über den Schalter –ea explizit aktiviert werden
- Es gibt zwei Varianten:
  - **assert BooleanExpression**;
  - **assert BooleanExpression ValueEvaluatedExpresssion**;
    In der zweiten Form wird der Wert, meistens ein String dem AssertionError zur Ausgabe und Erklärung mitgegeben

DESY

Java – java.lang.Object

- Ist die Wurzel des Java Klassenbaums
- Alle Objekte leiten (automatisch) von Object ab ...
- ... haben also Zugriff auf seine Methoden
- Da Object nichts über die Klassen "weiß", die von ihm ableiten, sind die Methoden in Object sehr allgemeine Implementierungen, die in den Unterklassen so wenig oder keinen Sinn machen und deshalb überschrieben werden müssen.
- Das Nutzen und Überschreiben ist für viele Mechanismen in Java notwendig, wie das Klonen (clone() und Cloneable), Vergleiche von Objekten (equals()), Nutzen von Objekten als Schlüssel oder Werte in Collections (hashCode() und equals()), Zwecks Ausgabe Objekte in eine String-Repräsentation überführen (toString()) und für Bereinigungsoperation durch die Garbage Collection (finalize())

Java – Objektvergleiche Comparable und equals()

- In Java existieren zwei Mechanismen für den Objektvergleich, das Nutzen bzw. Überschreiben der equals()-Methode aus Object und das Implementieren der Methode compareTo() über das Einbinden des Interface java.lang.Comparable<T>. Hierbei dient equals() als Test auf inhaltliche Gleichheit im Gegensatz zum Test auf Identität durch == (die default-Implementierung in Object testet auch nur auf Identität!) und compareTo() dient dem Vergleich von Objekten um eine Ordnung/ Reihenfolge auf diesen zu definieren.
- Diese beiden Verfahren sollten um Inkonsistenzen zu vermeiden, insbesondere aber, wenn Objekte in Collections und hier in Trees genutzt werden sollen, immer widerspruchsfrei zusammen implementiert werden – ist aber kein Muß!
- Wenn equals überschrieben wird, sollte auch hashCode() überschrieben werden, dabei muß darauf geachtet werden, daß bei der Erzeugung des

Java – Das Interface java.lang.Comparable

- Implementieren des Interface erlaubt:
  - Nutzung der sort() und binarySort() Methoden von Collections und Arrays
  - Benutzung als Key in a TreeMap
  - Benutzung als Elemente in TreeSet
- int compareTo() muß folgende Bedingungen erfüllen:
  - Der Rückgabewert von x.compareTo(y) hat das umgekehrte Vorzeichen von y.compareTo(x) wenn er nicht 0 ist
  - x.compareTo(y) wirft die gleichen Exceptions wie y.compareTo(x)
  - wenn x.compareTo(y) > 0 und y.compareTo(z) > 0, dann x.compareTo(z) > 0 und wenn x.compareTo(y) < 0 und y.compareTo(z) < 0, dann x.compareTo(z) < 0
  - wenn x.compareTo(y) == 0, dann hat x.compareTo(z) das gleiche Vorzeichen, wie y.compareTo(z)
- Zudem sollte x.compareTo(y) nur dann eine 0 zurückgeben, wenn x.equals(y) == true ist

Java – Implementieren von compareTo()

- Grundsätzlich muß man sich überlegen welche Attribute eines Objekts für die Sortierung relevant sind (Achtung! Konsistenz zu equals())
- Die einzelnen Attribute sollten wie folgt verglichen werden:
  - Primitive Datentypen können für den Vergleich die statische int compare(x,z) der zugehörigen Wrapperklassen nutzen, oder bei numerischen Typen mit x < y und x > y, sowie für booleans mit !x && y und x && !y arbeiten.
  - Objekte und Enumerations sollten die compareTo der zugehörigen Klasse/ Enumeration aufrufen
  - Collections und Arrays sind unter Umständen schwierig abzuhandeln, zuerst sollte auf gleiche Länge bzw. gleiche Anzahl von Elementen geprüft werden. Während es dann im Falle gleichlanger Arrays dann möglich ist Feld für Feld zu überprüfen, geht dies für einige Collections, mit undefinierter Ordnung während der Iteration nicht, hier benötigt man andere Strategien (Alle Elemente der einen Collection darauf überprüfen, ob sie in der anderen enthalten sind, mit contains())

Java – Überschreiben von equals()

- Das Überschreiben von equals() ist ebenfalls für die Nutzung von Collection notwendige Voraussetzung, da hierüber beispielsweise geprüft wird, ob ein Objekt schon in der Datenstruktur vorhanden ist
- Wenn equals() überschrieben wird muß auch hashCode() überschrieben werden, da sonst Inkonsistenzen entstehen können
- Bei der Implementierung sollte mit den Attributen der Klasse wie folgt vorgegangen werden:
  - Objekte, Enumerations und Collections nutzen das zugehörige equals()
  - Arrays nutzen java.util.Arrays.equals()
  - Alle primitiven Datentypen, außer float und double nutzen ==
  - float nutzt erst die Konversion zu int mit Float.floatToIntBits() und dann ==
  - double nutzt Double.doubleToLongBits() und dann ==
  - Alternative: Nutzung der Wrapper-Klassen für die primitiven Datentypen (gilt auch für compareTo())

Java – Überschreiben von hashCode()

- Hashfunktionen sind eine Klasse nicht umkehrbarer Funktionen und bilden meistens einen großen Datenbereich auf einen kleineren ab.
- Während sichere Hashfunktionen besonderen Wert auf Kollisionsfreiheit legen müssen ist dies für die Implementierung von hashCode in Java nicht primär wichtig, da schon der Rückgabewert die Ausgabemenge auf 2^{32} Werte beschränkt.
- Trotzdem sollte versucht werden eine gute Verteilung der Eingabemenge auf die Ausgabemenge zu erzielen
- Wichtiger sind folgende Eigenschaften:
  - Für zwei Objekte für die equals() true bzw. compareTo() eine 0 zurückgibt muß hashCode() den gleichen Wert liefern, d.h. aber nicht, daß ungleiche Objekte verschiedene hashCode-Werte haben müssen, oder daß Objekte mit dem gleichen hashCode gleich sein müssen!
  - hashCode() muß zur Berechnung die selben Attribute heranziehen, die von equals() genutzt werden um inhaltliche Gleichheit festzustellen!

Java – Überschreiben von hashCode()

- Die einzelnen Attribute sollten mittels XOR verknüpft werden (gute Verteilung! – siehe Wahrheitstabelle im Vergleich zu OR und AND)
- Ausgehend von einem Seed, am besten eine Primzahl werden die einzelnen Attribute dann wie folgt zur Ausgabe hinzugefügt:
  - Objekte über deren hashCode()
  - byte, char, short und int als int (upcast)
  - long über eine XOR Verknüpfung der oberen 32 Bit mit den unteren und dann einem downcast
  - float mit Float.floatBitsToInt()
  - double mit Double.doubleBitsToLong() und dann wie mit long
  - boolean über den ternären Operator: boolean ? 0 : 1
  - Arrays handeln jedes Feld separat und nach Typ ab.

Java – equals(), hashCode(), compareTo(), Beispiel

- Im folgenden Beispiel wurden für eine Klasse Person mit 4 Attributen die Methoden equals(), hashCode() und compareTo() überschrieben
- Die Felder der Beispielklasse sind alle als final deklariert, um keine Probleme (notwendiges rehashen bei Vergrößerung der Collection) bei der Nutzung in Collections zu bekommen, bei Immutable Klassen ist das kein Problem.
- Das Beispiel handelt eine robuste, mögliche Realisierung der 3 Methoden ab.
- Die compareTo()-Methode nutzt eine assert-Anweisung, um die Konsistenz zur equals()-Methode sicherzustellen

Java – equals(), Beispiel

```java
/* Person.java */
package examples.compare;

public final class Person implements Comparable<Person>{

    private final String firstName;
    private final String lastName;
    private final int socialId;
    private final boolean married;

    public Person(String firstName, String lastName,
                  int socialId, boolean married){
        this.firstName = firstName;
        this.lastName = lastName;
        this.socialId = socialId;
        this.married = married;
    }

    public String getFirstName(){return this.firstName;}
    public String getLastName(){return this.lastName;}
    public int getSocialId(){return this.socialId;}
    public boolean isMarried(){return this.married;}

    @Override
    public boolean equals(Object anObject){
        if(anObject == null){return false;}
        if(this.getClass() != anObject.getClass()){return false;}
        if(this == anObject){return true;}
        Person aPerson = (Person)anObject;
        return this.firstName.equals(aPerson.getFirstName())
            && this.lastName.equals(aPerson.getLastName())
            && (this.socialId == aPerson.getSocialId())
            && (this.married == aPerson.isMarried());
    }
```


DESY

Java – compareTo() und hashCode(), Beispiel

```
    @Override
    public int compareTo(Person aPerson){
        final int LESS = -1;
        final int EQUAL = 0;
        final int GREATER = 1;
        if(this == aPerson) return EQUAL;
        if(!this.married && aPerson.isMarried()) return LESS;
        if(this.married && !aPerson.isMarried()) return GREATER;
        if(this.socialId < aPerson.getSocialId()) return LESS;
        if(this.socialId > aPerson.getSocialId()) return GREATER;
        int comparison = this.lastName.compareTo(aPerson.getLastName());
        if(comparison != EQUAL) return comparison;
        comparison = this.firstName.compareTo(aPerson.getFirstName());
        if(comparison != EQUAL) return comparison;
        // as all comparisons have yielded equality
        // we finally verify consistency with equals()
        assert this.equals(aPerson);
        return EQUAL;
    }

    @Override
    public int hashCode(){
        int result = 31;
        result ^= this.firstName.hashCode();
        result ^= this.lastName.hashCode();
        result ^= this.socialId;
        result ^= this.married ? 0 : 1;
        // for f = byte, char, short do: result ^= (int)f;
        // for f = long do: result ^= (int)(f ^ (f >>> 32));
        // for f = float do: result ^= Float.floatToIntBits(f);
        // for f = double do: result ^= Double.doubleToLongBits(f);
        // and then as you do with long values
        // if f is an array then you have to handle each value seperately
        return result;
    }
}
```


DESY

Java – equals() und hashCode() ab Java 7

- Mit Java 7 wurde die Klasse java.util.Objects verfügbar
- Die Klasse hält die statische Methoden equals(), deepEquals(), hashCode() und hash() mit deren Hilfe sich die Realisierung der 2 Methoden vereinfachen bzw. vereinheitlichen läßt.

```
@Override
public boolean equals(Object anObject){
    if(anObject == null){return false;}
    if(this.getClass() != anObject.getClass()){return false;}
    if(this == anObject){return true;}
    Person7 aPerson = (Person7)anObject;
    return Objects.equals(this.firstName, aPerson.getFirstName())
        && Objects.equals(this.lastName, aPerson.getLastName())
        && Objects.equals(this.socialId, aPerson.getSocialId())
        && Objects.equals(this.married, aPerson.isMarried());
}


@Override
public int hashCode(){
    return Objects.hash(this.firstName, this.lastName,
                        this.socialId, this.married);
}
```


DESY

Java – Das Interface java.util.Comparator<T>

- Die Methoden zur Sortierung in den Collections Klassen erwarten zur Festlegung der Reihenfolge die Übergabe eines Comparator-Objekts. Das ist ein Objekt einer Klasse die das Comparator-Interface und damit die beiden Methoden compare(<T> object1, <T> object2) und equals() implementiert. compare funktioniert wie ein ausgelagertes compareTo und equals überschreibt die equals-Methode aus Object.
- Dies ist als Alternative zu den vorher genannten Methoden zu sehen und insbesondere dann von Interesse, wenn Datenstrukturen mit verschiedenen Sortierungen (aufsteigend, absteigend, ...) bedacht werden sollen, das es für Klassen durchaus mehrere Comparator Objekte geben kann.

Java – Immutable Objects

- Immutable nennt man Objekte von Klassen deren Attribute nach der Erzeugung nicht mehr verändert und von denen auch keine weniger restriktiven Klassen abgeleitet werden können.
- Joshua Bloch, Vater des Collection Frameworks in Java, meint dazu:

"Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible."

Java - Warum Immutable Objects?

- Volle Kontrolle über das Objekt, hat die Instanzierung geklappt, kann das Objekt nicht mehr in einen inkonsistenten Zustand geraten.
- Man braucht keine Implementierung von clone() oder einen Copy-Konstruktor und kann flache Kopien benutzen
- Prima geeignet als Schlüssel in Maps und Elemente in Sets
- Sind automatisch Thread-safe, keine Synchronisierungsprobleme

Java – Implementierung von Immutable Objects

- Es gibt zwei Strategien zur Implementierung von Immutables
- 1) "Echte" Immutables:
  - Die Klasse wird als final deklariert, keine Unterklassen möglich
  - Alle Felder der Klasse werden als private final deklariert
  - Setter-Methoden werden nicht implementiert und auch keine anderen Methoden, die den Zustand des Objekt ändern können
  - Im Konstruktor existieren keine Methodenaufrufe, das Objekt wird in einem Schritt erzeugt
- 2) Immutable mit "Hintertür"
  - Zur veränderbaren Klasse existiert ein Interface ohne Setter-Methoden
  - Nach Außen wird nur das Interface weitergereicht
  - Die veränderbare Klasse wird versteckt und Objekte werden via getInstance() Methode nach außen gereicht
  - Intern bleiben Objekte der Klasse veränderbar.

DESY

Java – Immutable mit Backdoor

```java
/* Immutable with backdoor */

public interface Point {

    public int getX();
    public int getY();
}

public class MutablePoint implements Point {

    private int x;
    private int y;

    public MutablePoint(int x, int y) { this.x = x; this.y = y; }

    public int getX() { return x; }
    public int getY() { return y; }

    public void setX(final int x) { this.x = x; }
    public void setY(final int y) { this.y = y; }
}

// Nutzung:

  Point p = widget.getPoint(20,50);  // Immutable
  ((MutablePoint)p).setX(100);       // Mutable
```


DESY

Java – Der varargs-Operator

- Seit Java 1.5 existiert die Möglichkeit in Parameterlisten von Methoden die Übergabe einer variablen Anzahl von Argumenten eines Typs mit Hilfe des varargs-Operator **TYP... bezeichner** zu definieren.
- Unter dem Namen des Bezeichners wird ein Array des definierten Typs übergeben
- Einer so definierten Parameterliste können sowohl eine beliebige Anzahl von Argumenten des genannten Typs, als auch ein Array des genannten Typs übergeben werden
- Pro Parameterliste kann der varargs-Operator nur einmal genutzt werden und muß am Ende der Parameterliste stehen.
- Vor dem varargs-Operator können beliebige andere Argumente stehen
- Die main-Methode kann somit auch wie folgt definiert werden:
  - public static void main(String... varargs)

DESY

Java – Generics, Templates für Datenstrukturen

- Generics wurden mit Java 1.5 eingeführt
- Im Prinzip handelt es sich um die Möglichkeit Variablenbezeichner für Datentypen zu benutzen
- Dies macht folgende Dinge unnötig:
  - Mehrfachimplementierungen für verschiedene Datentypen
  - Downcasts, weil Object als allgemeiner Datentyp übergeben wurde
- Die Typbezeichner werden bei Definition von Klassen, Interfaces, Methoden etc. benutzt und dann bei der Deklaration und Initialisierung durch einen konkreten Typ ersetzt:
  - class ArrayList<E>{...}
  - ArrayList<String> aList = new ArrayList<String>(); // vor 1.7
  - ArrayList<String> aList = new ArrayList<>(); // ab 1.7, Achtung bei getrennter Deklaration und Initialisierung – Lesbarkeit!

DESY

Java – Vor Einführung von Generics

```java
/*
 * BeforeGenerics.java
 * Deprecated now, must be run with the following flags:
 * javac -Xlint:deprecation BeforeGenerics.java
 * or : javac -Xlint:deprecation -Xlint:unchecked BeforeGenerics.java
 */
package examples.generics;

import java.util.ArrayList;
import java.util.Iterator;

public class BeforeGenerics{

    public static void main(String... varargs){
        // before SUN introduced generics
        // all Collection classes were collections
        // of references of type object, this were source
        // for some serious problems:
        ArrayList aList = new ArrayList();
        aList.add(new String("Hello"));
        aList.add(new Integer(5));
        for(Iterator it = aList.iterator(); it.hasNext();){
            String s = (String)it.next(); // ClassCastException
            System.out.println(s);
        }
    }
}
```

Ausgabe:

```
[:-)] [16:08:38]: /Users/dramburg/Code/Java/examples/generics
--> $ java examples.generics.BeforeGenerics
Hello
Exception in thread "main" java.lang.ClassCastException:
     java.lang.Integer cannot be cast to java.lang.String
      at examples.generics.BeforeGenerics.main(BeforeGenerics.java:20)
```


DESY

Java – Mit Generics

```java
/*
 * WithGenerics.java
 */
package examples.generics;

import java.util.ArrayList;
import java.util.Iterator;

public class WithGenerics{

    public static void main(String... varargs){

        ArrayList<String> aList = new ArrayList<String>(); // type safe
        ArrayList<Integer> intList = new ArrayList<>(); // possible since 1.7
        aList.add(new String("Hello"));
        //aList.add(new Integer(5)); // will not compile!
        aList.add(new String("Leute"));
        intList.add(new Integer(5));
        intList.add(new Integer(13));
        for(String s: aList){
            System.out.println(s);
        }
        for(Iterator it = intList.iterator(); it.hasNext();){
            Integer anInteger = (Integer)it.next(); // never throws an Exception
            System.out.println(anInteger);
        }
        //But problems are already possible:
        ArrayList<Number> numberList = new ArrayList<>();
        numberList.add(new Integer(13));
        numberList.add(new Double(7.9)); // compiles!
        for(Iterator it = numberList.iterator(); it.hasNext();){
            Integer anInteger = (Integer)it.next(); // ClassCastException
            System.out.println(anInteger);
        }
        for(Integer i: numberList){ // Compile time error! error: incompatible types:
            System.out.println(i); //  Number cannot be converted to Integer
        }
        for(Number i: numberList){
            System.out.println((Integer)i); // ClassCastException
        }
    }
}
```


DESY

Java – Wildcard Generics

- Der Wildcard Operator für generische Typen ist das ?
- Der Operator kommt zum Einsatz, wenn von einem Datentyp lediglich eine gemeinsame Elternklasse gefordert wird aber keine weitere Spezialisierung.
- ACHTUNG! Bei dieser Verwendung darf dann auch lediglich die Schnittstelle der gemeinsamen Oberklasse genutzt werden:
  - class A{methodA()}
class B extends A{methodB()}
class C extends A{methodC()}
  - aMethod(<? extends A> anAObject){B anBObject = (B)anAObject;} // GEHT NICHT!

Java – Generics, Konventionen für Typbezeichner

- E – Element (in Collections)
- K,V – Key, Value (beispielsweise in Maps)
- N – Number
- T – Type
- S,U,V etc. – 2., 3., 4. Type

Java – Generics und varargs

- Ergänzen sich prima im Zusammenspiel beim Überschreiben von Methoden, da wo es Sinn macht. Man kann an Stelle von:
  - public void aMethod(String[ ] arr){...}
    public void aMethod(Integer[ ] arr{...}
    public void aMethod(Byte[ ] arr){...}
- ... einfach auch das folgende machen:
  - public void aMethod(T... arr){doSomethingMindfulWith(T);}

Java – Klassen- und Objektbeziehungen

- Ist-ein(oder-mehrere)-Beziehung, Polymorphie
  - Wird erreicht über Vererbung oder Schnittstellenimplementierung
  - Vielgestaltigkeit setzt eigentlich den Mechanismus der späten Bindung voraus. Späte Bindung meint, daß der genaue Typ eines Objekts erst zur Laufzeit und nicht schon zum Zeitpunkt des Kompilierens bekannt sein muß.
- Hat-ein(-oder-mehrere)-Beziehung, Assoziation
  - Komposition – Die Teile existieren nach Löschung des Ganzen nicht weiter. Beispiel: Haus und Zimmer, Haus hat ein oder mehrere Zimmer, wird das Haus abgerissen hören auch die Zimmer auf zu existieren.
  - Aggregation – Die Teile existieren auch nach Löschung des Ganzen weiter. Beispiel Vertrag und Vertragspartner, Vertrag hat mehrere Vertragspartner, die auch nach Aufhebung des Vertrags weiter existieren.

Java – Einschub: Softwareentwicklung

- Beschreibt einen Prozess währenddessen ausgehend von einer Spezifikation (Anforderungen des Auftraggebers) über eine Modellierung eine Software (mit Dokumentation)
- Zur Durchführung des Prozesses existieren verschieden Verfahren:
  - Wasserfallmodell, Spiralmodell, V-Modell, Agile Methode
- Wie diese Verfahren durchgeführt werden bestimmt die zur Anwendung kommende Softwareentwicklungs-Philosophie
  - Extreme Programming, Rapid Prototyping, Rational Unified Process (UML), Scrum, Kanban
- Softwarelebenszyklus
  - Meint die gesamte Spanne der Existenz einer Software von der Aufnahme der Anforderungen (Lastenheft) über die Phasen Entwurf, Implementierung, Test, Auslieferung und Installation beim Kunden, sowie die Wartung während der Betriebszeit beim Kunden.

Java – Einschub: UML

- Unified Modelling Language
- Entwickelt von Grady Booch, Ivar Jacobson, James Rumbaugh während ihrer Arbeit am Rational Unified Process als Angestellte der Firma Rational Inc.
- Grafische Modellierungssprache zur Spezifikation und dem Entwurf von Software, sowie ihrer Dokumentation
- Ziel war eine Sprache zu entwickeln mit deren Hilfe sich die Struktur und das Verhalten einer Software vollständig beschreiben und dokumentieren lässt.
- Regelt begrifflich hauptsächlich die Beziehungen zwischen den genutzten Entitäten
- Formuliert kein Vorgehensmodell zur Softwareentwicklung

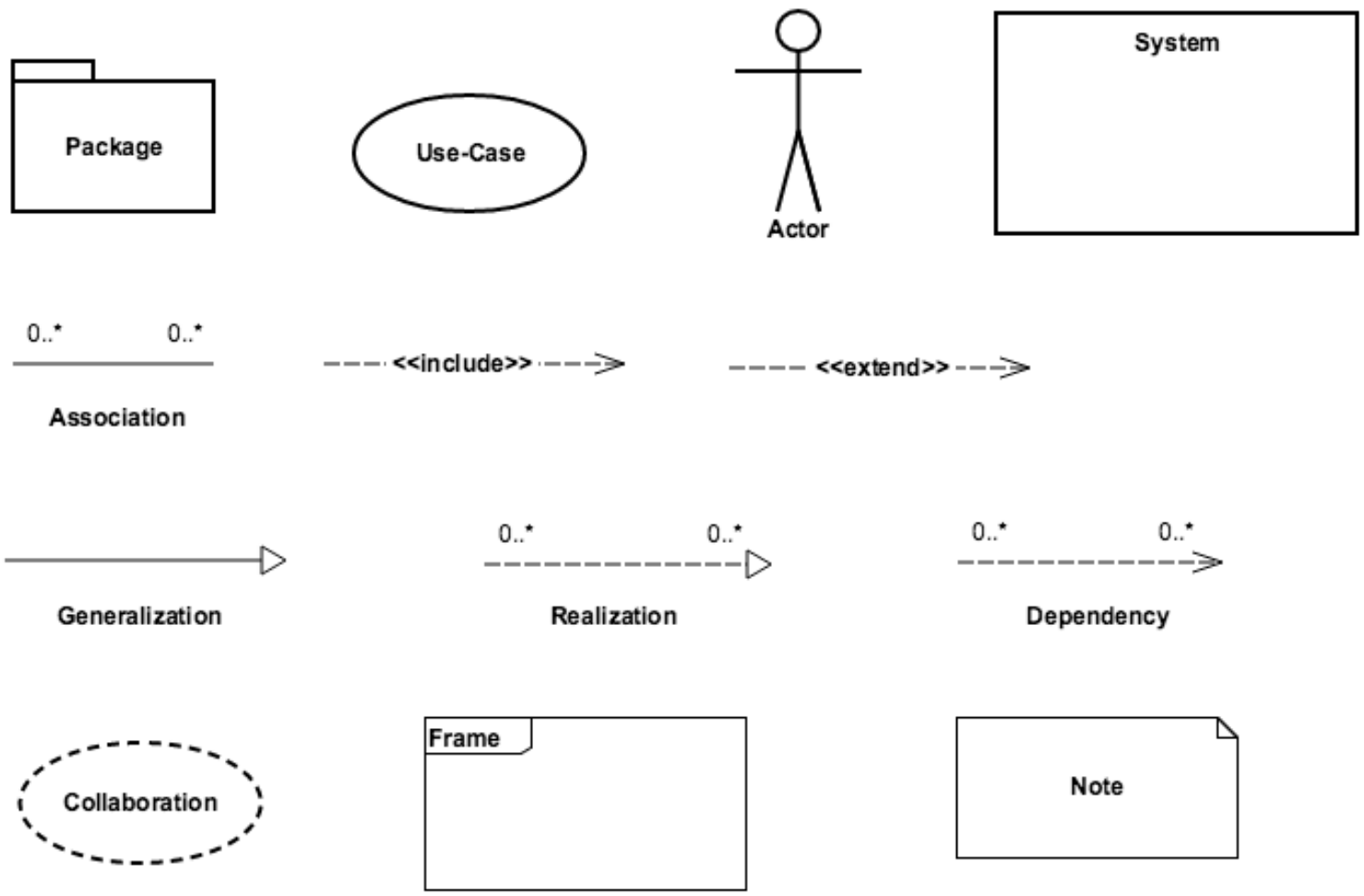
Java – Einschub: UML Sprachelemente

- UML definiert Sprachelement denen gewisse Begriffe und Diagrammtypen zugeordnet sind. Dabei können Begriffe und grafische Gestaltungselemente durchaus in verschiedenen Sprachelementen mit abweichender oder veränderter Bedeutung auftauchen.
  Sprachelemente sind:
  - Aktionen, Aktivitäten, Allgemeines Verhalten, Anwendungsfälle, Informationsflüsse, Interaktionen, Klassen, Komponenten, Kompositionsstrukturen, Profile, Verteilungen, Zustandsautomaten

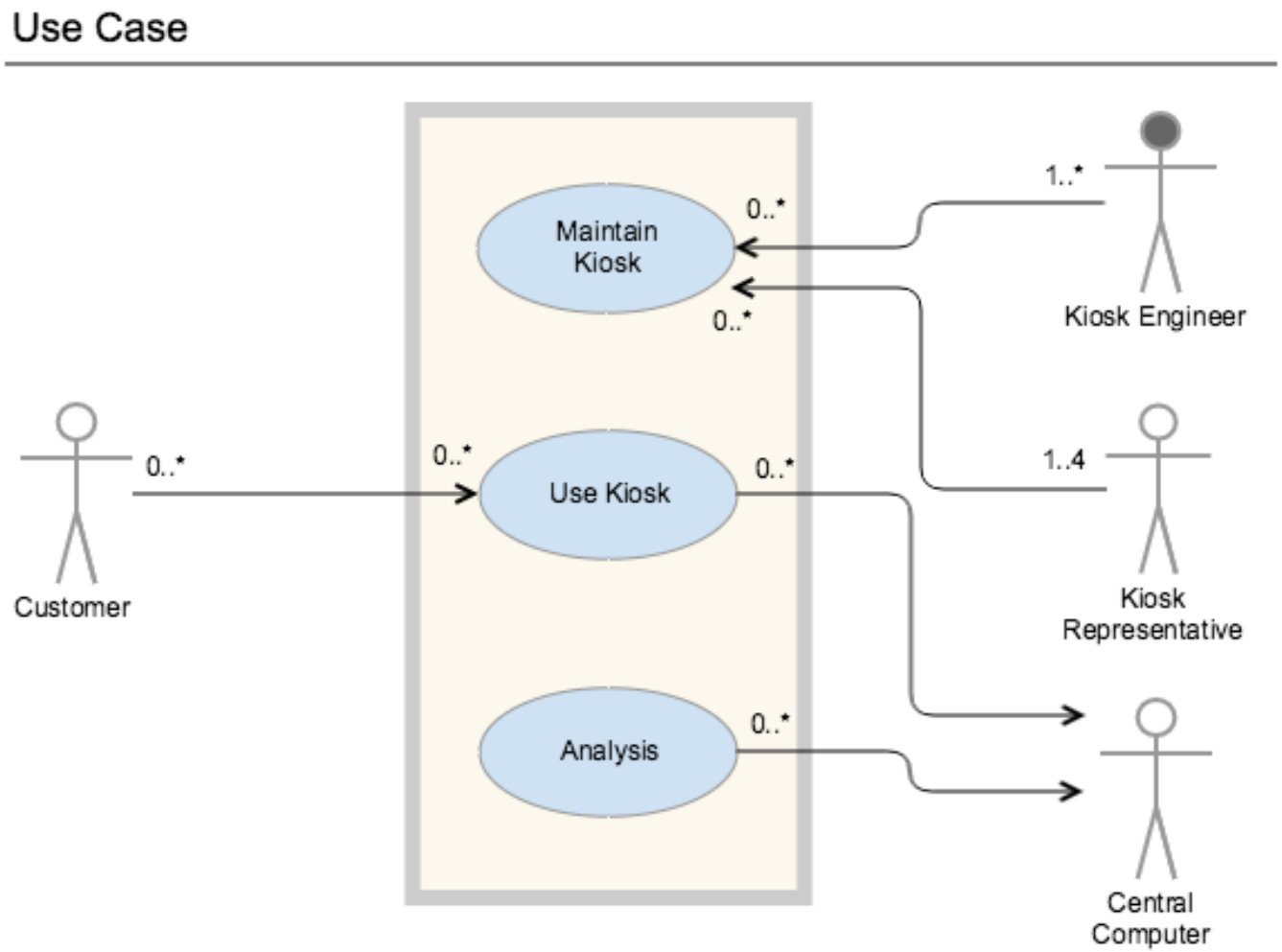
Java – Einschub: UML Diagrammtypen

- Strukturdiagramme
  - Klassendiagramm, Kompositionsstrukturdiagramm, Komponentendiagramm, Verteilungsdiagramm, Objektdiagramm, Paketdiagramm, Profildiagramm
- Verhaltensdiagramme
  - Aktivitätsdiagramm, Anwendungsfalldiagramm (Use-Case), Interaktionsübersichtsdiagramm, Kommunikationsdiagramm, Sequenzdiagramm, Zeitablaufsdiagramm, Zustandsdiagramm
- Über die Verwendung als Sprache zur Spezifikation und Dokumentation hinaus haben sich besonders Klassen- und Aktivitätsdiagramme, sowie Use-Case-, Sequenz- und Zustandsdiagramme als Erklärungshilfen in der Literatur rund um IT-Themen eingebürgert und sollten, wenn auch nicht im Entwurfsprozess eingesetzt, zumindest doch aus diesem Grund verstanden werden.

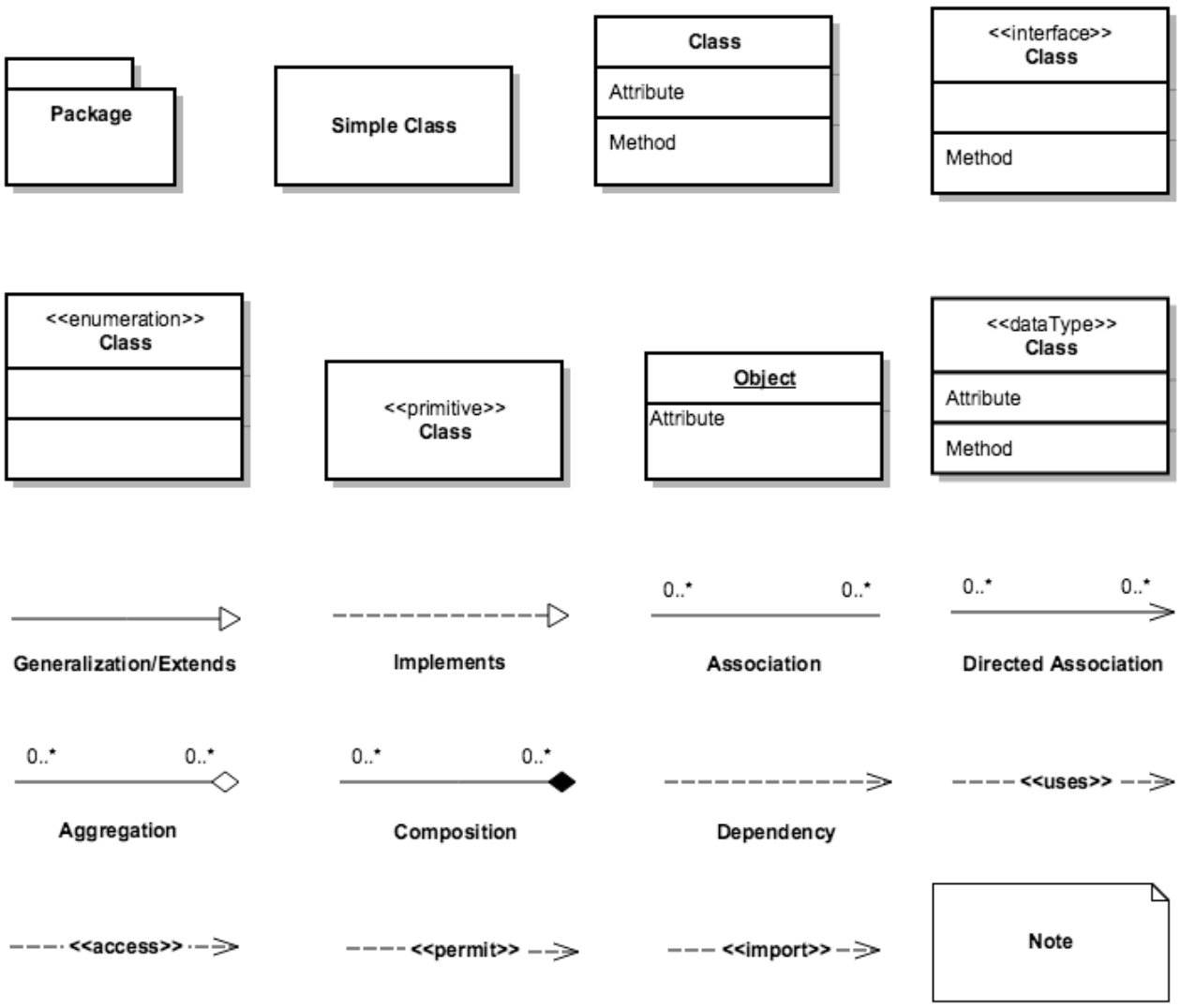
Java – Einschub: UML Use-Case-Diagramm, Legende



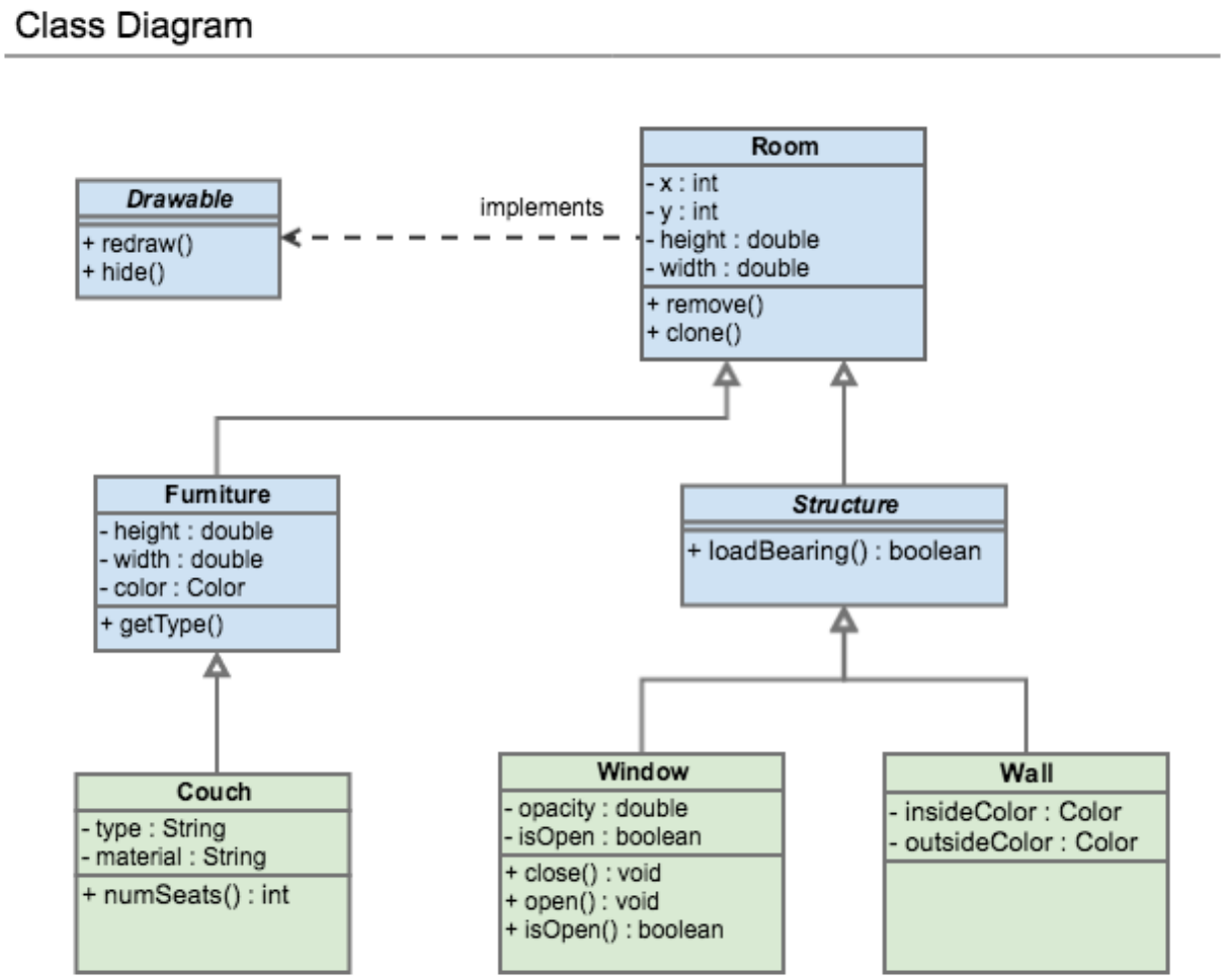
Java – Einschub: UML Use-Case-Diagramm



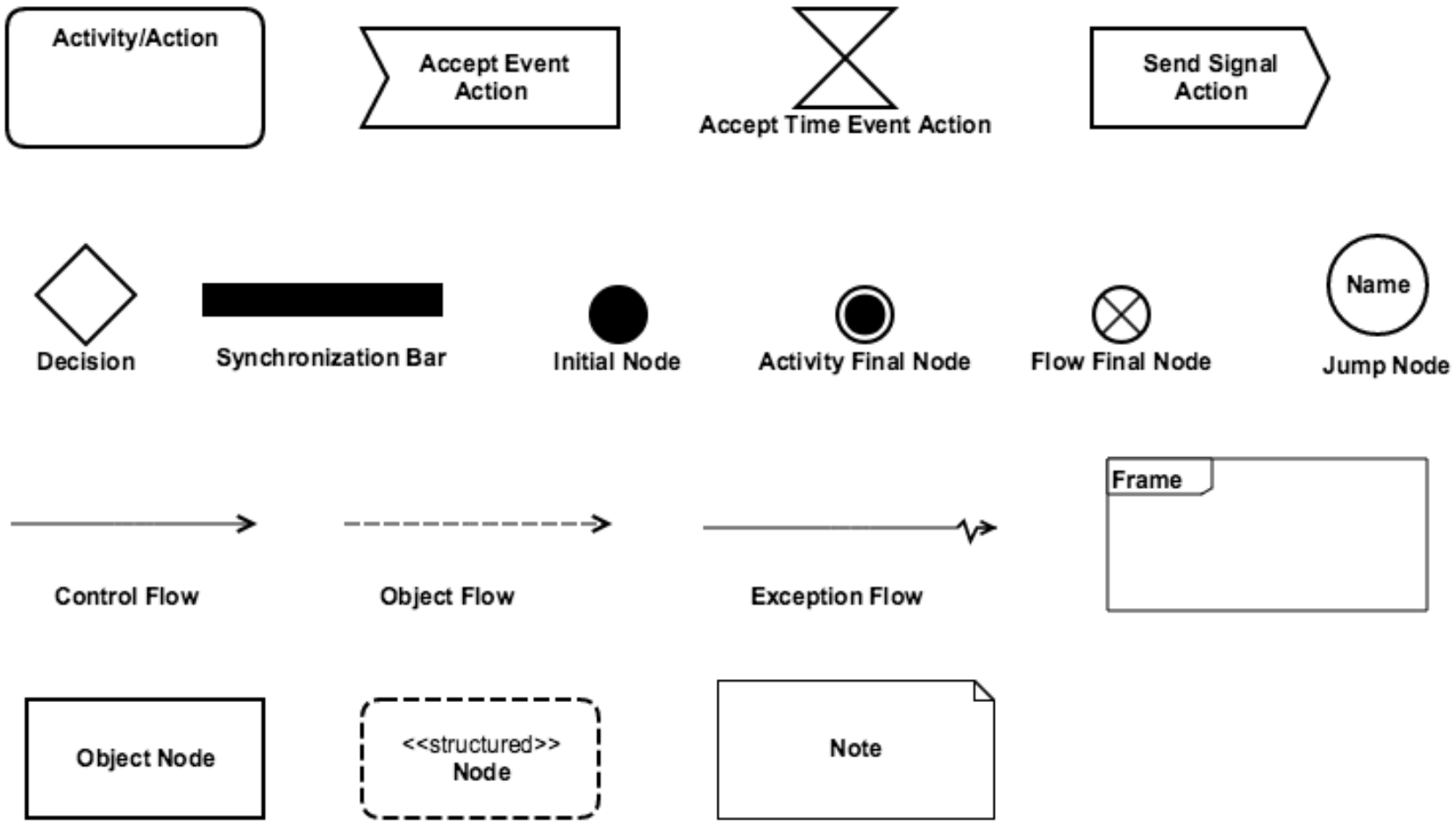
Java – Einschub: UML Klassendiagramm, Legende



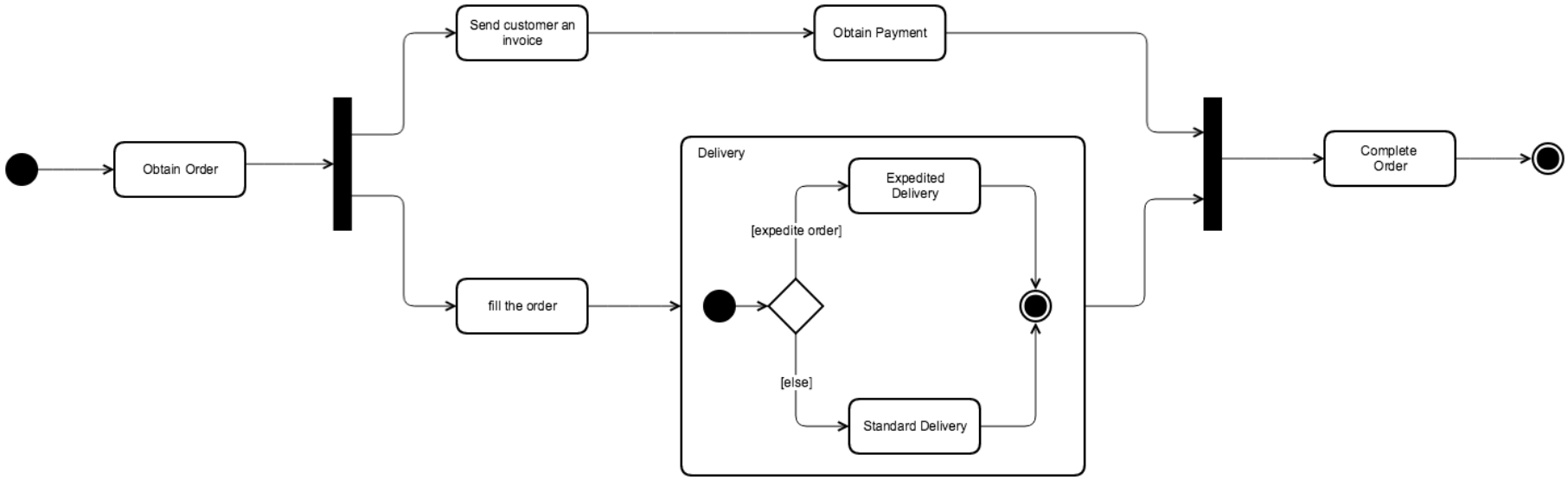
Java – Einschub: UML Klassendiagramm



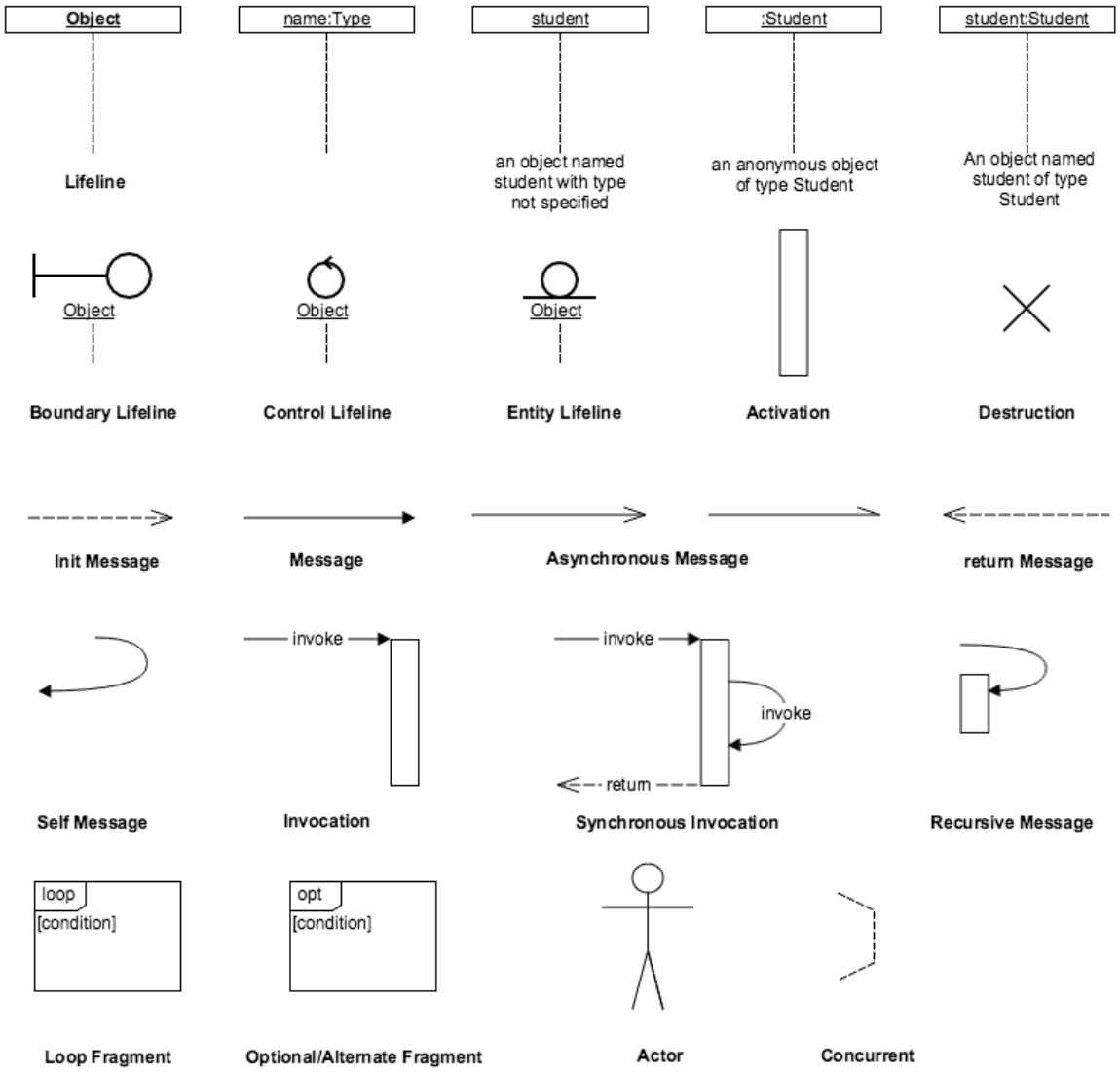
Java – Einschub: UML Activity, Legende



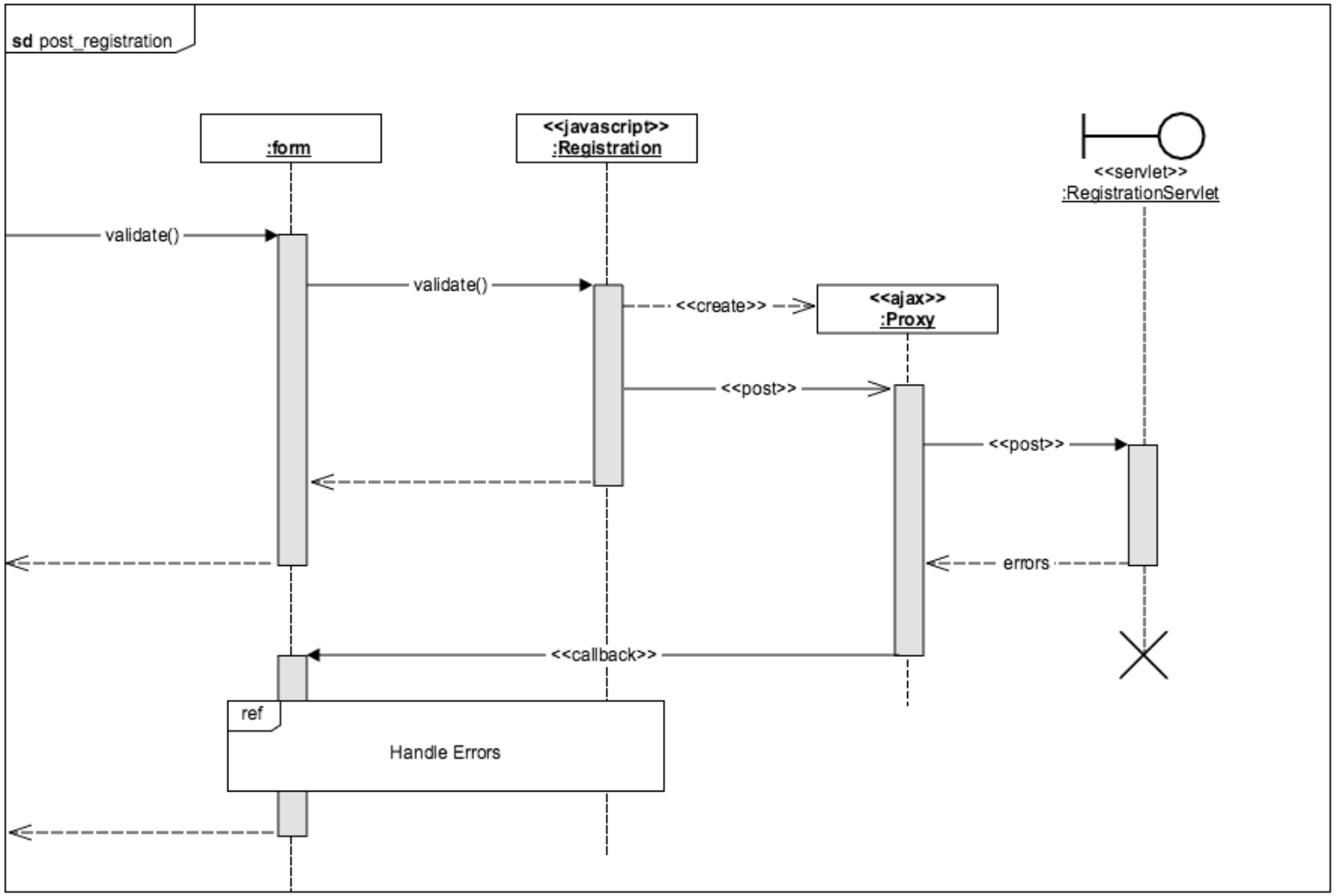
Java – Einschub: UML Activity



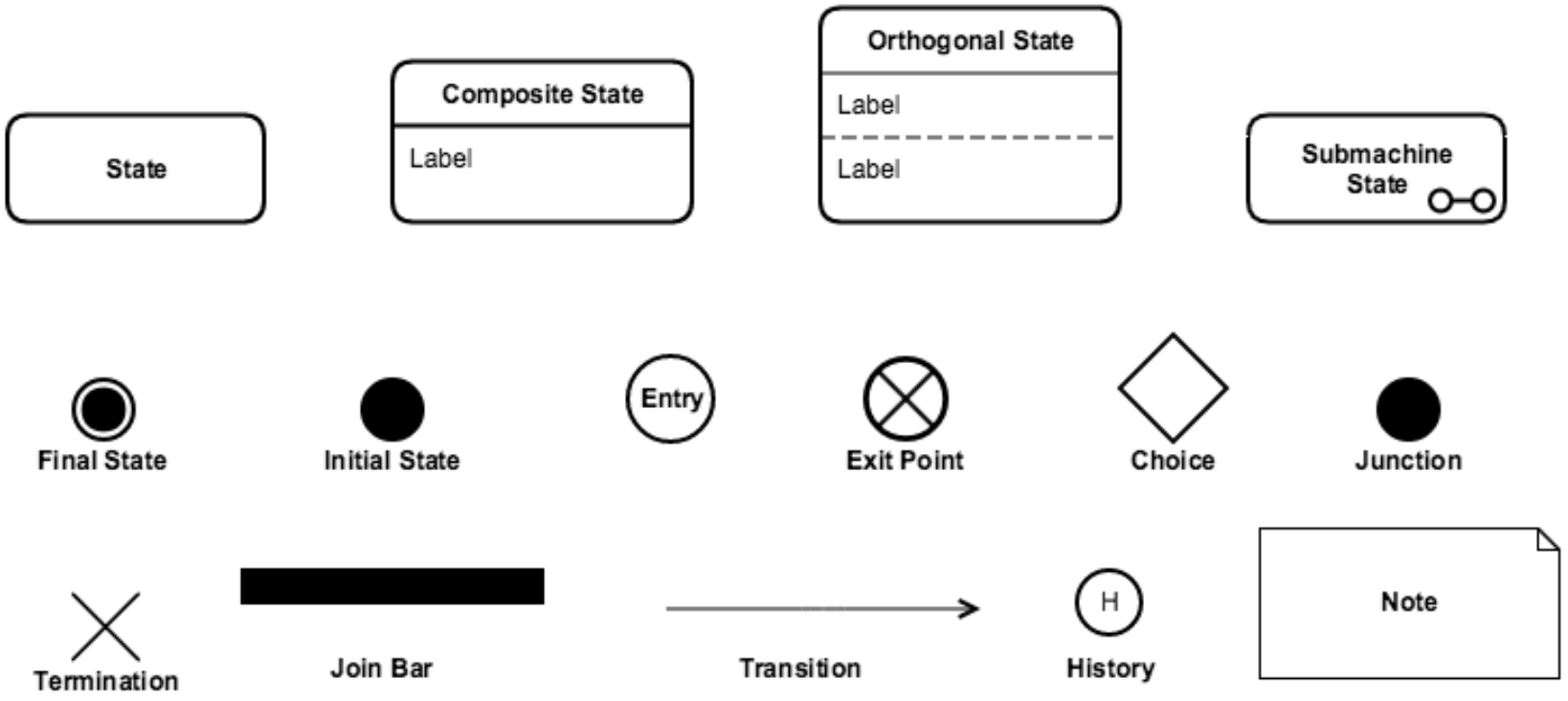
Java – Einschub: UML Sequenzdiagramm, Legende



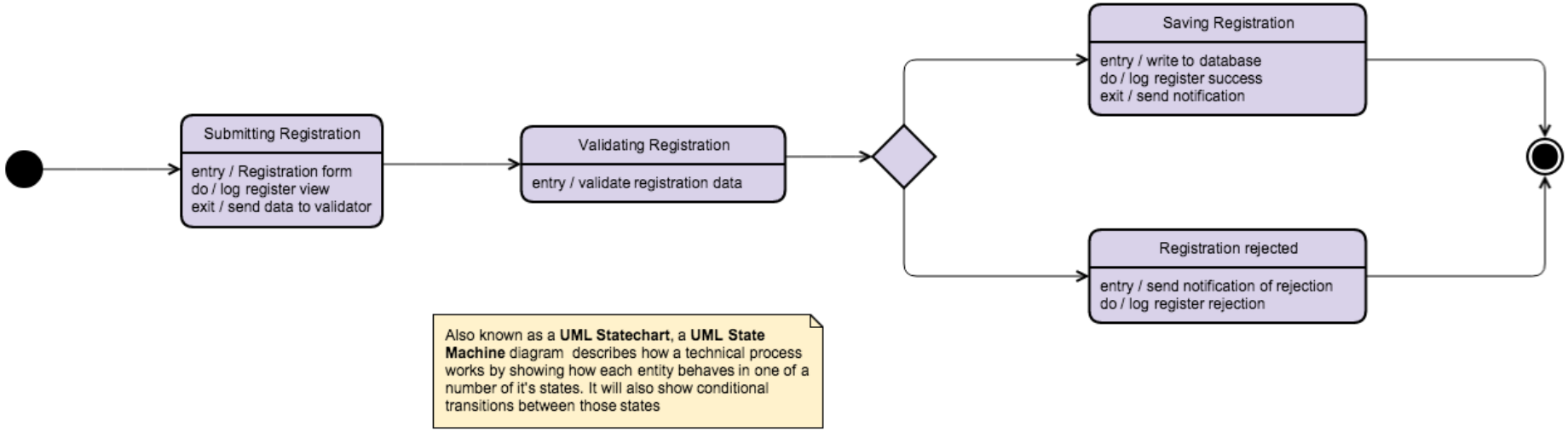
Java – Einschub: UML Sequenzdiagramm



Java – Einschub: UML Zustandsdiagramm, Legende



Java – Einschub: UML Zustandsdiagramm



Java – Array

- Statische, indizierte Datenhaltung in einer Sequenz von Feldern, wie der klassische Array in C
- Streng typisiert.
- Build-in der Sprache Java
- Einfache direkte Initialisierung nur in einer Zeile möglich:
  - ```
    int[ ] numbers = {1,2,3,4,5};
    String[ ] names;
    names = new String[ ]{"eins", "zwei", "drei", "vier", "fuenf"};
    ```
- Zugriff und Zuweisung über Index-Operator [ ]
- Abfrage der Länge des Array über das Attribut length (Achtung Feld keine Methode wie bei String!)
    ```

Java – java.util.Arrays

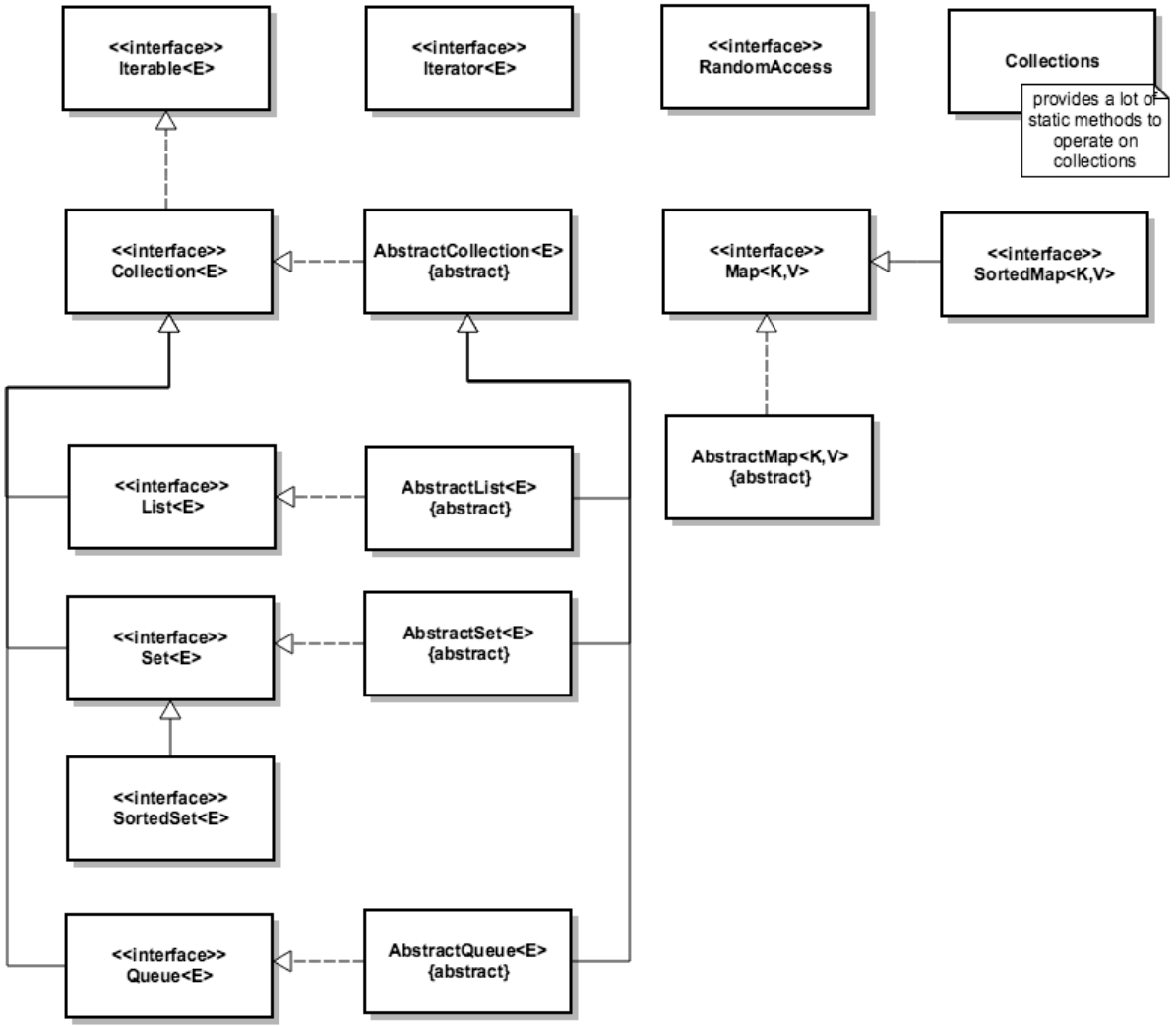
- Helfer-Klasse mit statischen Methoden zum Arbeiten mit und auf Arrays
  - Zum Suchen in Arrays, binarySearch()
  - Zum Kopieren von Arrays, copyOf(), oder von Teilbereichen copyOfRange
  - Zum Elementeweisen Vergleichen, equals()
  - Zur Berechnung des Hashwerts, hashCode()
  - Um den Array mit einem Element aufzufüllen, fill()
  - Zum Sortieren anhand eines Comparator-Objekts, oder anhand ihrer natürlichen Ordnung bei primitiven Datentypen, sort()
  - ... und mehr

Java – Das Collections Framework

- Besteht aus einer Sammlung von Interfaces und Klassen, die einen Zugriff auf dynamisch erweiterbare Datenhaltungsobjekte bietet. Dieses sind im wesentlichen Listen, Sets (Mengen) und Maps (Speicherung von Schlüssel-Wertpaaren). Die einzelnen Implementierungen unterscheiden sich sehr im Laufzeitverhalten, deshalb sollte eine Collection immer passen zum Anwendungsfall gewählt werden.
- In Collections können keine primitiven Datentypen gespeichert werden, es sind nur Objekte erlaubt, sowohl für Elemente als auch Schlüssel. Für primitive Datentypen müssen die Wrapperklassen (auch in Kombination mit Autoboxing) benutzt werden.
- Die Collections finden sich im Paket java.util (nicht synchronized, bis auf Vector) und im Paket java.util.concurrent (synchronized, also Thread-safe)
- Die Klasse java.util.Collections bietet statische (Hilfs-)Methoden für den Zugriff auf und das Arbeiten mit den Collections

Java – Collections Framework, Überblick

Java Collections Framework, Überblick



Java – Collections Framework, java.util.Collections

- Utilit-Klasse für alle Collections
- Hat nur statische Methoden:
  - Zum Elemente vertauschen, swap()
  - Zum Sortieren, sort()
  - Zum Suchen von kleinstem bzw. größtem Element einer Collection anhand eines übergebenen Comparator, min() bzw. max()
  - Zum Umwandeln einer Collection in eine synchronisierte Version, synchronizedXYZ() mit XYZ = Collection, List, Set, Map, SortedSet und SortedMap
  - Zum Herstellen einer immutable View einer Collection, unmodifiableXYZ mit XYZ wie oben
  - Zum Überprüfen, ob zwei Collections keine gemeinsamen Elemente haben, disjoint()
  - und noch einiges mehr...

Java – Collections Framework, Iterator

- Ein Iterator ist ein handle auf einen Mechanismus, der mir jedes Element einer Collection genau einmal liefert.
- Dieser Mechanismus wird über zwei Interfaces und der Implementierung ihrer Methoden erreicht
- java.lang.Iterable<T>
  - Implementierung macht Objekte mit der foreach-Schleife nutzbar (Ausnahme [ ])
  - Definiert die Methode iterator(), deren Aufruf einen Iterator<T> zurückliefert
- java.util.Iterator<E>
  - Klasse die das Interface implementiert wird zum Iterator
  - boolean hasNext() – liefert true solange wir nocht nicht am Ende der Collection sind
  - E next() – liefert ein Element des Iterators, darf nur aufgerufen werden wenn vorheriger Aufruf von hasNext() true zurückliefert.
  - remove() – löscht das letzte von next() zurückgegebene Element, Implementierung optional, d.h. wenn nicht gewünscht wirft remove eine UnsupportedOperationException

Java – Collections Framework, Iterator Beispiel

```java
/*
 * Cinema.java
 */
package examples.collections;

import java.util.Iterator;
import java.util.Arrays;

class ArrayIterator<E> implements Iterator<E>{
    private E[] anArray;
    private int index;

    public ArrayIterator(E[] anArray){
        this.anArray = anArray;
        this.index = 0;
    }
    @Override
    public boolean hasNext(){return index < anArray.length;}
    @Override
    public E next(){return anArray[index++];}
    @Override
    public void remove(){
        throw new UnsupportedOperationException();
    }
}

class Movie{
    private String title;

    public Movie(String title){
        this.title = title;
    }
    public String getTitle(){return title;}
}
```

Java – Collections Framework, Iterator Beispiel

```
public class Cinema implements Iterable<Movie>{
    private String name;
    private Movie[] screenedMovies;

    public Cinema(String name){
        this.name = name;
        this.screenedMovies = new Movie[0];
    }
    public void addMovie(Movie movie){
        int oldLength = screenedMovies.length;
        screenedMovies = Arrays.copyOf(screenedMovies, oldLength + 1);
        screenedMovies[oldLength] = movie;
    }
    @Override
    public Iterator<Movie> iterator(){
        return new ArrayIterator<Movie>(screenedMovies);
    }

    public static void main(String... args){
        Cinema cinema = new Cinema("Master Blockbuster");
        Movie actionMovie = new Movie("Arnie says: Ooorschgsicht!");
        Movie animeMovie = new Movie("Eisss √Ñtsch 200");
        Movie scifiMovie =
            new Movie("Schtaa Worsch 73 - The return of anyone");
        cinema.addMovie(actionMovie);
        cinema.addMovie(animeMovie);
        cinema.addMovie(scifiMovie);
        for(Movie m: cinema){
            System.out.println(m.getTitle());
        }
    }
}
Ausgabe:
--> $ javac Cinema.java
[:-)] [17:09:35]: /Users/dramburg/Code/Java/examples/collections
--> $ java examples.collections.Cinema
Arnie says: Ooorschgsicht!
Eisss Ätsch 200
Schtaa Worsch 73 – The return of anyone
```


DESY

Java – Collections Framework, java.util.Collection<E>

- Basis Interface des Collection Framework für Listen, Sets und Queues
- Definiert die Methoden
  - boolean add(E element)
  - boolean addAll(Collection<? extends E> collection)
  - void clear()
  - boolean contains(Object object)
  - boolean containsAll(Collection<?> collection)
  - boolean isEmpty()
  - boolean remove(Object object)
  - boolean removeAll(Collection<?> collection)
  - int size()
  - Object[ ] toArray()
  - <T> T[ ] toArray(T[ ] array)

Java – Collections Framework, Umgang mit Interfaces

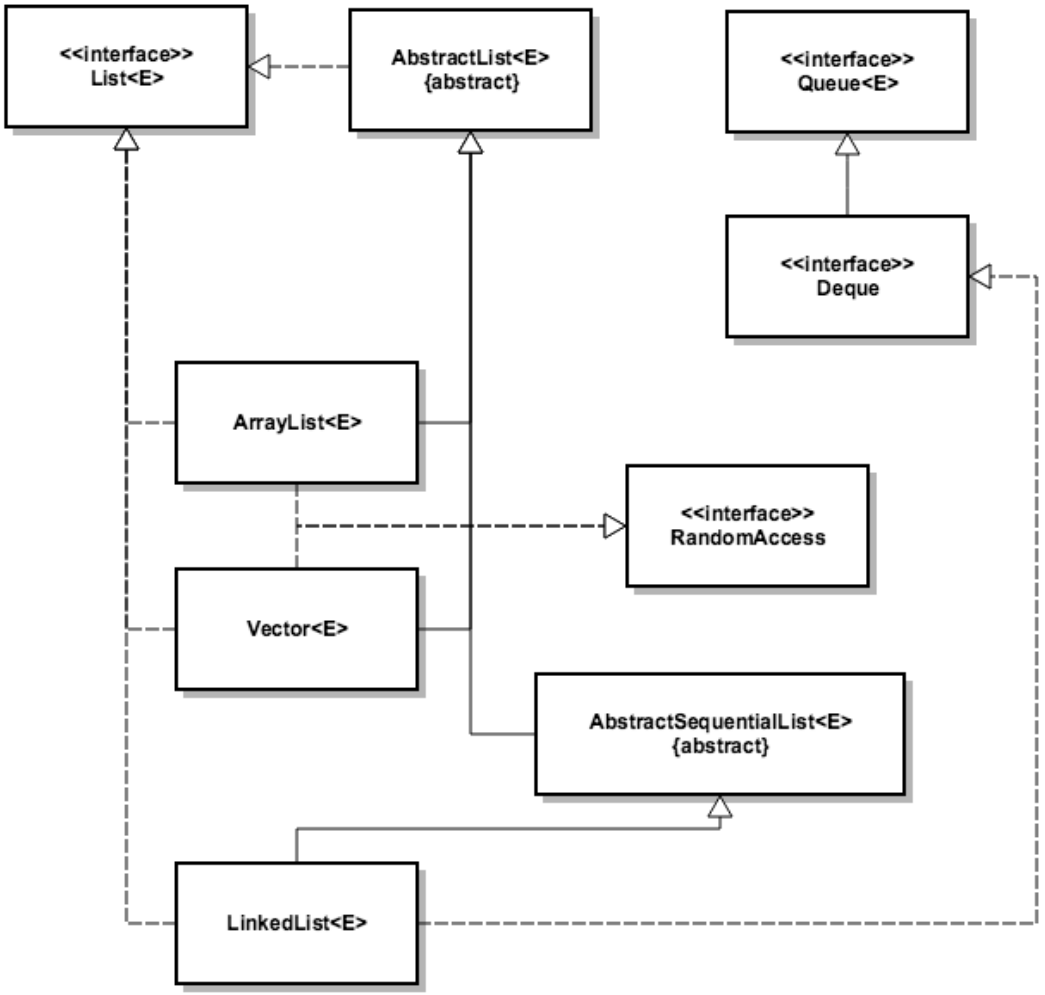
- Wenn die Methoden des Interface für die geforderte Aufgabe ausreichend sind sollte bei der Deklaration von Variablen immer der Typ des Interface genommen werden, also etwa:
  - Collection<String> aCollection = new ArrayList<String>();
- ... oder, falls doch Methoden des List Interfaces gebraucht werden:
  - List<String> aList = new ArrayList<String>();
- Das hat den Vorteil bei Bedarf eines anderen Collection-Typs die tatsächlich verwendete Collection nur bei der Definition austauschen zu müssen, also:
  - Collection<String> aCollection = new HashSet<String>();
  - List<String> aList = new LinkedList<String>();
- ... da im weiteren Code ja nur die Methoden des Interface genutzt werden konnten, fallen dort keine weiteren Änderungen an.

Java – Collections Framework, die Default-Implementierungen

- Zu allen Collection Interfaces existieren innerhalb des Framework abstrakte Klassen, die jeweils eine Basis/Implementierung der Methoden des Interfaces zu Verfügung stellen. Abgeleitete Klassen müssen somit nur noch die sinnvollen Aspekte überschreiben:
  - Collection – AbstractCollection
  - List – AbstractList
  - Set – AbstractSet
  - Queue – AbstractQueue
  - Map - AbstractMap

Java – Collection Framework, Listen

Java Collections Framework, Listen



Java – Collections Framework, java.util.List<E>

- Basis Interface für alle Collections vom Typ Liste
- Listen sind in Java indiziert, halten Elemente also an bestimmten Positionen vor, deshalb erweitert das List-Interface die Methoden des Collection-Interface um solche die mit oder auf dem Index arbeiten:
  - void add(int index, E element)
  - boolean addAll(int index, Collection<? extends E> collection)
  - E get(int index)
  - int indexOf(Object object)
  - E remove(int index) // return value ist hier das ersetzte Element!
  - E set(int index, E element)
  - int size() // gibt die Anzahl der Elemente zurück!
  - List<E> subList(int fromIndex, int toIndex)

Java – Collections Framework, java.util.ArrayList<E>

- Instanzierbare Implementierung des List Interface
- Implementiert das Random Access Interface
  - Die Methoden size, isEmpty, get, set, iterator und listOperator laufen in konstanter Zeit, Laufzeit von add ist linear abhängig von der Anzahl der Elemente n, also O(n)
- Im wesentlichen ein dynamischer Array
- Nicht Thread-safe, da unsynchronisiert
- ArrayList hat eine Kapazität, die Anzahl der Elemente des angegebenen Typs die aufgenommen werden können. Die Kapazität wird automatisch erweitert. Will man von Anfang an eine bestimmte Kapazität vorrätig halten, kann der entsprechende Konstruktor **ArrayList**(int initialCapacity) oder die Methode **ensureCapacity**(int minCapacity) benutzt werden
- Mit **trimToSize**() wird die Kapazität auf die Anzahl der enthaltenen Elemente gesetzt

Java – Collections Framework, java.util.Vector<E>

- Synchronized Variante des dynamischen Array
- Implementiert ebenfalls das Interface RandomAccess
- Für die Nutzung in einer Single-Thread Umgebung sollte bevorzugt ArrayList benutzt werden
- Auch Vector benutzt intern eine Kapazität und einen Wert capacityIncrement um den die Kapazität vergrößert wird, sobald der Platz ausgeht. Beides kann über den Konstruktor **Vector**(int initialCapacity, int capacityIncrement) bei der Erzeugung gesetzt werden. Die aktuelle Kapazität kann mit der Methode **capacity**() abgefragt werden.
- Mit setSize() kann die Anzahl der Elemente gesetzt werden,

Java – Collections Framework, java.util.LinkedList

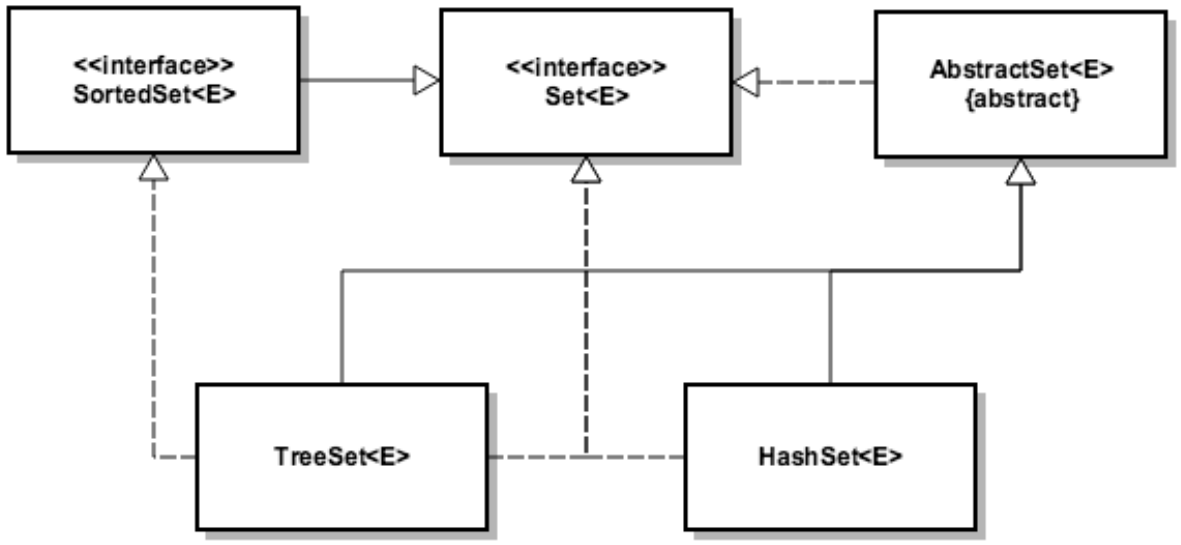
- Implementierung einer verlinkten Liste, hier einer doppelt verlinkten. Einfacher, dynamisch erweiterbarer Datentyp, der aus Knoten (nodes) besteht mit zwei Attributen bei der einfach verketteten Liste und drei Attributen bei der doppelt verketten, dem zu haltenden Element und einem Zeiger auf den nächsten Knoten, bzw. noch dem vorigen Knoten bei der doppelt verketteten Liste.
- Alle Operationen, wie suchen, einfügen, etc. fangen vorne, beim ersten Element an (einfach), bzw. auch hinten, beim letzen Element (doppelt). Das macht die Operationen bei langen Listen unter Umständen sehr "teuer".
- Die Collections Implementierung kann gleichzeitig noch als Deque benutzt werden

Java – Collections Framework, Queue, Deque und Stack

- Im Collections Framework finden sich auch Implementierungen bzw. Optionen zu Implementierung der Abstrakten Datentypen (ADT) Stack (Stapel, LIFO), Queue (Warteschlange, FIFO) und Deque (Warteschlange mit zwei Enden! Kombination aus LIFO und FIFO)
- java.util.Stack
  - Vector basierte Implementierung
  - Besser selber implementieren via Deque Interface
- java.util.PriorityQueue
  - Prioritätengesteuerte Queue-Implementierung
  - Bei Bedarf an reiner Queue, selber implementieren via Deque Interface
- java.util.ArrayDeque
  - Dynamische Array-basierte Implementierung des Deque Interface

Java – Collections Framework, Sets

Java Collections Framework, Sets



Java – Collections Framework, java.util.Set

- Basis-Interface für alle Setimplementierungen
- Ist die Abstraktion des mathematischen Mengenbegriffs
- Elemente können nur einmal enthalten sein
- Definiert u.a. folgende Methoden:
  - boolean add(E element), fügt element zum Set hinzu wenn es noch nicht vorhanden war und liefert genau dann true zurück, ansonsten bleibt das Set unverändert und false wird zurückgegeben
  - boolean addAll(Collection<? extends E> collection), fügt alle Elemente aus collection, die nicht im Set enthalten sind dem Set hinzu und liefert true, sobald das Ursprungsset verändert wurde. Ist collection ein Set liefert die Operation die Vereinigungsmenge
  - boolean containsAll(Collection<? extends E> collection), liefert true wenn alle Elemente aus collection im Ursprungsset enthalten sind, wenn collection ein Set ist dann liefert der Aufruf true bei einer Untermenge
  - boolean retainAll(Collection<?> collection), entfernt aus dem Ursprungsset alle Elemente die nicht in collection enthalten sind. Ist collection ein Set entspricht dies der Schnittmengenbildung. Liefert true, wenn das Ursprungsset verändert wurde

Java – Collections Framework, java.util.Set

- Fortsetzung Methoden:
  - boolean removeAll(Collection<?> collection), entfernt alle Elemente aus collection im Ursprungsset, liefert true sobald das Ursprungsset verändert wurde. Ist collection ein Set liefert die Operation die Differenzmenge.

Java – Collections Framework, java.util.HashSet

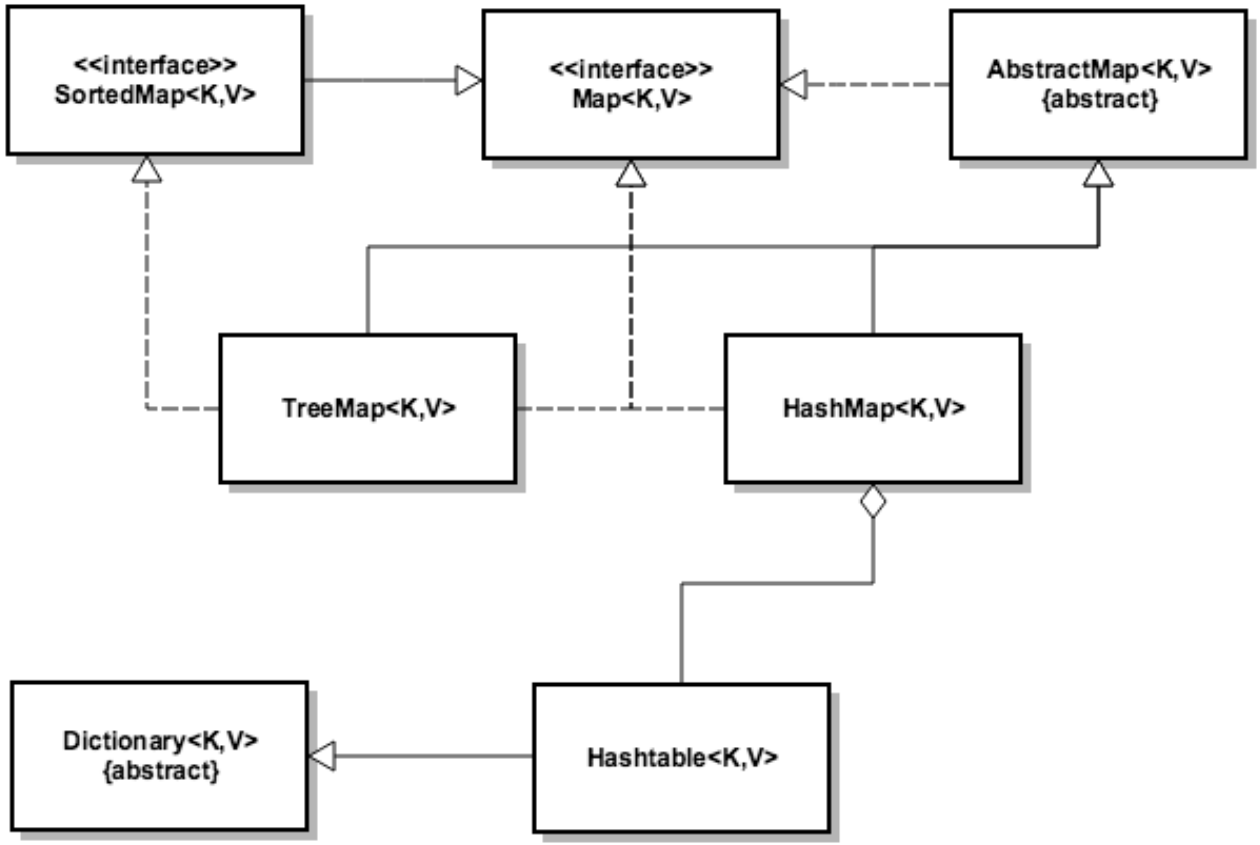
- Implementierung des Set Interface basierend auf einer hash table
- Die Klasse bietet konstante Laufzeit für die Operationen add, remove, contains und size
- Auch hier ist durch die Implementierung über eine hash table, wie bei HashMap die richtige Dimensionierung der Werte für InitialCapacity und loadFactor wichtig (siehe dort)

Java – Collections Framework, java.util.TreeSet

- Eine TreeMap basierte Implementierung des Set Interface.
- Sortierte Collection. Sortierung erfolgt über einen Comparator oder die natürliche Ordnung der Objekte
- Die Basisoperationen, wie add, remove und contains garantieren eine Laufzeit O(log n)

Java – Collections Framework, Maps

Java Collections Framework, Maps



Java – Collections Framework, java.util.Map

- Basis Interface für alle schlüsselbasierten Datenstrukturen im Collections Framework
- Intern werden die Schlüssel-Wert Paare in sogenannten Entry-Objekten gehalten. Entry ist in Map als inneres statisches Interface definiert.
- Ein Schlüssel kann nur einmal enthalten sein, wird mit put ein Wert zu einem schon vorhanden Schlüssel hinzugefügt, wird der alte Wert überschrieben
- Somit definiert das Interface Operationen in Abhängigkeit zu Schlüsseln und den darunter abgelegten Werten:
  - boolean containsKey(Object key), liefert true wenn der Schlüssel key bereits in der Map enthalten ist
  - boolean containsValue(Object value), liefert true wenn der Wert value bereits in der Map enthalten ist
  - Set<Map.Entry<K,V>> entrySet() liefert ein Set mit allen Schlüssel-Wert Paaren zurück

Java – Collections Framework, java.util.Map

- Fortsetzung Methoden:
  - V get(Object key) liefert den unter key abglegten Wert zurück
  - Set<K> keySet() gibt ein Set mit allen in der Map vorhandenen Schlüsseln zurück
  - V put(K key, V value), legt den Wert value unter key ab. War schon ein Wert unter key hinterlegt wird der alte Wert zurückgegeben und durch den neuen überschrieben. War noch kein Wert unter key abgelegt wird der neue Wert unter dem Schlüssel eingetragen und null zurückgegeben.
  - V remove(Object key), löscht den Eintrag unter key und gibt den assoziierten Wert zurück oder null, wenn der key nicht vorhanden war
  - Collection<V> values() gibt eine Collection mit allen Werten dieser Map zurück

Java – Collections Framework, java.util.HashMap

- Hashtable basierte Implementierung des Map-Interface
- Bei einer guten Verteilung der Schlüssel über die Hashfunktion erhält man eine Datenstruktur mit konstantem Zeitverhalten für die Einfüge- (put(K,V)) und Herausgabe-Funktion (get(K)).
- Bei HashMap ist die Initialisierung mit der richtigen Anfangsgröße initialCapacity und dem geeigneten Faktor als Beschreibung für die Auslastung der HashMap der loadFactor wichtig um teure rehash Operationen, die bei Vergrößerung der der HashMap anfallen zu vermeiden. Faustregel zum Vermeiden von rehashing:
  - initialCapacity > maximumEntries / loadFactor
  - Ein guter Wert für den loadFactor ist 0.75

Java – Collections Framework, java.util.Hashtable

- Threadsichere Version der HashMap, ähnlich wie Vector erst mit Java 1.2 zum Collections Framework dazugekommen.
- Für das Laufzeitverhalten gelten die gleichen Bedingungen wie für die HashMap
- Für massive Nebenläufigkeit sollte ConcurrentHashMap aus dem Paket java.util.concurrent benutzt werden.

Java – Collections Framework, Aufgabe

- Untersucht je zwei Realisierungen des List-, Map- und Set-Interfaces auf ihr Laufzeitverhalten, für folgende Operationen mit n=1, 1000, 10000, 100000, ...
  - Hinzufügen von n Elementen in eine neue, bzw. schon bestehende Collection mit n Elementen, sowohl als einzelne Elemente, als auch, da wo unterstützt über das Hinzufügen einer Collection gleichenTyps
  - Analog dazu das Finden, bzw. Entfernen eines Elements in einer Collection mit n Elementen (hier lohnt die Untersuchung natürlich erst ab n = 1000 ;-)
- Die einzelnen Fälle sollten parametrisiert laufen und mindestens 10 mal wiederholt werden
- Interessant sind hier sowohl die Einzelbetrachtung mit bester und schlechtester Fall, als auch das arithmetische Mittel über alle Läufe eines Falls
- Als Werte zum Einfügen sollten einmal alle Fälle mit primitive Datentypen durchlaufen werden und einmal mit Objekten, wie Strings oder was eigenes, wie z.B. Person
- Welche Collections sind für welche Anwendungsfälle und Mengen von Elementen besonders performant. Wie wirkt sich das Setzen einer initialen Kapazität auf das Laufzeitverhalten aus?

DESY

Java – GUI-Programmierung, Weshalb?

- Komplexität vor dem Nutzer verbergen
- Hemmschwellen abbauen
- Bei guter Oberfläche, Intuition fördern
- Benutzern nur auf spezielle Teile der Anwendung Zugriff gewähren
- Überprüfen von Nutzereingaben und Abfangen von unsinnigen/ böswilligen Eingaben
- Klare Trennung von Bedienung, Daten und eigentlicher Anwendung -> MVC-Pattern (Model-View-Controller)

Java – GUI-Programmierung

- AWT – Abstract Widget Toolkit (SUN)
  - Nutzt die Grafikbibliotheken des Betriebssystems/Desktop-Environment über Wrapperklassen, sogenannte Peers
  - Oberflächen sehen auf jedem System anders aus
  - Da die Funktionalität auf verschieden Systemen unterschiedlich, nur kleinster gemeinsamer Nenner implementiert (Low-Level-Komponenten)
- SWT – Standard Widget Toolkit (IBM)
  - Setzt wie AWT auf native Elemente des Systems, versucht aber High-Level-Komponenten (Toolkit JFace) zur Verfügung zu stellen, die wo nicht vorhanden, "emuliert" werden müssen
  - Muß extra installiert werden, kommt als Bibliothek in Eclipse mit

Java – GUI-Programmierung

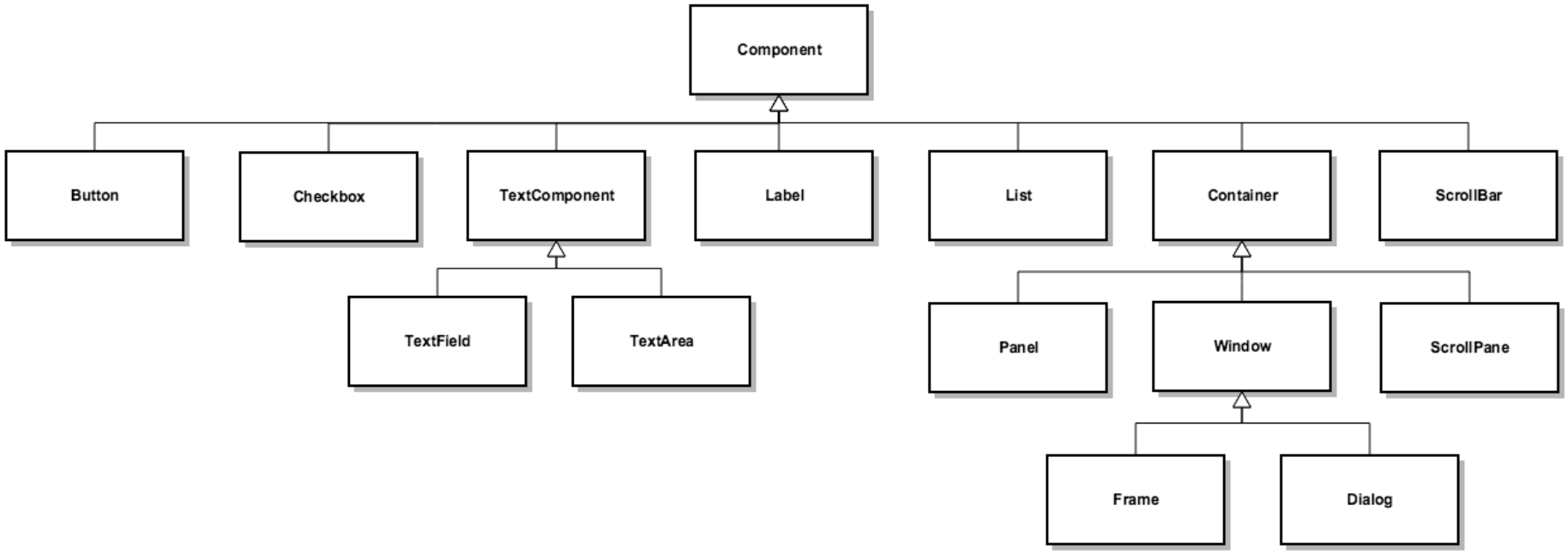
- Swing
  - Bis auf das Eventsystem unabhängig vom Betriebssystem, Komponenten werden im Java Interpreter gerendert
  - Aussehen der Komponenten vollständig anpassbar
  - High-Level Komponenten wie Standard-Dialoge, FileChooser, ColorChooser
  - Bringt ein Systemübergreifendes Look & Feel mit
  - Kann allerdings auf Look & Feel des Betriebssystems zugreifen
- JavaFX - seit 2014/2015 vollständiges GUI-Toolkit
  - ursprünglich von SUN entwickelt um Flash und CSS abzulösen ;-)
  - Als Überlegungen zu einer Modernisierung von Swing-Komponenten anstanden, nahm Oracle die Ideen zu JavaFX wieder auf
  - Vollständig unabhängig von AWT und Swing, Platformübergreifend, Einbettung von HTML-Inhalten, XML-basierte Erstellung von Oberflächen, Animationen möglich
  - Scene-Graph basierte Erstellung von 2D- und 3D-Inhalten

Java – AWT, Wichtige Klassen

- Nach wie vor wichtige AWT-Klassen (auch unter Swing) sind:
  - Das AWT-Eventsystem und alle Klassen unter java.awt.event
  - java.awt.Color
  - java.awt.Font
  - java.awt.Point
  - java.awt.Dimension
  - java.awt.Rectangle
  - java.awt.Graphics
  - java.awt.Toolkit
  - Die Layout-Klassen unter java.awt, wie FlowLayout, GridLayout, GridBagLayout und BorderLayout

Java – AWT-Komponenten, Übersicht

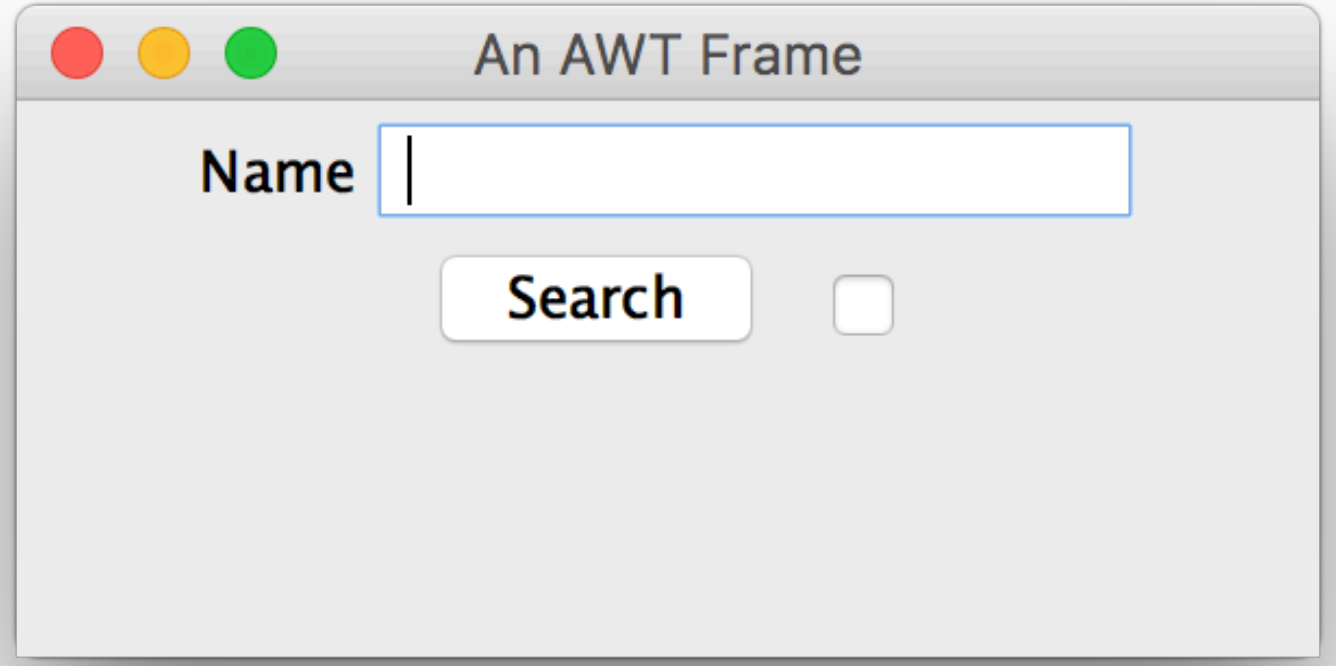
Java AWT Komponenten, package java.awt



Java – AWT Frame, Beispiel

```
/* AWTFrame.java */

package examples.gui;

import java.awt.Frame;
import java.awt.FlowLayout;
import java.awt.Checkbox;
import java.awt.Frame;
import java.awt.Label;
import java.awt.TextField;
import java.awt.Button;

public class AWTFrame extends Frame{

    public AWTFrame(){
        this.setTitle("An AWT Frame");
        this.setSize(300,150);
        this.setLayout(new FlowLayout());
        this.add(new Label("Name"));
        this.add(new TextField(20));
        this.add(new Button("Search"));
        this.add(new Checkbox());
    }

    public static void main(String... args){
        AWTFrame frame = new AWTFrame();
        frame.setVisible(true);
    }
}
```

Java – AWT Frame, Screenshot



Java – Adapter-Klassen

- Adapter Klassen werden in Java abstrakte Klassen genannt die ein oder mehrere Interfaces implementieren und dabei sinnvolle Default-Implemetierungen für die Methoden des Interfaces bieten.
- Das ist vorteilhaft wenn ein Interface mit vielen Methoden genutzt werden soll/muß, aber nur eine der Methoden benutzt wird – nur diese muß dann überschrieben werden.
- Viel in der GUI-Programmierung unter Java genutzte Adapter sind die unter java.awt.event liegenden Klassen ComponentAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter und WindowAdapter

Java – AWT & Swing, Events und Listener

- java.awt.event:
- ActionEvent <-> ActionListener
- KeyEvent <-> KeyListener, KeyAdapter
- MouseEvent <-> MouseListener, MouseMotionListener
- MouseWheelEvent <-> MouseWheelListener
- MouseEvent, MouseWheelEvent <-> MouseAdapter
- javax.swing:
- Action, AbstractAction
  - Nur für einige Komponenten verfügbar (JButton, JMenuItem, ...)
  - Implementiert ActionListener Interface und hält darüber hinaus noch Informationen über die Komponente, wie Name, Icon, Tastatur-Shortcut)
  - Vorteilhaft bei Verwendung gleicher Funktionalität an verschiedenen Stellen, wie als Button, Menüeintrag in Menüleiste und Kontextmenü

Java – AWT & Swing, Implementierung Listener

- Anonyme Klasse, als Parameter im Aufruf von:

```
aComponent.addXyzListener(new XyzListener(){
      public void xyzAction(XyzEvent e){
          doThings()  ...   }});
```

- Implementierung der entsprechenden Interfaces bei abgeleiteten Komponentenklassen:

```
class MySpecialButton extends JButton implements ActionListener{
    ...
    public void actionPerformed(ActionEvent e){...}
}
```

- Als eigenständige Listener-Klasse:

```
class MySpecialKeyListener implements KeyListener{
     public void keyTyped(KeyEvent e){...}
     public void keyPressed(KeyEvent e){...}
     public void keyReleased(keyEvent e){...}
}
```

Java – AWT & Swing, Implementierung Listener

- Spezialfall ActionListener als Action bzw. von AbstractAction abgeleitet
- Macht Sinn, wenn eine Aktion von verschiedenen Stellen ausgelöst werden soll/muß (Jbutton, MenuItem, etc.)
- Liste der unterstützten Komponenten findet sich in der Java Doku zur Klasse AbstractAction
- Beim Ableiten von AbstractAction sollte dessen Konstruktor mit super() aufgerufen und initialisiert werden, dieser erwartet zwei Parameter, einen String als sichtbarer Titel der nutzenden Komponente und ein ImageIcon (kann auch null sein).
- Weitere Properties werden mit putValue(key, value) gesetzt. Standard sollten die folgenden Properties sein:
  - SHORT_DESCRIPTION, String – Text für Tooltip
  - MNEMONIC_KEY, Integer aus KeyEvent-Konstante – Tasten Ansteuerung der Komponente, sofern sichtbar, wenn der Keyode im Namen vorkommt unterstrichen

Java – Swing: ActionListener, Beispiel

```
class ThemesButtonListener implements ActionListener{

    public void actionPerformed(ActionEvent e){
        String actionName = ((AbstractButton)e.getSource()).getText();
        SwingDemo3 rootComponent = SwingDemo3.getInstance();
        rootComponent.setLookAndFeel(
                      "javax.swing.plaf.metal.MetalLookAndFeel");
        if(actionName.equals("Metal Theme")){
            MetalLookAndFeel.setCurrentTheme(
                      new DefaultMetalTheme());
        }else if(actionName.equals("Ocean Theme")){
            MetalLookAndFeel.setCurrentTheme(new OceanTheme());
        }else{
            MetalLookAndFeel.setCurrentTheme(new ColorTheme());
            UIManager.put("swing.boldMetal", Boolean.FALSE);
        }
        try{
            UIManager.setLookAndFeel(new MetalLookAndFeel());
        }catch(UnsupportedLookAndFeelException ex){
            ex.printStackTrace();
        }
        SwingUtilities.updateComponentTreeUI(rootComponent);
    }
}

class MenuItemListener implements ActionListener{

    public void actionPerformed(ActionEvent e){
        String actionName = ((AbstractButton)e.getSource()).getText();
        System.out.println(actionName + " was selected!");
    }
}

...
JButton defaultThemeButton = new JButton("Metal Theme");
JButton oceanThemeButton = new JButton("Ocean Theme");
JButton colorThemeButton = new JButton("Color Theme");

defaultThemeButton.addActionListener(new ThemesButtonListener());
oceanThemeButton.addActionListener(new ThemesButtonListener());
colorThemeButton.addActionListener(new ThemesButtonListener());
```

Java – Swing: KeyListener, Beispiel

```
class SampleKeyListener implements KeyListener{

    public void keyTyped(KeyEvent e){
        // only called if the key can be
        // converted into an Unicode character
        // so it's not called for ESC, F1 and so on
        JTextArea a = (JTextArea)e.getComponent();
        String keyText = KeyEvent.getKeyText(e.getKeyCode());
        a.append("KEY TYPED: " + keyText + "\n");
        a.append(e.paramString() + "\n\n");
    }

    public void keyPressed(KeyEvent e){
        JTextArea a = (JTextArea)e.getComponent();
        String keyText = KeyEvent.getKeyText(e.getKeyCode());
        a.append("KEY PRESSED: " + keyText + "\n");
        a.append(e.paramString() + "\n\n");
    }

    public void keyReleased(KeyEvent e){
        JTextArea a = (JTextArea)e.getComponent();
        String keyText = KeyEvent.getKeyText(e.getKeyCode());
        a.append("KEY RELEASED: " + keyText + "\n");
        a.append(e.paramString() + "\n\n");
    }
}

...
JTextArea keyText = new JTextArea(40,80);
keyText.setEditable(false);
keyText.addKeyListener(new SampleKeyListener());
```


DESY

Java – Swing: Full MouseListener, Beispiel

```
class FullMouseListener implements
          MouseListener, MouseMotionListener, MouseWheelListener{
    // java.awt.event.MouseAdapter implements all Mouse Event Listener
    // MouseListener, MouseMotionListener and MouseWheelListener

    public void mouseClicked(MouseEvent e){
        Point p = e.getLocationOnScreen();
        String s = e.paramString();
        JTextArea a = (JTextArea)e.getComponent();
        a.append("Mouse was CLICKED (pressed & released) at x="
                      + p.getX() + ", y=" + p.getY()
                      + " and created this event: " + s + "\n");
    }
    public void mouseDragged(MouseEvent e){
        Point p = e.getLocationOnScreen();
        String s = e.paramString();
        JTextArea a = (JTextArea)e.getComponent();
        a.append("Mouse has DRAGGED at x="
                      + p.getX() + ", y=" + p.getY()
                      + " and created this event: " + s + "\n");
    }
    public void mouseEntered(MouseEvent e){
        Point p = e.getLocationOnScreen();
        String s = e.paramString();
        JTextArea a = (JTextArea)e.getComponent();
        a.append("Mouse has ENTERED a JComponent at x="
                      + p.getX() + ", y=" + p.getY()
                      + " and created this event: " + s + "\n");
    }
    public void mouseExited(MouseEvent e){
        Point p = e.getLocationOnScreen();
        String s = e.paramString();
        JTextArea a = (JTextArea)e.getComponent();
        a.append("Mouse has EXITED a JComponent at x="
                       + p.getX() + ", y=" + p.getY()
                       + " and created this event: " + s + "\n");
    }
```


DESY

Java – Swing: Full MouseListener, Beispiel

```
public void mouseMoved(MouseEvent e){
        Point p = e.getLocationOnScreen();
        String s = e.paramString();
        JTextArea a = (JTextArea)e.getComponent();
        a.append("Mouse was MOVED at x="
                        + p.getX() + ", y=" + p.getY()
                        + " and created this event: " + s + "\n");
    }
    public void mousePressed(MouseEvent e){
        Point p = e.getLocationOnScreen();
        String s = e.paramString();
        JTextArea a = (JTextArea)e.getComponent();
        a.append("Mouse was PRESSED at x="
                        + p.getX() + ", y=" + p.getY()
                        + " and created this event: " + s + "\n");
    }
    public void mouseReleased(MouseEvent e){
        Point p = e.getLocationOnScreen();
        String s = e.paramString();
        JTextArea a = (JTextArea)e.getComponent();
        a.append("Mouse was RELEASED at x="
                        + p.getX() + ", y=" + p.getY()
                        + " and created this event: " + s + "\n");
    }
    public void mouseWheelMoved(MouseWheelEvent e){
        Point p = e.getLocationOnScreen();
        String s = e.paramString();
        JTextArea a = (JTextArea)e.getComponent();
        a.append("MouseWheel was MOVED at x="
                       + p.getX() + ", y=" + p.getY()
                       + " and created this event: " + s + "\n");
    }
}
...
mouseText = new JTextArea(40,80);
FullMouseListener fullMouseListener = new FullMouseListener();
mouseText.addMouseListener(fullMouseListener);
mouseText.addMouseMotionListener(fullMouseListener);
mouseText.addMouseWheelListener(fullMouseListener);
```

Java – Swing: Action, Beispiel

```
class InformationAction extends AbstractAction{

    public InformationAction(){
        super("Information", new ImageIcon(
                "~/Code/Java/examples/resource/enterprise03g.png"));
        this.putValue(SHORT_DESCRIPTION,
                      "Shows a JOptionPane information message");
        this.putValue(MNEMONIC_KEY,
                      new Integer(KeyEvent.VK_I));
    }

    public void actionPerformed(ActionEvent e){
        JOptionPane.showMessageDialog(null,
                    "An JOptionPane Information Message",
                    "Information",
                    JOptionPane.INFORMATION_MESSAGE);
    }
}

...

this.add(new JMenuItem(new InformationAction()));

...

JButton informationButton = new JButton(new InformationAction());
```

Java - Swing

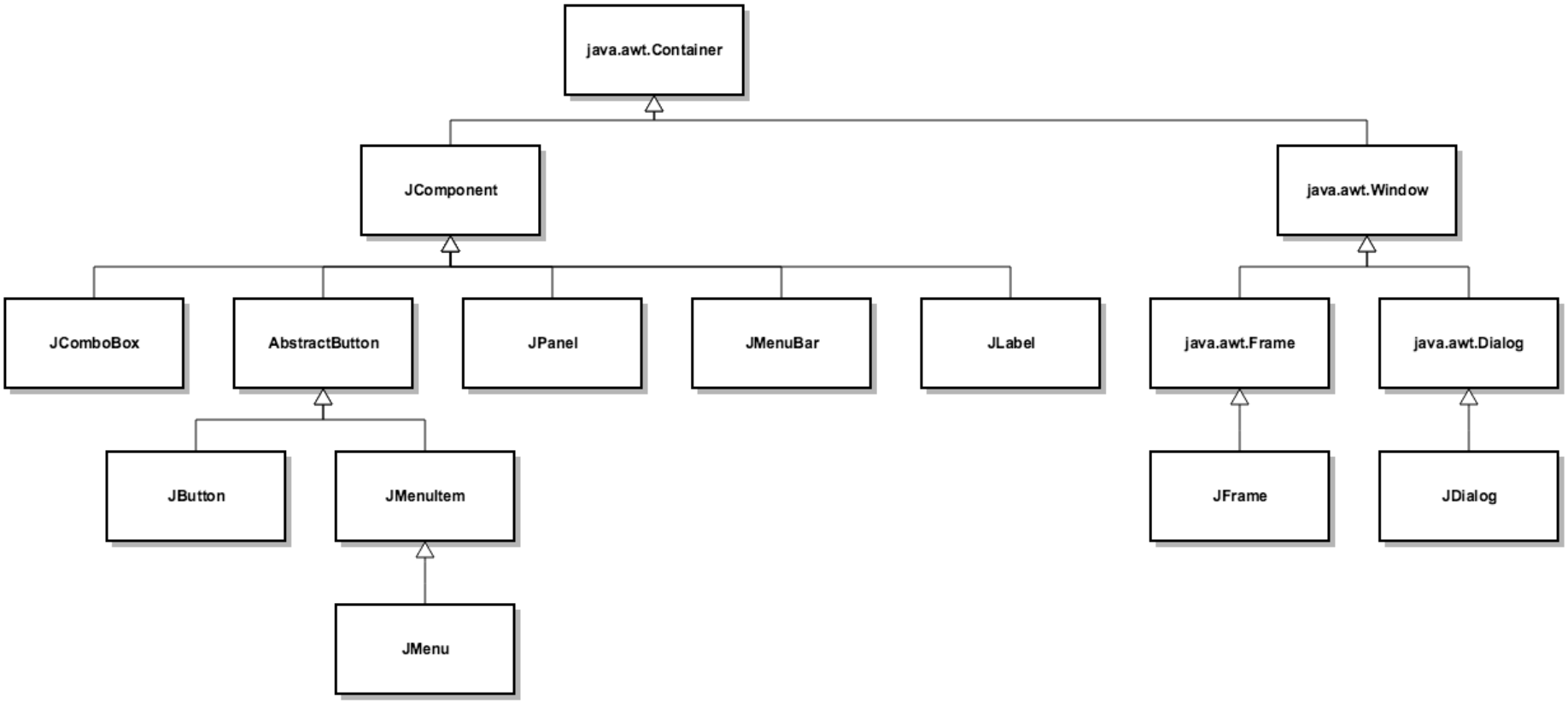
- Swing-Komponenten sind nicht Threadsafe
- Jedes von JComponent abgeleite Objekt kann nur einmal innerhalb der Anwendung verwendet werden. Wird eine Referenz auf ein solches Objekt verschiedenen Komponenten hinzugefügt, wird es schlußendlich nur in der Komponente auftauchen, der es zuletzt hinzugefügt wurde. Soll eine Komponente, beispielsweise
  JButton mySpecialOKButton = new JButton("Ok");
  mehrfach benutzt werden, müssen "echte" Kopien angefertigt werden.
- Swing Anwendungen schreiben besteht im wesentlichen aus:
  - Top-Level Container erzeugen und konfigurieren
  - Komponenten erzeugen, konfigurieren (evtl. noch mit Listenern versehen) und mit add() anderen Komponenten hinzufügen
  - Alles was noch keiner Komponente hinzugefügt wurde mit add dem Top-Level Container hinzufügen
  - Den Top-Level Container mit pack() rendern und mit setVisible(true) anzeigen

Java – Swing, Top-Level Container

- Top-Level Container sind Klassen, die einer JComponent nicht mit add() hinzugefügt werden können
- Werden mehrere Top-Level-Container in einer Anwendung genutzt werden sie hierarchisch geordnet, d.h. ein Top-Level Container fungiert als Parent für die anderen (Konstruktoraufruf)
- In Swing existieren drei Top-Level Container:
  - JFrame
  - JDialog
  - JApplet

Java – Swing-Komponenten, Übersicht

Java Swing Komponenten, package javax.swing, wenn nicht angegeben

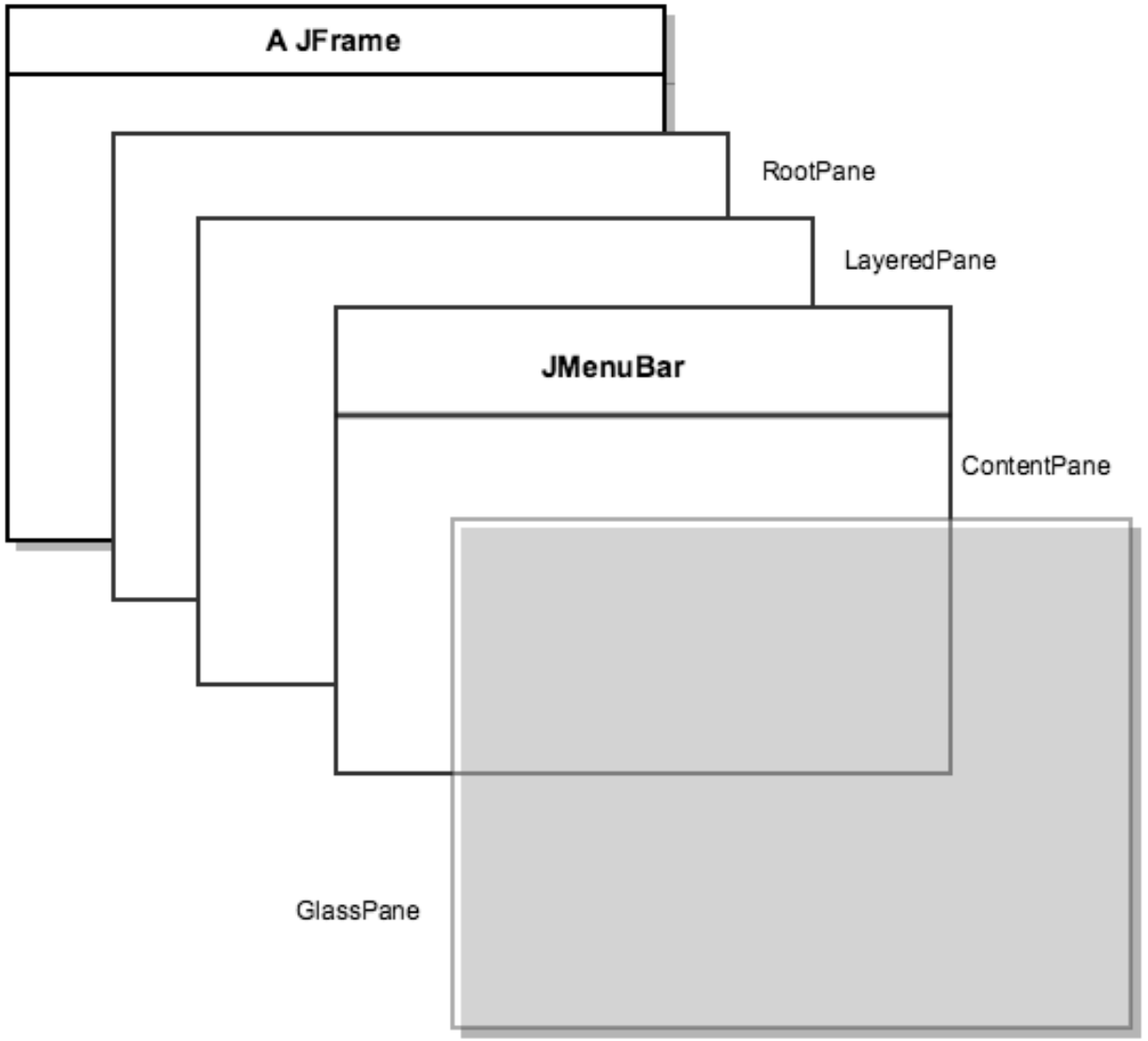


Java – Swing, Zeichen-Ebenen die Panes

- RootPane
  - Hier wird die Titelleiste des Frames gerendert
  - Hierauf liegt die LayeredPane
- LayeredPane
  - Auf der LayeredPane kommt als erstes die MenuBar und die ContentPane
  - Weitere Komponenten (bzw. Lagen), die hier zum Liegen kommen sind Dialoge und PopupMenu, die beim sichtbar werden, die Inhalte der ContentPane überlagern
- ContentPane
  - Standardmäßig kommen hier die Komponenten der Oberfläche zum Liegen
  - In den ersten Swing-Versionen mußten von JComponent abgeleitete Komponenten mit aFrame.getContentPane().addComponent(aJComponent) hinzugefügt werden, mittlerweile gibt es dazu den Shortcut aFrame.add(aJComponent)
- GlassPane
  - Durchsichtige Zeichenfläche, voreingestellt nicht angezeigt, für Komponenten die alles Überlagern müssen/sollen

DESY

Java – Swing, Zeichen-Ebenen die Panes



Java – Swing: Fenster mit JFrame

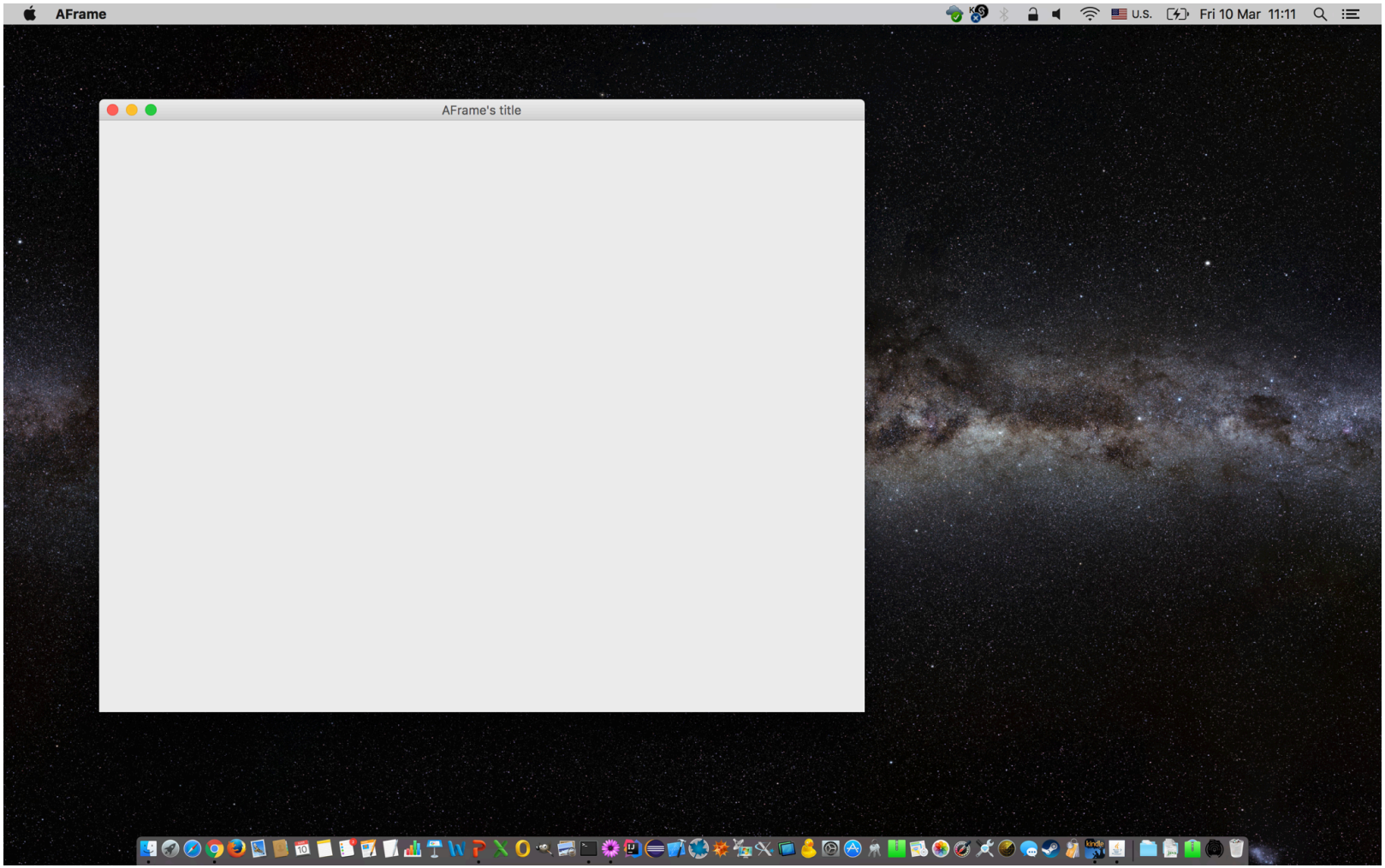
- Swing Top-Level Komponente
- Erzeugung auf zwei Arten, durch Ableitung von JFrame und setzen der Komponenten im Konstruktor (meistens für das "Hauptfenster" der Anwendung) oder durch Konstruktoraufruf von JFrame und setzen der Komponenten über die Methoden der Instanz (meistens für weitere "Kind"-Fenster der Anwendung
- Beim Betätigen des Schließen-Buttons wird die Anwendung nicht vollständig geschlossen, der JFrame wird in der default-Einstellung nur auf unsichtbar geschaltet, soll der Schließen-Button die Anwendung wirklich beenden, muß die **DefaultCloseOperation** explizit gesetzt werden:

  aFrame.**setDefaultCloseOperation**(JFrame.EXIT_ON_CLOSE);

Java – Swing: Fenster mit JFrame, Beispiel

```java
/* AFrame.java */

package examples.gui;

import javax.swing.JFrame;

public class AFrame extends JFrame{

    public AFrame(){
        super("AFrame's title");
        this.setLocation(100,100);
        this.setSize(800,640);
    }

    public static void main(String... args){
        AFrame aFrame = new AFrame();
        aFrame.setVisible(true);
    }
}
```

Java – Swing: Fenster mit JFrame, Beispiel



Java – Swing: JFrame, Rendern und Starten

- Die tatsächliche Größe des JFrames hängt davon ab welche Größen im Renderprozess für die auf ihm befindlichen Komponenten ermittelt werden, mit aFrame**.pack()** startet man den RenderProzess und nimmt die resultierende Größe mit aFrame**.setSize()** und aFrame**.setBounds()** kann man versuchen die Größe zu fixieren.
- Die letzte Operation auf dem JFrame sollte immer der Aufruf von aFrame.**setVisible(true)** sein
- Um Sicherzustellen das die Erzeugung der GUI auf einem speziellen Thread im Java-Interpreter läuft, dem AWTEventDispatchThread empfiehlt die Doku die Swing-GUI Erzeugung über die statischen Methoden:

  SwingUtilities.invokeLater(aRunnable) – asynchron bzw.
  SwingUtilities.invokeAndWait(aRunnable) – synchron
- Damit sieht das erste Beispiel wie folgt aus:

Java – Swing: Fenster mit JFrame, Beispiel 2

```java
/* AFrame2.java */

package examples.gui;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class AFrame2 extends JFrame{

    public AFrame2(){
        super("AFrame2's title");
        this.setLocation(100,100);
        this.setSize(800,640);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //this.pack(); //ohne Inhalt nur Titelleiste
    }

    public static void createGUI(){
        AFrame2 aFrame = new AFrame2();
        aFrame.setVisible(true);
    }

    public static void main(String... args){
        SwingUtilities.invokeLater(new Runnable(){
            public void run(){
                createGUI();
            }
        });
    }
}
```

Java – Swing: Menüs

- Es gibt zwei Arten von Menüs, "statische" Menüs in der Menüleiste und kontextabhängige Menüs, meistens bei rechtem Mausklick auf eine Komponente der Anwendung
- Menüs in der Menüleiste
  - JMenuBar erzeugen
  - JMenu erzeugen
  - JMenuItem erzeugen, mit Action oder ActionListener versehen und mit add() dem JMenu hinzufügen
  - JMenu mit add() der JMenuBar hinzufügen und dann dem Frame mit setJMenuBar() die JMenuBar hinzufügen
- Kontextmenüs
  - MouseListener schreiben (MouseAdapter, nur mousePressed() interessant), Trigger ist myMouseEvent.isPopupTrigger(), default rechte Maustaste!
  - Im MouseListener JPopupMenu erzeugen und JMenuItems hinzufügen
  - Listener mit addMouseListener() an eine Komponente hängen

Java – Swing: Menüs – JMenuBar, Beispiel

```
JMenuBar internalMenuBar = new JMenuBar();
JMenu fileMenu = new JMenu("File");
JMenu helpMenu = new JMenu("Help");

JMenuItem open = new JMenuItem("Open");
open.addActionListener(new MenuItemListener());
JMenuItem save = new JMenuItem("Save");
save.addActionListener(new MenuItemListener());
JMenuItem exit = new JMenuItem("Exit");
exit.addActionListener(new MenuItemListener());
JMenuItem help = new JMenuItem("Help");
help.addActionListener(new MenuItemListener());
JMenuItem about = new JMenuItem("About");
about.addActionListener(new MenuItemListener());

fileMenu.add(open);
fileMenu.add(save);
fileMenu.add(exit);
helpMenu.add(help);
helpMenu.add(about);
internalMenuBar.add(fileMenu);
internalMenuBar.add(helpMenu);
...
// this ist hier der JFrame
this.setJMenuBar(internalMenuBar);
```

Java – Swing: Menüs – JPopupMenu, Beispiel

```
class DialogPopupMenu extends JPopupMenu{
    public DialogPopupMenu(){
        this.add(new JMenuItem(new ErrorAction("Show Error", null,
                                "Shows an error message",
                                new Integer(KeyEvent.VK_E))));
        this.add(new JMenuItem(new InformationAction()));
        this.add(new JMenuItem(new YesNoAction()));
        this.add(new JMenuItem(new YesNoCancelAction()));
        this.add(new JMenuItem(new ModalDialogAction()));
        JMenuItem fileMenuItem = new JMenuItem(
                 new FileChooserAction());
        fileMenuItem.setAccelerator(
                  KeyStroke.getKeyStroke(KeyEvent.VK_1,
                         ActionEvent.ALT_MASK));
        this.add(fileMenuItem);
    }
}

class DialogPopupListener extends MouseAdapter{
    public void mousePressed(MouseEvent e){
        if(e.isPopupTrigger()){
            DialogPopupMenu popup = new DialogPopupMenu();
            popup.show(e.getComponent(), e.getX(), e.getY());
        }
    }
}

JPanel dialogPanel = new JPanel();
dialogPanel.addMouseListener(new DialogPopupListener());
```


DESY

Java – Swing: Layouts und LayoutManager

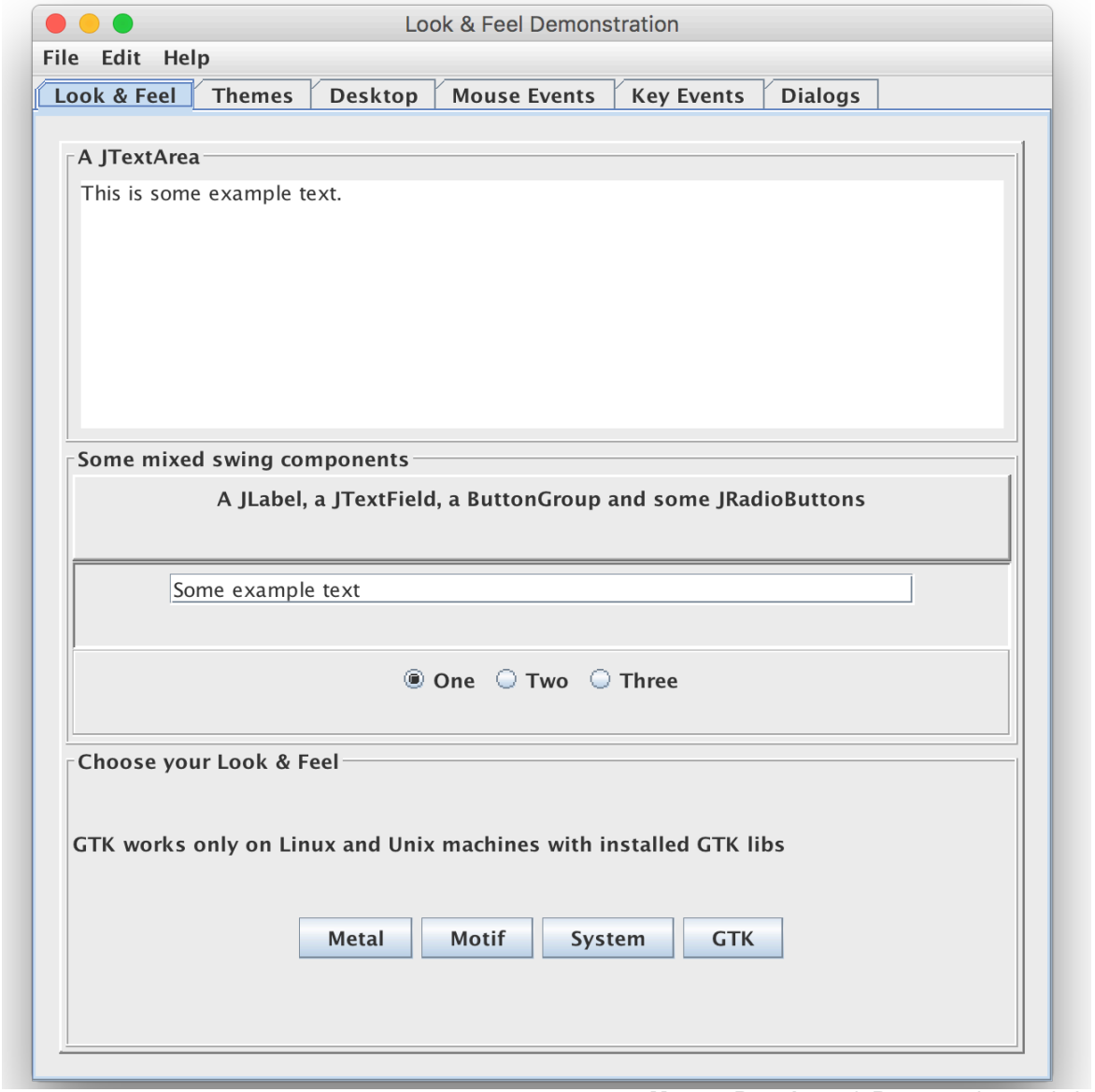
- Alle von java.awt.Container und damit in Swing alle von javax.swing.JComponent abgeleiteten Klassen haben eine Methode setLayout(java.awt.LayoutManager layout) mit der das Layout, also die Art und Weise wie die Komponenten innerhalb dieser Komponente zueinander angeordnet werden sollen, gesetzt werden kann.
- java.awt.LayoutManager ist ein Interface, dies wird von den allgemeinen LayoutManagern unter java.awt wie beispielsweise FlowLayout und GridLayout implementiert, aber auch von speziellen LayoutManagern, die oft als innere Klasse der Komponente für deren Layout sie zuständig ist, definiert werden, z.B. BasicTabbedPaneUI.TabbedPaneLayout oder JRootPane.RootLayout
- Alle Komponenten haben ein voreingestelltes Layout, bei JFrame und JPanel ist dies das FlowLayout

Java – Swing: Layouts und LayoutManager, Beispiele

```
JPanel allComponentsPanel = new JPanel();
allComponentsPanel.setLayout(new GridLayout(3,1));
...
JPanel textAreaPanel = new JPanel();
textAreaPanel.add(textArea);
...
JPanel mixedPanel = new JPanel();
mixedPanel.setLayout(new GridLayout(3,1));
...
JPanel labelPanel = new JPanel();
labelPanel.setLayout(new FlowLayout());

...

basePanel.add(allComponentsPanel);
...
tabs.addTab("Look & Feel", null,
        basePanel,
        "Demonstrates the change of Swing ... Look & Feel");
```


DESY

Java – Swing: Layouts und LayoutManager, Screenshot



Java – Swing: Look & Feel

- Look & Feel bestimmt das Erscheinungsbild der Komponenten, also Vorder- und Hintergrundfarbe, Schriftfarbe und Schriftart und im weiteren Sinne auch die Form der Komponente (abgerundete Ecken, etc.)
- In Swing ist das Look & Feel einer Anwendung über den javax.swing.UIManager änderbar, das Standard Look & Feel in Java heißt Metal, darüber hinaus ist in Java das alte Unix Look & Feel Motif verfügbar. Zusätzlich kann auf das Look & Feel des Betriebssystems auf dem die Anwendung läuft zugegriffen werden und auch, wenn die Bibliotheken installiert sind auf das GTK Look & Feel
- Prinzipiell ist natürlich auch möglich ein eigenes Look & Feel zu entwickeln und seiner Anwendung mitzugeben

Java – Swing: Look & Feel, Beispiele

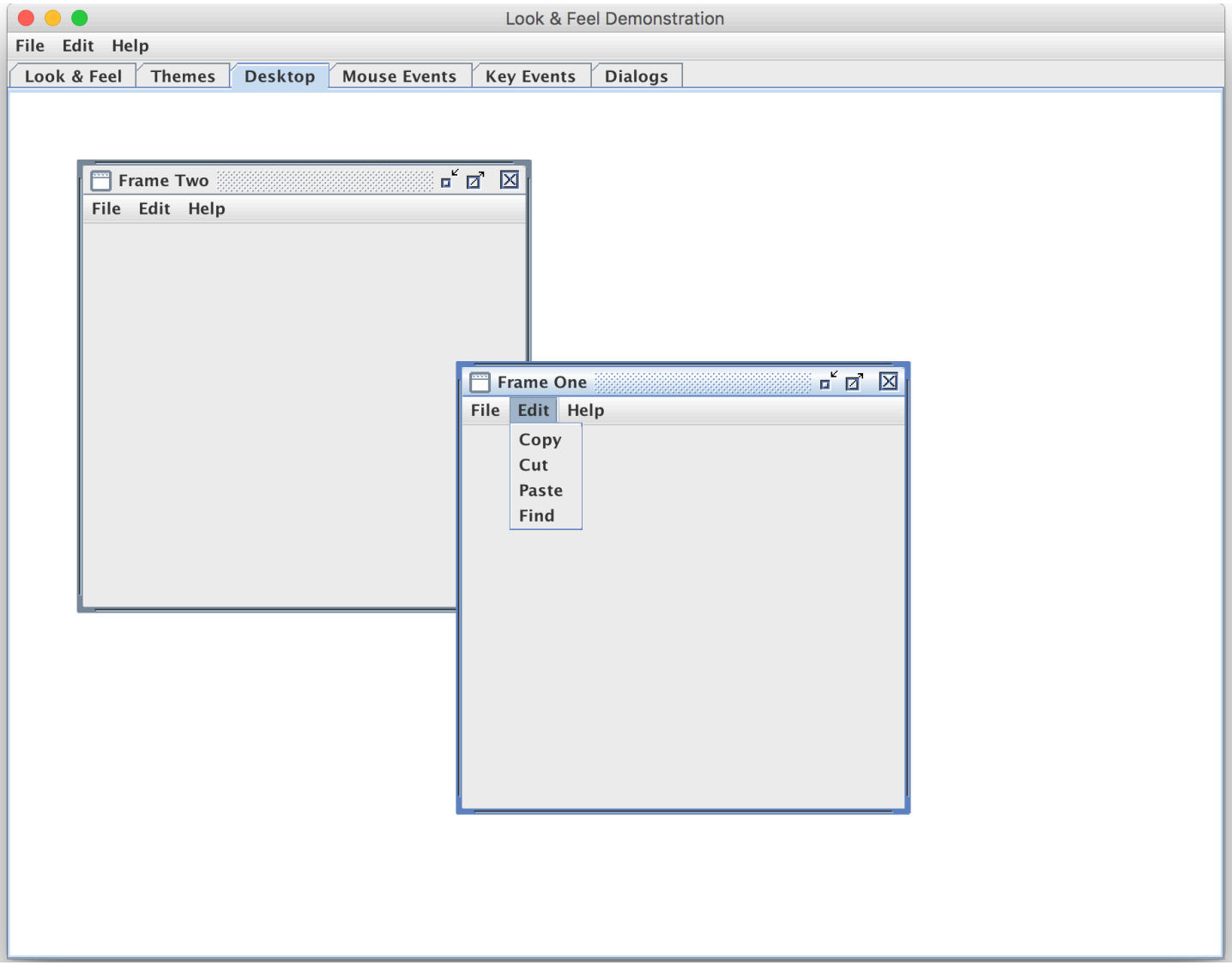
Java – Swing: Farben und Schriften

- java.awt.Color
- java.awt.Font

Java – Swing: DektopPane & JInternalFrame

- JInternalFrames ahmen das Verhalten von JFrame in einer speziellen Umgebung, dem JDesktopPane nach, sie sind jedoch keine TopLevel-Container
- JInternalFrames, können minimiert, maximiert und per Drag / Drop auf dem JDesktopPane verschoben werden, sie können den Focus bekommen und eine Menüleiste. Ebenso können sie alle Komponenten enthalten, die ein JFrame enthält
- Positionierung mit setLocation() ist relativ zum JDesktopPane

Java – Swing: DektopPane & JInternalFrame, Screenshot



Java – Swing: DektopPane & JInternalFrame, Beispiel

```
// a JDesktopPane with JInternalFrames in a separate tab
JDesktopPane desktop = new JDesktopPane();
// constructor params: title, resizable, closeable, maximizable,
iconifiable
JInternalFrame frameOne =
      new JInternalFrame("Frame One", true, true, true, true);
JInternalFrame frameTwo =
      new JInternalFrame("Frame Two", true, true, true, true);

frameOne.setSize(340,340);
frameOne.setLocation(25,25);
frameOne.setJMenuBar(this.createJMenuBar());
frameOne.setVisible(true);

frameTwo.setSize(340,340);
frameTwo.setLocation(50,50);
frameTwo.setJMenuBar(this.createJMenuBar());
frameTwo.setVisible(true);
//this.setContentPane(desktop);
desktop.add(frameOne);
desktop.add(frameTwo);
...
tabs.addTab("Desktop", null, desktop,
     "Some kind of virtual desktop with internal frames");
this.add(tabs);
```


DESY

Java – Swing: JPanel, Fläche zum Strukturieren

- JPanel Objekte haben keine Funktion außer der, Elemente aufzunehmen und strukturiert anzuordnen
- JPanel sind sehr hilfreich, wenn:
  - Gruppen von Bedienelementen angelegt werden sollen
  - Wenn Bereiche von Elementen durch anderes Layout, Farbe, Schrift oder Borders besonders hervorgehoben oder in den Hintergrund gestellt werden sollen
  - Um für alle Elemente auf diesem JPanel einen Tooltip anzuzeigen
  - Um nur bestimmte Bereiche für einen Mouse- oder KeyListener empfänglich zu machen

Java – Swing: JPanel, Beispiele

Java – Swing: JScrollBar und JScrollPane

- JScrollBar
  - Implementierung einer Scroll-Leiste
  - Festlegung von Richtung und wann die Leiste sichtbar sein soll
  - Kann einer Komponente zugeordnet werden, z.B. einem JPanel oder einer JTextArea
- JScrollPane
  - Ist im wesentlichen ein Panel mit zwei Scroll-Leisten (vertikal und horizontal)
  - Gebrauchsfertig in der Voreinstellung
  - Mit add wird dem ScrollPane beispielsweise eine JTextArea zugeordnet für die die Scroll-Leisten dann sichtbar werden, sobald der Inhalt des Textbereichs größer wird als der anzuzeigende Bereich.
  - Sehr umfangreiche Konfiguration möglich

Java – Swing: JTabbedPane

Java – Swing: Borders und BorderFactory

Java – Swing: Borders und BorderFactory, Beispiele

Java – Swing: JButton

Java – Swing: ButtonGroup & JRadioButton

Java – Swing: JTextField

Java – Swing: JTextArea

Java – Swing: JLabel

Java – Swing: Dialoge mit JOptionPane

Java – Swing: Dialoge mit JDialog

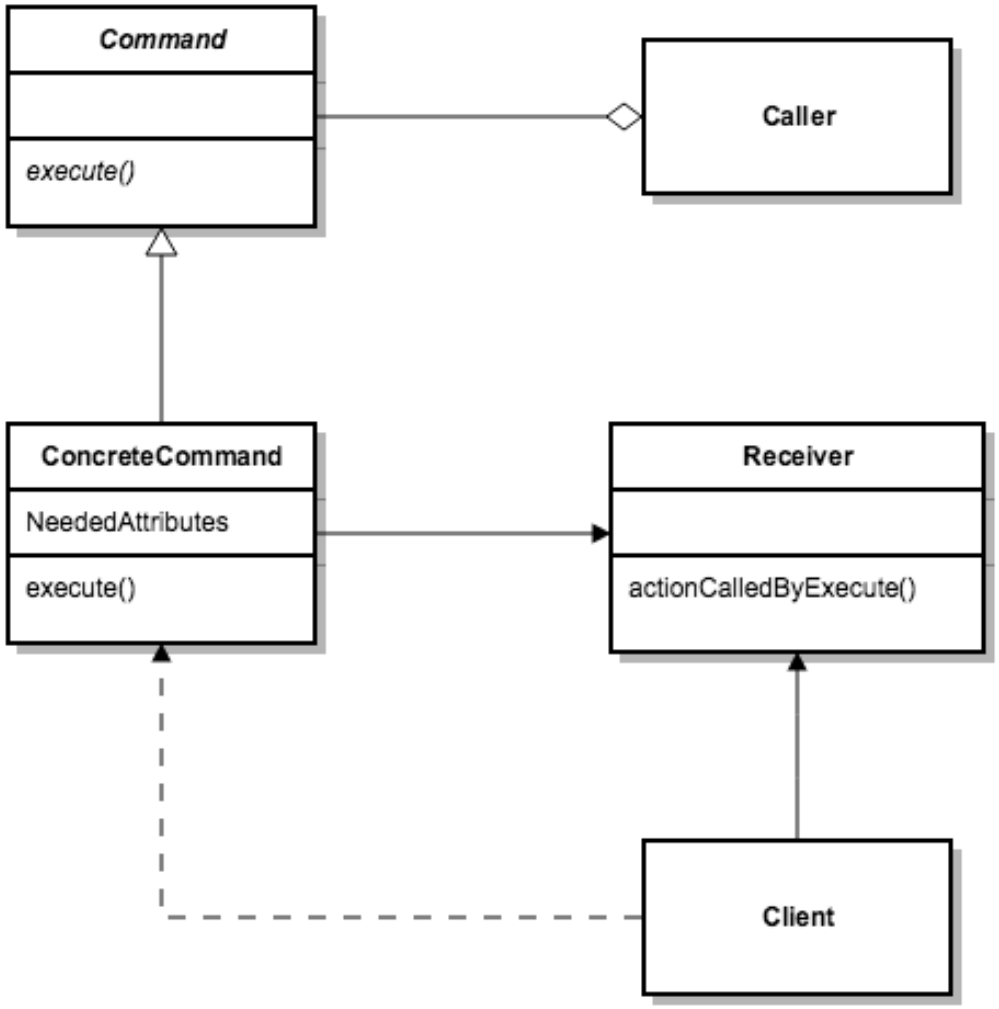
Java – Swing: JFileChooser

Java – Swing: JColorChooser

Java – Einschub: Design Pattern

Java – Einschub: Design Pattern, Singleton

Java – Einschub: Design Pattern, Command



Java – Aufgabe: Ein Texteditor