

Software Engineering

Teil 1
Crashkurs Java und Android

Wintersemester 2023/24

ÜBERBLICK

1.1	Java-Basics
1.2	Vererbung
1.3	Schnittstellen
1.4	Container
1.5	Exceptions
1.6	XML
1.7	Android

1.1 JAVA BASICS

- Die Programmiersprache Java wurde 1995 von Sun Microsystems entwickelt. 2010 wurde Sun und damit Java von Oracle übernommen
- Versionen:
 - JDK 1.0-1.4 (1996-2002)
 - J2SE 5.0 (2004)
 - Java SE 6 (2006)
 - Java SE 7 (2011)
 - Java SE 8 – LTS (März 2014)
 - Java SE 11 – LTS (September 2018)
 - Java SE 17 - LTS (September 2021)
 - Java SE 20 (März 2023)
 - Java SE 21 – LTS (September 2023)

ÜBERBLICK

- Java ist eine rein objektorientierte Programmiersprache und nicht hybrid wie C++
 - Die Syntax ist sehr ähnlich zu C und C++
 - for, while, if-else, switch, Variablendeklaration,... identisch
 - Die Sprachmittel sind im Vergleich zu C++ eingeschränkt:
 - Keine „echten“ Zeiger (d.h. keine Zeigerumwandlungen und Zeigerarithmetik)
 - Keine Speicherverwaltung durch den Programmierer
 - Keine Operator-Überladung
 - Keine Mehrfachvererbung
 - Java besitzt eine automatische Speicherverwaltung (garbage collection), daher keinen delete-Operator
- ➔ Java ist „einfacher“ als C++
- Java-Code wird in plattformunabhängigen Bytecode übersetzt, der von der Java Virtual Machine zur Laufzeit interpretiert wird

GRUNDLEGENDE KONTROLLFLUSS-SYNTAX

- Im Wesentlichen identisch mit C/C++.

if/else

```
if(x==4) {  
    // act1  
} else {  
    // act2  
}
```

do/while

```
int i=5;  
do {  
    // act1  
    i--;  
} while(i!=0);
```

for

```
int j;  
for(int i=0;i<=9;i++){  
    j+=i;  
}
```

switch

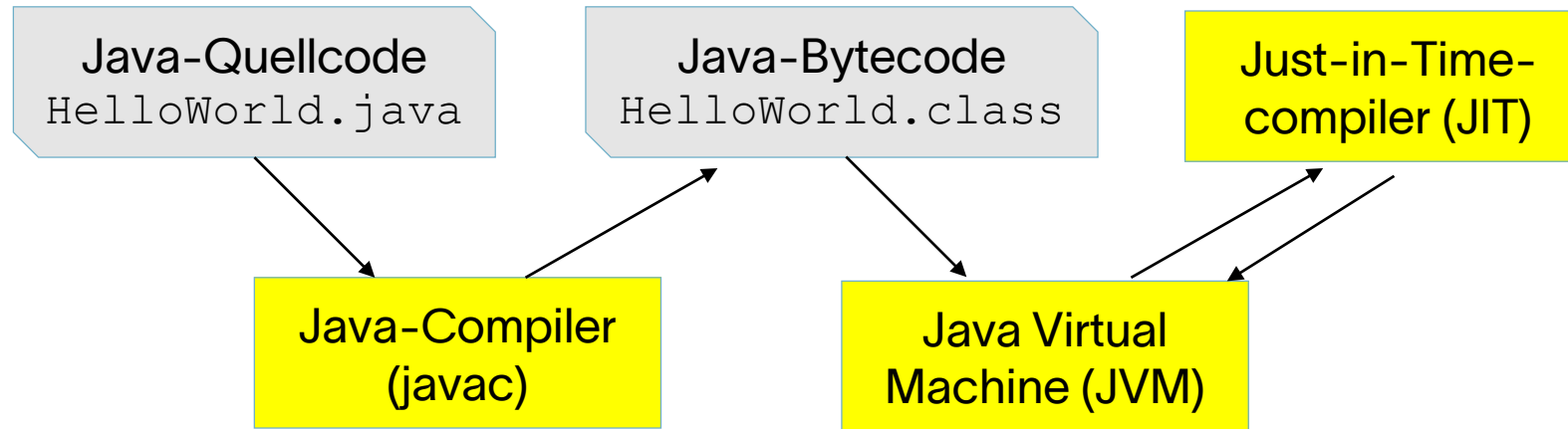
```
char c;  
c= ... ;  
switch(c) {  
    case 'a':  
    case 'b':  
        // act1  
        break;  
    default:  
        // act2  
}
```

HELLO WORLD

```
public class HelloWorld {  
    public static void main(String[ ] args) {  
        System.out.println("Hello World!");  
    }  
}
```

- Java kennt nur Klassen – rein objektorientierte Sprache
- Die **main**-Methode muss also immer in eine Klasse eingebettet sein
- Die Datei muss so wie die (öffentliche) Klasse heißen, also hier **HelloWorld.java**
- Die Methode **println()** des Standardausgabe-Streams **System.out** dient zur Ausgabe auf der Konsole

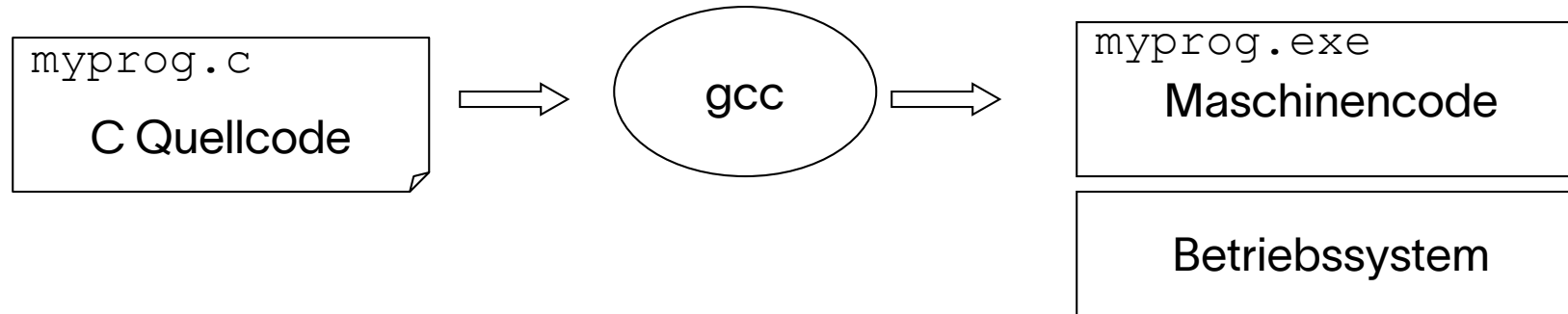
BYTECODE UND JAVA VIRTUAL MACHINE



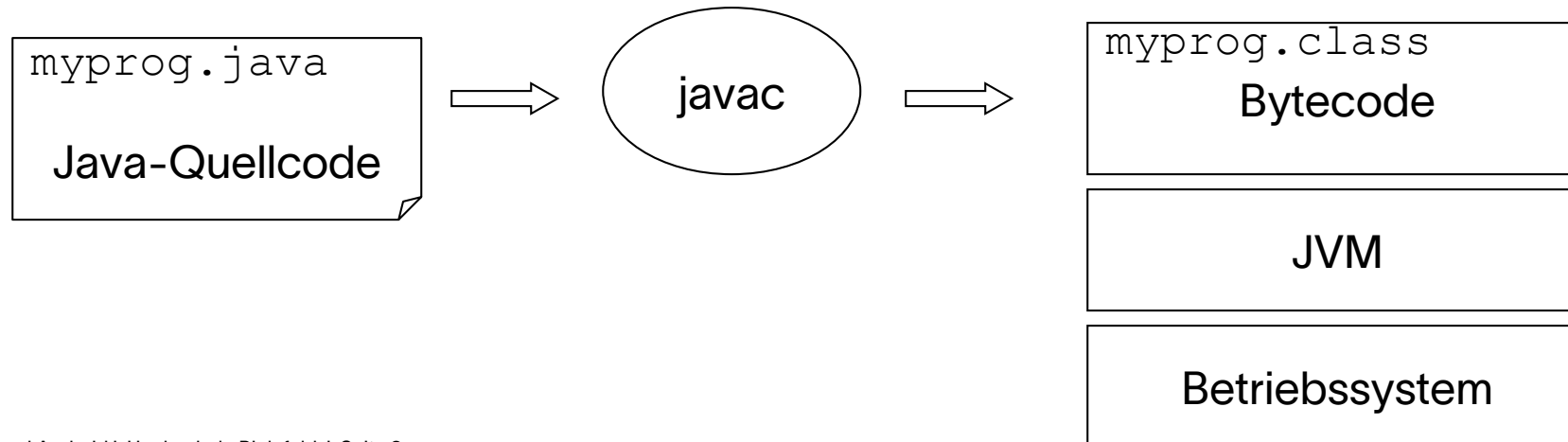
- Der Java-Sourcecode wird vom Compiler in maschinenunabhängigen Java-Bytecode übersetzt
- Der Java-Bytecode wird von der virtuellen Maschine (JVM) interpretiert
- Eventuell übersetzt der Just-In-Time-Compiler (JIT) der JVM einzelne Methoden vor ihrer Ausführung in Maschinencode (zur Performanceverbesserung)

VERGLEICH

plattformabhängig



plattformunabhängig



ELEMENTARE DATENTYPEN

Name	Art	Speicherbedarf
byte	vorzeichenbehaftete ganze Zahl	1 Byte
short	vorzeichenbehaftete ganze Zahl	2 Bytes
int	vorzeichenbehaftete ganze Zahl	4 Bytes
long	vorzeichenbehaftete ganze Zahl	8 Bytes
float	Fließkommazahl	4 Bytes
double	Fließkommazahl	8 Bytes
boolean	boolescher Wert	1 Byte
char	Unicode-Zeichen	2 Bytes

- Die Größe der Datentypen ist genau spezifiziert und nicht plattformabhängig wie in C/C++
- Der logische Datentyp heißt **boolean** (statt **bool**) und kann nicht in andere Typen konvertiert werden

STRINGS IN JAVA

- **String** ist eine Standard-Java-Klasse
- Mit dem Operator + kann man Strings konkatenieren
- Zahlen werden in diesem Kontext automatisch umgewandelt
- Strings sind unveränderbar (immutable)
- Alle Java-Klassen haben eine **toString()** -Methode, entweder selbst implementiert oder geerbt

```
String str = "abc";  
System.out.println("abc");  
String cde = "cde";  
System.out.println("abc" + cde);  
String c = "abc".substring(2,3);  
String d = cde.substring(1, 2);
```

KLASSEN UND OBJEKTE

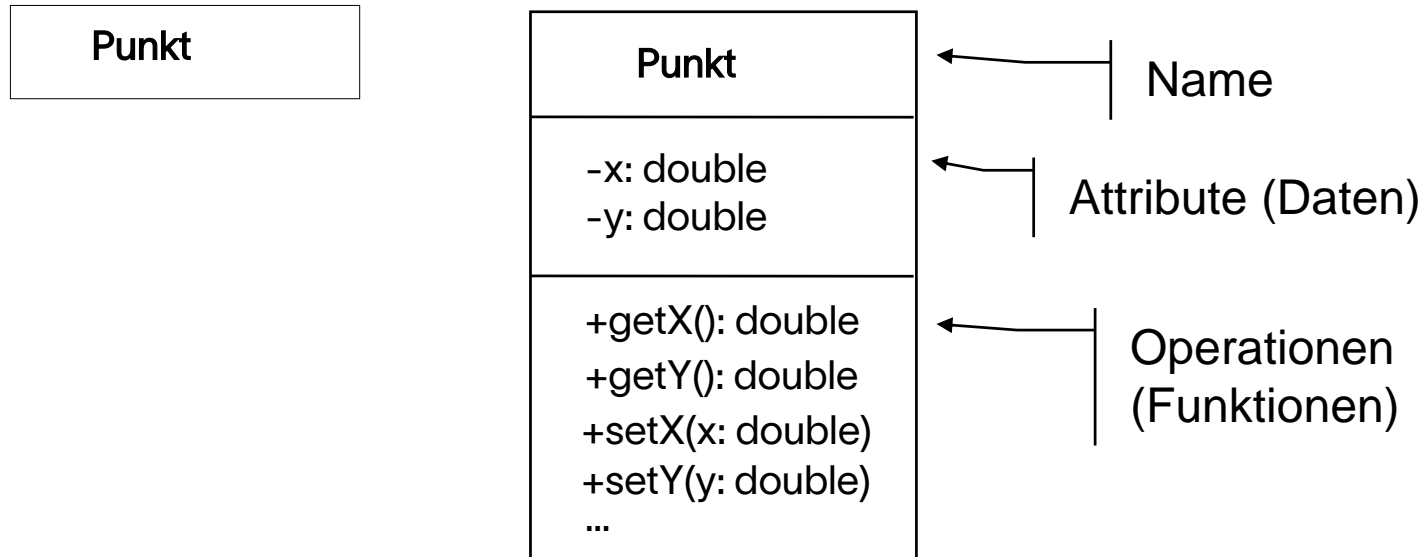
```
public class Punkt{  
    private double x;  
    private double y;  
  
    public Punkt(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public double getX() {  
        return x;  
    }  
  
    public void setX(double x) {  
        this.x = x;  
    }  
    ...  
}
```

- Alles, was zu einer Klasse gehört, muss in der Klasse stehen
- Zu jeder Klasse gibt es Konstruktoren, die mit dem Operator **new** aufgerufen werden.

AUFRUF VON METHODEN

```
class Test{  
    public static void main(String[] args) {  
  
        Punkt p = new Punkt(10,20);  
        System.out.println("x-Wert: "+p.getX());  
    }  
}
```

NOTATION FÜR KLASSEN IN UML



STATISCHE METHODEN

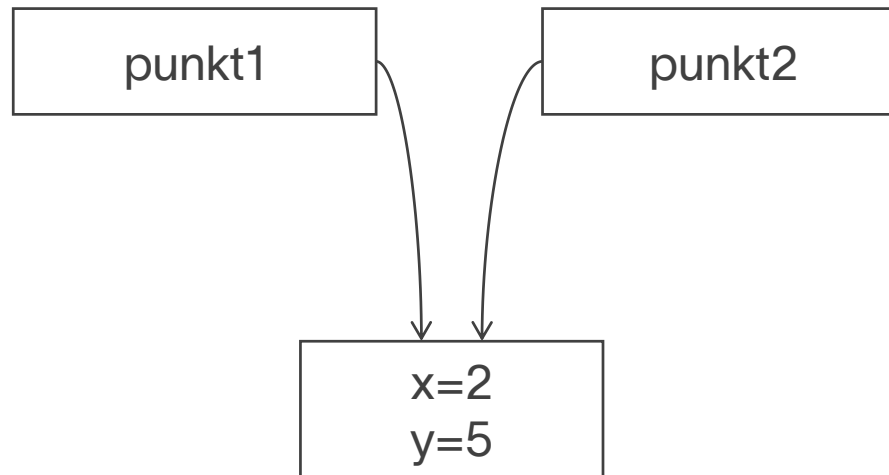
- Statische Methoden können ohne eine Instanz aufgerufen werden

```
public class MWSt{  
    public static double berechne(double x){  
        return x*0.19;  
    }  
}
```

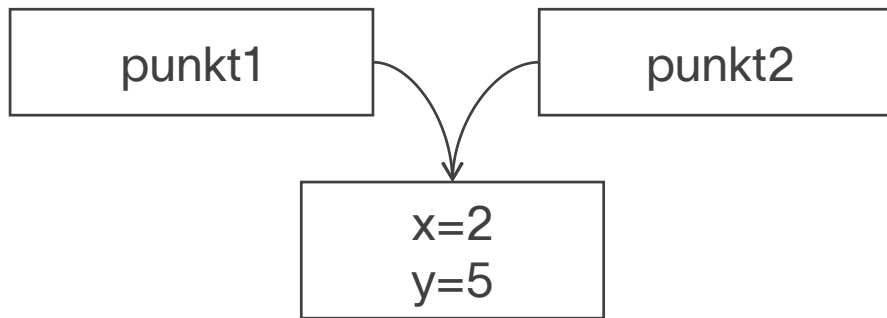
- Aufruf mit `m=MWSt.berechne(100) ;`
- Viele Bibliotheksfunktionen sind statisch:
 - `static double Math.sin(double a)`
- Statische Methoden können nicht auf Instanzvariablen zugreifen

VARIABLEN AUF OBJEKTE SIND REFERENZEN

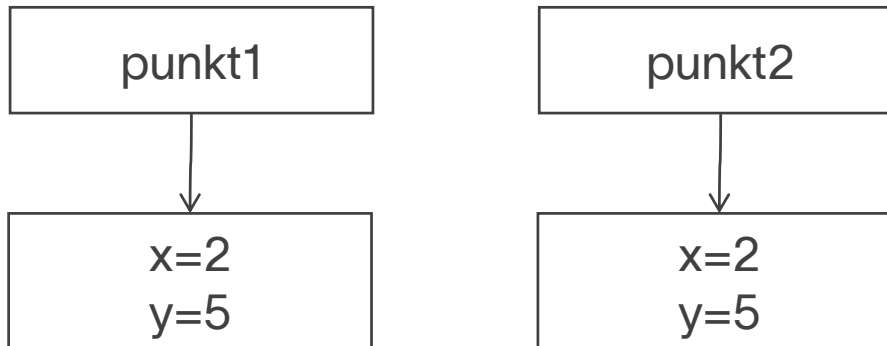
```
Punkt punkt1 = new Punkt(2,5);  
Punkt punkt2 = punkt1;  
punkt2.setX(3);  
System.out.println("x-Wert: "+punkt1.getX());
```



OBJEKTIDENTITÄT VERSUS OBJEKTGLEICHHEIT



`punkt1==punkt2`



`punkt1.equals(punkt2)`
(falls geeignet implementiert)

VARIABLEN MIT ELEMENTAREN DATENTYPEN SPEICHERN WERTE

```
int i=3;  
int j=i;
```

```
j=5;  
System.out.println(i);
```

i=3

/

j=3 5

METHODENAUFRUFE

- Hat eine Methode Objekte als Parameter, so werden die Werte der Objektreferenzen übergeben
- Die als Parameter übergebenen Objekte können also verändert werden.
- Die Referenzen selbst können nicht verändert werden, da nur ihr Wert übergeben wurde. Sie zeigen nach Ausführung der Methode immer noch auf dieselben Objekte mit der derselben Objekt-Id (auch wenn sich die Eigenschaften der Objekte geändert haben können).
- Daher hat Java bei elementaren Datentypen call-by-value und bei Objekten call-by-reference-value.
- Echtes call-by-reference würde bedeuten, dass man in den Methoden auch die Referenzen auf andere Objekte „umbiegen“ kann.

ARRAYS

- Java-Arrays sind **Objekte** und müssen mit **new** instanziiert werden

```
double[] values = new double[10];  
System.out.println( values.length ); // 10
```

- Array-Objekte haben ein Attribut **length**, über das man die Länge auslesen kann

PACKAGES

- Java-Code hat eine hierarchische Struktur
- Jede Klasse gehört zu einem Paket
- Sollen Klassen aus einem anderen Paket verwendet werden, muss ein **import**-Befehl verwendet werden.
- Alternativ kann im Quelltext auch der voll qualifizierte Klassenname verwendet werden.

```
package de.fhbi.swe.test

public class HelloWorld {
    public static void main(String[ ] args) {
        System.out.println("Hello World!");
    }
}
```

```
import java.util.Timer;
import java.math.*;
import de.fhbi.swe.util.*;
```

SICHTBARKEIT IN JAVA

- Vor Attributen und Methoden kann ein Sichtbarkeitsattribut stehen, das definiert, von wo aus auf sie zugegriffen werden darf
 - **private**: nur innerhalb der Klasse
 - **protected**: auch aus Unterklassen (vgl. 1.2)
 - **public**: von überall her
 - default (wenn nichts angegeben ist): nur innerhalb des selben package
- Für Klassen sind nur die Sichtbarkeitsattribute default und **public** erlaubt (Ausnahme: innere Klassen).
- Das Standard-Vorgehen in der Objektorientierung orientiert sich am Konzept des abstrakten Datentyps:
 - Attribute sind **private** oder **protected**
 - Methoden sind **public** (außer internen Hilfsmethoden)

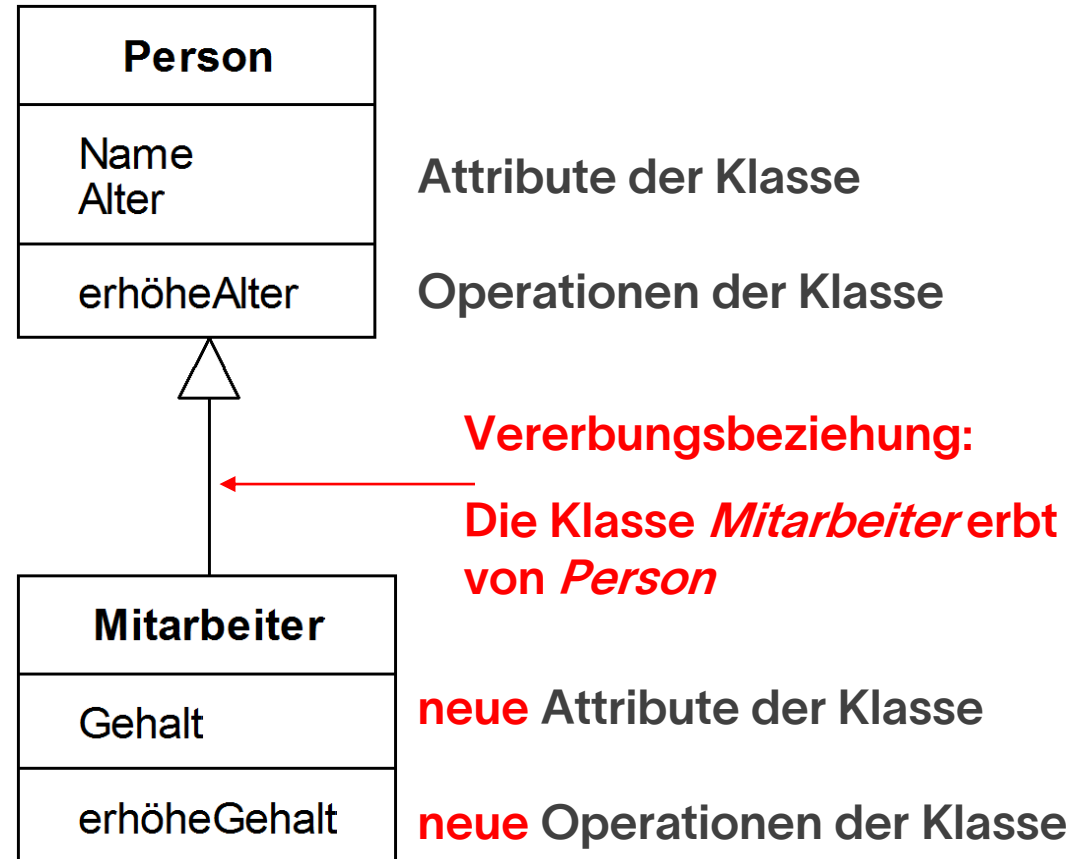
1.2 VERERBUNG

- Vererbung ("inheritance") ist ein Mechanismus, um neue Klassen mit Hilfe bereits bestehender Klassen zu definieren
- Eine Unterklasse ("subclass") erbt von der Oberklasse ("superclass")
 - die Implementierungen aller nicht-privaten Operationen
 - die nicht-privaten Attributdeklarationen (Instanzvariablen)
- Eine Unterklasse kann
 - geerbte Attribute und Operationen redefinieren
 - neue Attribute und Operationen definieren
- Durch Vererbung entstehen Klassenhierarchien
- Vererbung ist eine transitive Beziehung

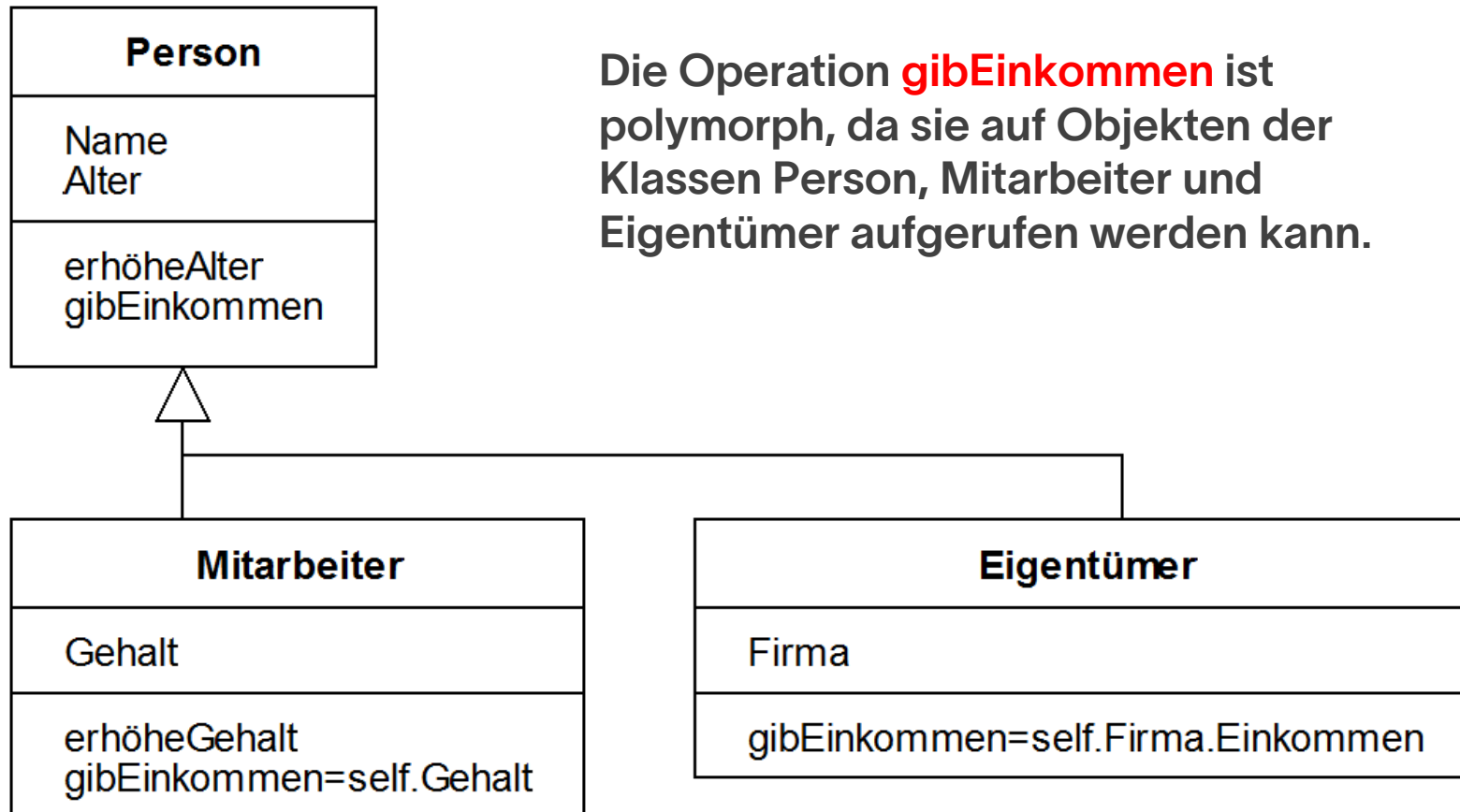
VERERBUNGSBEISPIEL

Die geerbten Operationen können durch eigene Operationen überschrieben werden (dynamische Polymorphie durch spätes Binden)

Achtung: die geerbten Attribute und Operationen werden in aller Regel nicht erneut aufgeführt.



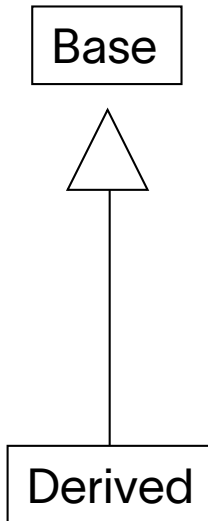
BEISPIEL FÜR POLYMORPHIE



DYNAMISCHES BINDEN

- Bei objektorientierten Sprachen kann ein Operationsaufruf in jeder erbenden Unterklasse durch eine andere Methode implementiert sein
- Die klassenabhängige Auswahl der Implementierung eines Operationsaufrufs zur Programmlaufzeit nennt man **dynamisches Binden**
- Im Gegensatz dazu wird die Auswahl einer bestimmten Prozedur aus einer Menge gleichbenannter Prozeduren zur Übersetzungszeit *Overloading* genannt (anhand der Parametertypen), auch bekannt als statische Polymorphie
 - **Beispiel:** der Operator "+" kann auf integer, float, ... realisiert sein. Der Übersetzer entscheidet anhand der Parameter, welchen Maschinencode er für das "+" erzeugen muss.

VERERBUNG



Im Gegensatz zu C++
kann es in Java immer
nur eine direkte
Oberklasse geben!

```

class Base {
    protected int i;
    Base() {...}

    Base(int i) {...}

    protected void foo() {...}
}

class Derived extends Base {
    private double x;
    Derived() {...}

    Derived(int i, double x) {
        super(i);
        this.x = ...
    }

    @Override
    protected void foo() {
        super.foo();
        ...
    }
}
  
```

Zugriff auf den
Konstruktor der
Oberklasse mit dem
Schlüsselwort **super**

Zugriff auf gleichnamige
Methode der Oberklasse

VERERBUNG

- Alle Java-Klassen sind von der Oberklasse **Object** abgeleitet
- Es gibt nur einfache Vererbung (d.h. jede Klasse hat genau eine direkte Oberklasse)
 - Damit ist die Vererbungshierarchie ein Baum
- Methoden der Oberklasse dürfen von einer Methode mit der gleichen Signatur überschrieben werden (d.h. alle Methoden sind „virtuell“ im Sinne von C++)
 - Die Sichtbarkeit darf dabei erweitert, aber nicht eingeschränkt werden

BEISPIEL

```
class Base {  
    public void foo() {  
        System.out.println("Base");  
    }  
}  
  
class Derived extends Base {  
    @Override  
    public void foo() {  
        System.out.println("Derived");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Base b = new Derived();  
        b.foo(); // Derived.foo() wird ausgeführt  
    }  
}
```

1.3 INTERFACES (SCHNITTSTELLEN)

- Interfaces ähneln abstrakten Klassen
- Sie stellen die Methodensignaturen bereit, aber keinerlei Implementierung (Ausnahmen seit Java 8)
- Es gibt keine Attribute.
- Konstanten sind möglich:
 - `public static final double PI = 3.14159;`
- Interfaces können **nicht** instanziiert werden. Dennoch kann eine Variable als Typ ein Interface haben.
- Eine Klasse implementiert ein Interface, wenn sie für alle Methoden eine Implementierung bereitstellt. Dies wird durch das Schlüsselwort *implements* angezeigt.
- Im Gegensatz zur Vererbung kann eine Klasse mehrere Interfaces implementieren
- In der Regel ist das Interface *public*, sonst ist es nicht sinnvoll nutzbar.

INTERFACES

```
public interface Buyable{  
    public double price();  
}
```

```
public interface Edible{  
    public int calories();  
}
```

```
public class Chocolate implements Buyable, Edible{  
  
    @Override  
    public double price(){  
        return 0.99;  
    }  
  
    @Override  
    public int calories(){  
        return 500;  
    }  
}
```

INTERFACES

- Mit Interfaces kann vor der Implementierung die Schnittstelle festgelegt werden.
 - ➔ Trennung von Design und Implementierung
- Der Compiler prüft, ob die Klassen tatsächlich die vorgegebenen Signaturen einhalten.
- Interfaces in Java sind ein teilweiser Ersatz für die Mehrfachvererbung in C++
- Hinweis: Seit Java 8 sind in Interfaces auch default-Methoden und statische Methoden erlaubt: <http://docs.oracle.com/javase/tutorial/java/land/defaultmethods.html>

1.4 CONTAINER

- Container sind Datenstrukturen, die Objekte aufnehmen und zurückgeben können
- Container können mit beliebigen Objekten arbeiten („raw type“)
- Besser: typsichere Container

```
List<String> l = new ArrayList<String>();  
  
l.add("Hallo");  
l.add("Harry");  
  
for (String s:l) {  
    System.out.println(s);  
}
```

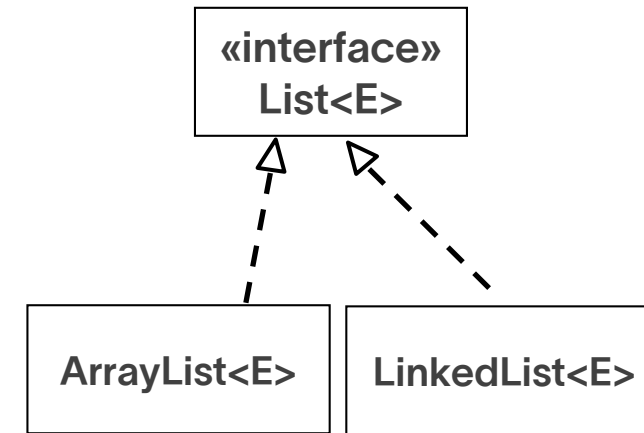

CONTAINER

- Es gibt verschiedene Typen von Containern:
 - Collection: enthält Elemente
 - Set: jedes Element nur einmal
 - List: mehrfach möglich
 - Queue: Warteschlange
 - Map: Key-Value-Paare
- Hierbei handelt es sich um Interfaces, die von den konkreten Klassen implementiert werden

WICHTIGSTER CONTAINERTYP FÜR UNS: LIST<E>

- Basisoperationen
 - `int size();`
 - `boolean isEmpty();`
 - `boolean contains(Object element);`
 - `boolean add(E element);`
 - `boolean remove(Object element);`
 - ...
`List<String> list1`
`= new ArrayList<String>();`

`List<String> list2`
`= new LinkedList<String>();`



Schleifen-Syntax

```
for (Object o : collection){
    System.out.println(o);
}
```

1.5 EXCEPTIONS (AUSNAHMEN)

- Exceptions sind Objekte, die im Fehlerfall in einem Programm erzeugt werden
- Man sagt, dass eine Exception „geworfen“ wird, wenn ein bestimmter Fehlerfall eintritt
- In diesem Fall wird der normale Kontrollfluss unterbrochen und an die Stelle gesprungen, wo die Exception „gefangen“ wird
- Mit **throw** wird eine neue Exception ausgelöst
 - **throw new IllegalArgumentException("Keine negative Zahl erlaubt");**
- Es gibt unzählige Typen von Exceptions (in der Java-API und selbst definiert): **IOException, PrinterException, ClassNotFoundException, InterruptedException, ParseException, ...** Alle sind Subklassen von **Exception**.

ALLGEMEINE SYNTAX

```
try{  
    // gefährliche Operationen  
}  
catch(<Exception class1> e1){  
    // Fehlerbehandlung  
}  
catch(<Exception class2> e2){  
    // Fehlerbehandlung  
}  
finally{  
    // wird immer ausgeführt  
}
```

BEISPIEL: ZAHLKONVERTIERUNG

```
String stringToConvert = "19%";

try{
    Integer.parseInt( stringToConvert );
}
catch ( NumberFormatException e ){
    System.out.println("Konvertierung nicht möglich!");
}
System.out.println( "Weiter geht's" );
```

EXCEPTIONS

Die Verwendung von Exceptions bietet mehrere Vorteile:

1. Fehlercode und regulärer Code werden getrennt
2. Exceptions können die Aufrufhierarchie hochgereicht werden
3. Fehler können durch die Vererbungshierarchie der Exceptions gruppiert und typisiert werden.

BEISPIEL OHNE EXCEPTIONS (PSEUDOCODE)

```
errorCodeType readFile {  
    initialize errorCode = 0;  
    open the file;  
    if (theFileIsOpen) {  
        determine the length of the file;  
        if (gotTheFileLength) {  
            allocate that much memory;  
            if (gotEnoughMemory) {  
                read the file into memory;  
                if (readFailed) {  
                    errorCode = -1;  
                }  
            } else {  
                errorCode = -2;  
            }  
        } else {  
            errorCode = -3;  
        }  
        close the file;  
        if (theFileDidntClose && errorCode == 0) {  
            errorCode = -4;  
        } else {  
            errorCode = errorCode and -4;  
        }  
    } else {  
        errorCode = -5;  
    }  
    return errorCode;  
}
```

BEISPIEL MIT EXCEPTIONS (PSEUDOCODE)

```
readFile {  
    try {  
        open the file;  
        determine its size;  
        allocate that much memory;  
        read the file into memory;  
        close the file;  
    } catch (fileOpenFailed) {  
        doSomething;  
    } catch (sizeDeterminationFailed) {  
        doSomething;  
    } catch (memoryAllocationFailed) {  
        doSomething;  
    } catch (readFailed) {  
        doSomething;  
    } catch (fileCloseFailed) {  
        doSomething;  
    }  
}
```


BEISPIEL FÜR HOCHREICHEN IN DER AUFRUFHIERARCHIE

- Wenn eine aufgerufene Methode eine Exception deklariert, muss diese immer entweder gefangen und behandelt oder an den Aufrufer weitergereicht werden. Im letzteren Fall muss dies im Methodenkopf deklariert werden

```
method1 {  
    try {  
        call method2;  
    } catch (exception) {  
        doErrorProcessing;  
    }  
}  
method2 throws exception {  
    call method3;  
}  
method3 throws exception {  
    call readFile;  
}
```

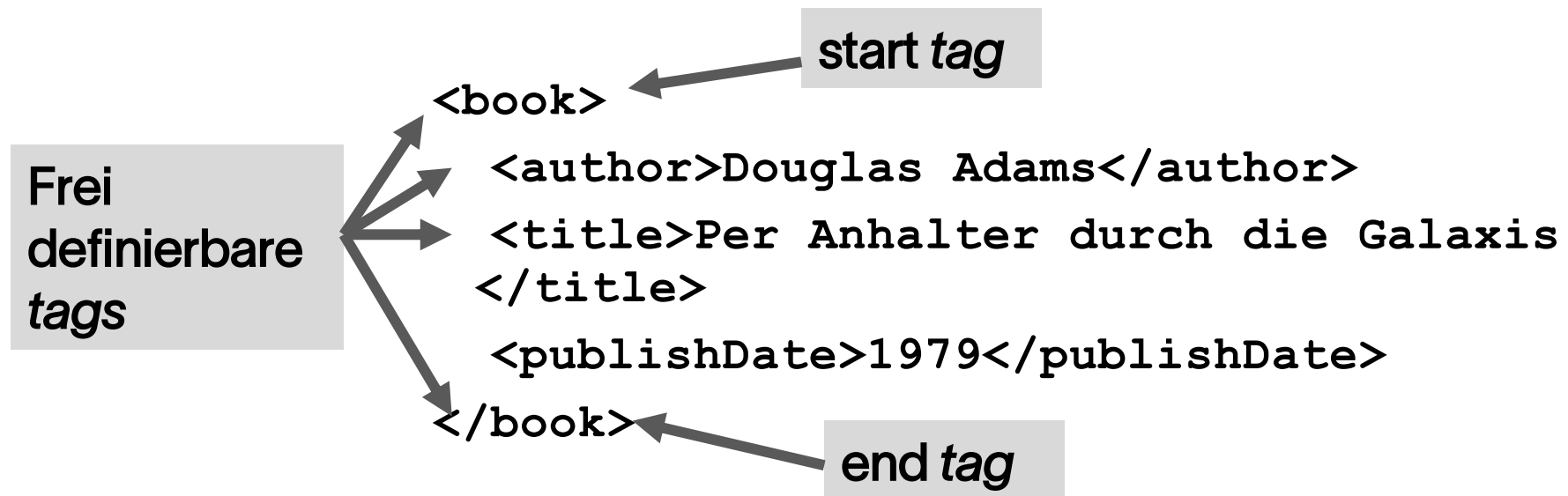
Behandeln der
Exception

Weiterreichen an
Aufrufer

1.6 XML

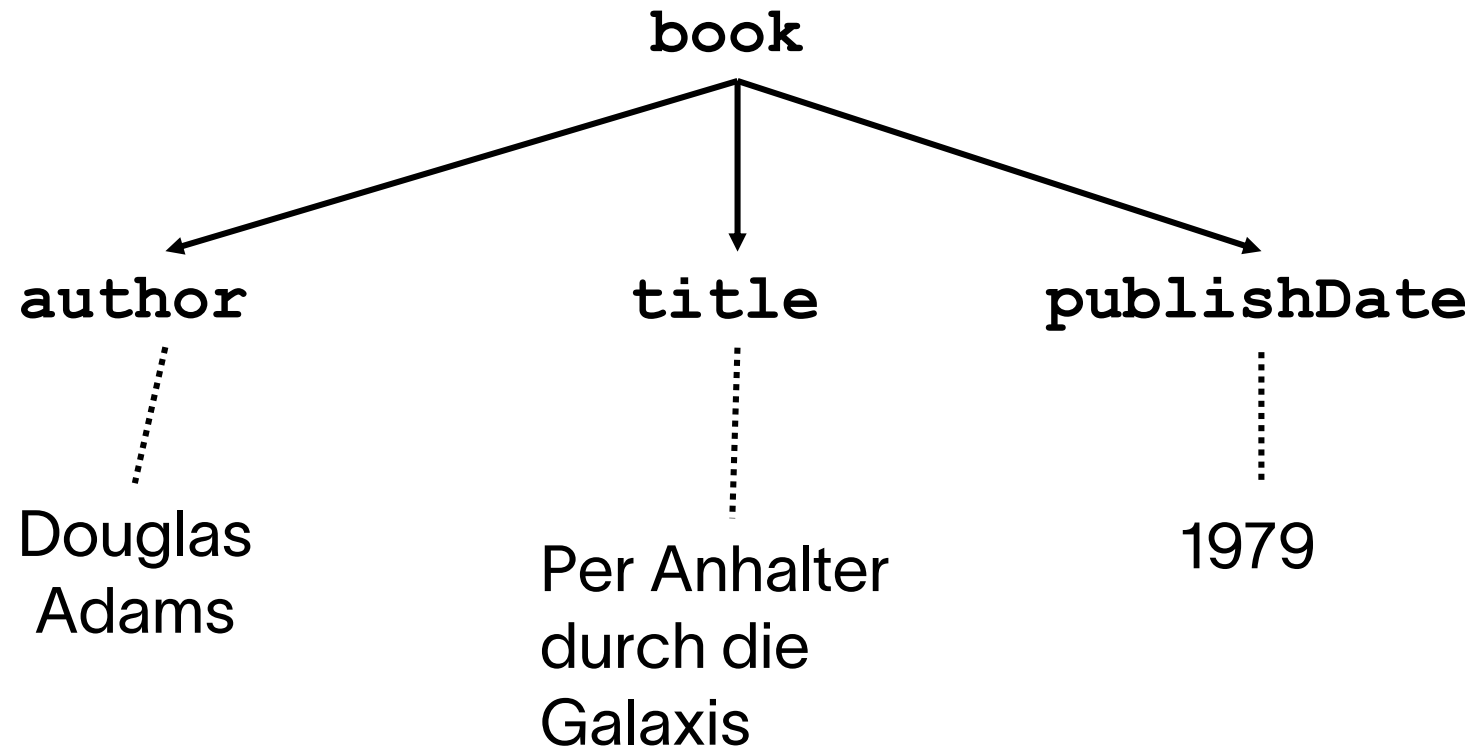
- Extensible Markup Language
- XML-Dokumente beschreiben strukturierte Informationen
- XML-Dokumente enthalten
 - Elemente (definiert durch tags)
 - Attribute
 - Text

BEISPIEL XML-DOKUMENT (MIT TAGS UND TEXT)



- Ein Element besteht aus allem zwischen *start tag* und *end tag*
- Als Inhalt sind Text und/oder Subelemente erlaubt

XML DOKUMENT ALS BAUM



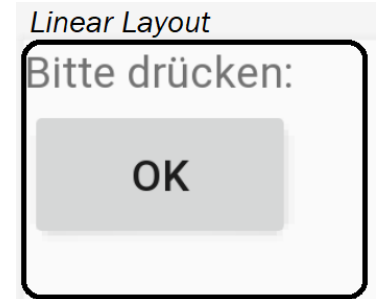
BEISPIEL: ANDROID-LAYOUT-FILE

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Bitte drücken:" />
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="OK" />
</LinearLayout>
```

Attribute

Leeres Element,
hat nur Attribute

Design-
Ansicht:



ÜBERBLICK: ELEMENTE IN XML

- Frei definierbare *tags*: `article`, `title`, `author`
 - mit *start tag*: `<article>` etc.
 - und *end tag*: `</article>` etc.
- Elemente: `<article> ... </article>`
- Elemente haben einen Namen (`article`) und Inhalt (`...`)
- Elemente können verschachtelt werden
- Elemente können leer sein.
Kurzschreibweise:
`<Button/>` ist äquivalent zu `<Button></Button>`
- Jedes XML-Dokument hat genau ein Root-Element und bildet einen Baum.

ELEMENTE VS. ATTRIBUTE

Elemente können Attribute mit Namen und Wert besitzen:

z.B. `<TextView android:text="Bitte drücken:" />`

Unterschiede:

- Pro Element ist nur ein Attribut mit demselben Namen zulässig (aber beliebig viele Subelemente)
- Attribute haben keine Unterstruktur. Elemente können Subelemente enthalten

NAMESPACES

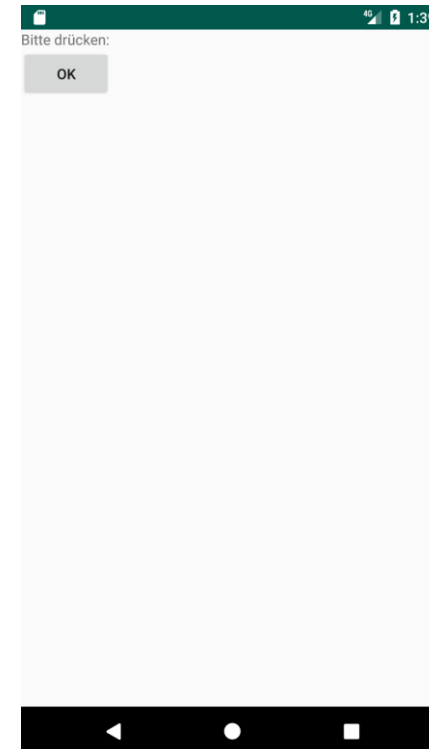
- Da die Attributnamen nicht über alle möglichen XML-Dokumente eindeutig sein können (z.B. Attribut name="..." oder id="..."), gibt es *Namespaces* zur Unterscheidung
- Im Android-XML steht
`xmlns:android="http://schemas.android.com/apk/res/android"`
- Dabei ist http://... ein eindeutiger Ressource-Identifizier (URI). In der Regel findet man unter dieser Adresse keine Internetseite mit Informationen. Grund für diese Konvention ist, dass bei jeder nur die Internet-Domain nutzen soll, die ihm gehört und damit die Eindeutigkeit sichergestellt ist
- Theoretisch hätte man auch
`xmlns:android="hajfhkjadh987134anii23fe.,m.,mzdf"` schreiben können, solange die ID eindeutig ist

1.7 ANDROID

- Android ist ein Betriebssystem für mobile Geräte
- Basis in ein Linux-Kernel
- Standardprogrammiersprachen für die Anwendungsentwicklung sind Kotlin und Java
- Im folgenden werden einige wichtige Grundlagen für die Programmierung einer App unter Android erläutert

ANDROID ACTIVITIES

- Eine Activity stellt eine Bildschirmseite in einer App dar
- Das Layout ist in einem XML-File hinterlegt.
Namenskonvention: Zur Klasse **BlaBlaActivity.java** gehört das XML-File **activity_bla_bla.xml**
- Layout-Elemente können auch in der Activity-Klasse dynamisch hinzugefügt werden (z.B. **layout.addView(button)**), aber dies nutzt man nur für Elemente, die zur Laufzeit hinzukommen. Alle anderen View-Elemente sollte man separat im Layout-XML-File definieren.



MINIMALGERÜST EINER ACTIVITY-KLASSE

```
package swe.fhbielefeld.de.beispielapp;
```

```
import android.app.Activity;
```

```
import android.os.Bundle;
```

```
public class MainActivity extends Activity {
```

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.activity_main);
```

```
}
```

```
}
```

Standardname der
Startactivity

Verpflichtende
onCreate()-Methode

Aufruf der onCreate()-
Methode der
Oberklasse

Verknüpfung mit XML-Layout-File

ACTIVITY LIFECYCLE

- Neben der **onCreate()** -Methode existiert noch eine Reihe weiterer Lifecycle-Methoden
- Zwischen **onStart()** und **onStop()** ist die Activity sichtbar
- Alle Methoden, die man nicht selbst implementiert, werden von der Oberklasse Activity geerbt
- Eigene Implementierungen müssen immer als erstes die super-Methode aufrufen

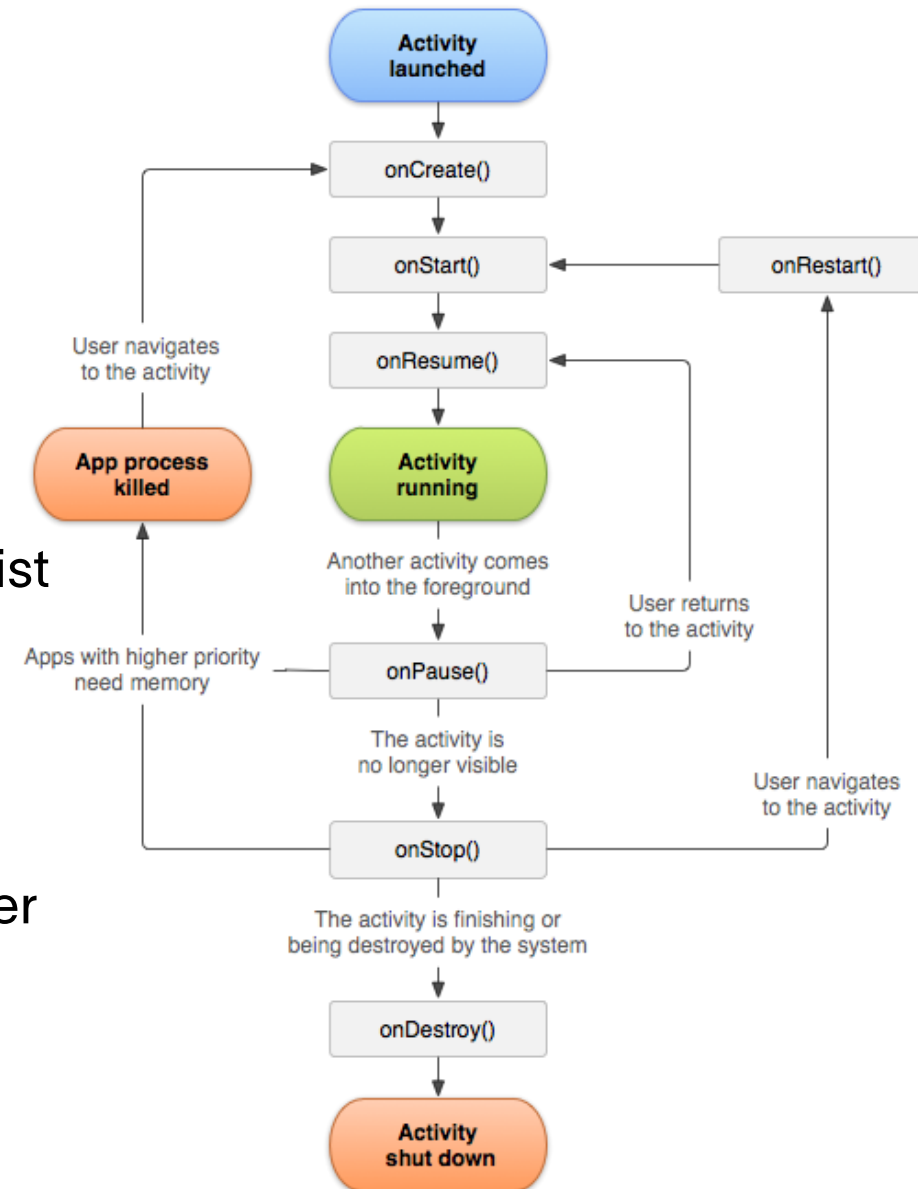


Bild: <https://developer.android.com/reference/android/app/Activity.html>
CC BY 2.5 Android Open Source Project

DIE FINDVIEWBYID()-METHODE

- Um auf die im XML-File definierten Views zuzugreifen, gibt es die Methode `findViewById()`
- Ist im XML-Dokument eine Id definiert, z.B. `android:id="@+id/button"`, so kann man auf das Element wie folgt zugreifen:


```
Button b = findViewById(R.id.button);  
b.setText("Neuer Text");
```

und dabei alle Methoden der Klasse Button nutzen
- Die Klasse `R.java` wird automatisch aus den gültigen XML-Files generiert (und von Android-Studio gut versteckt). Dort wird für jedes View-Element eine eindeutige hexadezimale ID hinterlegt

LISTENER

- Um auf Benutzerinteraktionen reagieren zu können, muss mit den gewünschten View-Elementen (z.B. Buttons) ein OnClickListener verknüpft werden:

```
Button b = findViewById(R.id.button);  
b.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        // ... Hier steht, was passieren soll  
    }  
});
```



Interface wird mit anonymer innerer Klasse implementiert

- Alternativ kann man auch im XML-File ein Attribut `android:onClick="tueDas"` hinzufügen und muss dann in der Activity eine Methode `public void tueDas(View view)` erstellen, in der implementiert ist, was passieren soll

INTENTS

- Für den Wechsel zwischen zwei Activities verwendet man die Klasse **Intent**
- Einem neuen Intent gibt man im Konstruktork den aktuellen Kontext `this` und die neu aufzurufende Activity-Klasse mit

- Außerdem kann man (falls nötig) Daten hinzufügen:

```
Intent intent = new Intent(this, SecondActivity.class);  
intent.putExtra("Telefonnummer", "617263871"); //optional  
startActivity(intent);
```

- Auf der Empfängerseite (in der SecondActivity) kann man die Daten innerhalb der `onCreate()`-Methode wieder auslesen:

```
String nummer = getIntent().getStringExtra("Telefonnummer");
```

LOGGING

- Für Logausgaben gibt es die Klasse `Log` mit den statischen Methoden `v()`, `d()`, `i()`, `w()` und `e()` für die Loglevel verbose, debug, info, warn und error.
- Diese Methoden haben zwei String-Parameter: *tag* und *message*
- Als Tag sollte man in seiner Klasse eine String-Konstante definieren:

```
private static final String TAG =  
MyActivity.class.getSimpleName();
```

- Dann kann man folgendermaßen loggen:

```
Log.d(TAG, "Schritt durchgeführt");  
Log.e(TAG, "Es ist ein Fehler aufgetreten");
```


- In Android Studio sieht man die Logs in der Logcat-Ausgabe (Alt-6 drücken). Dort Filterung nach Loglevel.

EXCEPTIONS IM LOGGING

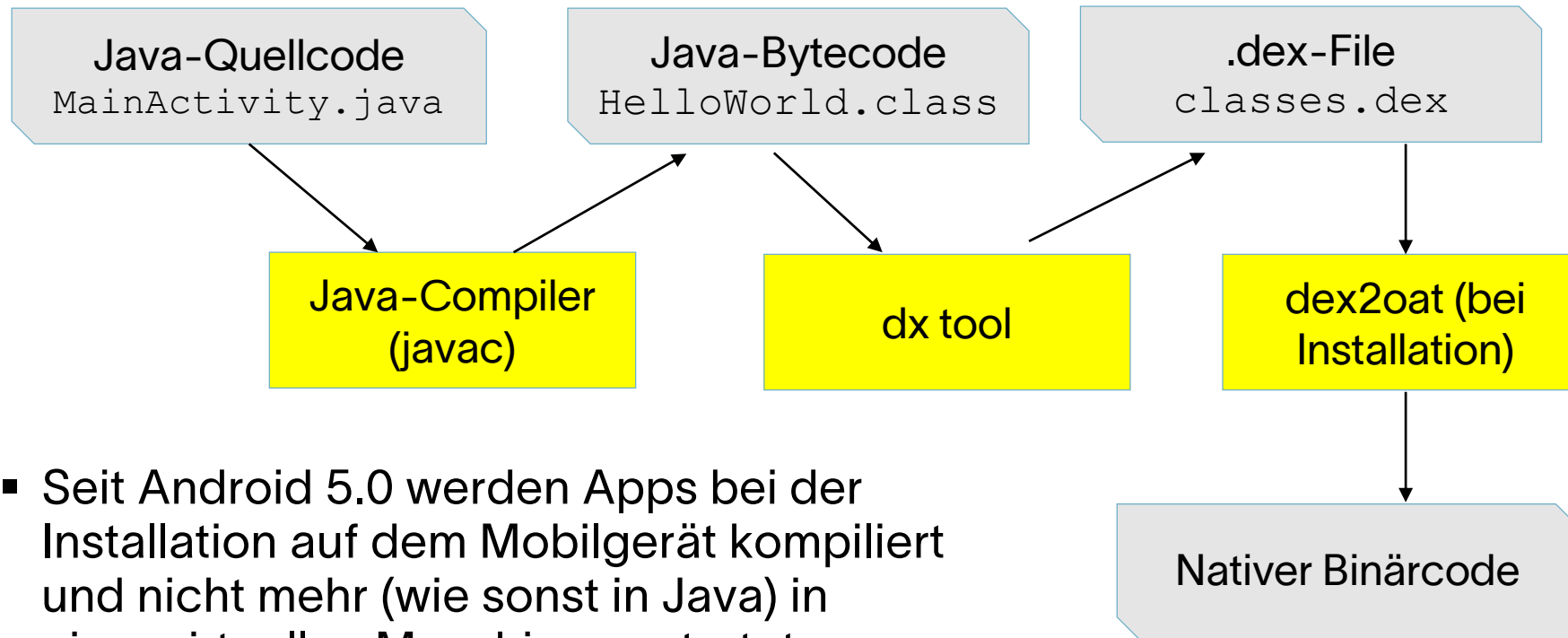
- Die Logging-Methoden gibt es noch in einer Variante mit drei Argumenten: *tag*, *message* und *throwable*
- Damit kann man Exceptions wie folgt verarbeiten:

```
try{  
    ...  
}  
catch (Throwable tr){  
    Log.e(TAG, "Es ist ein Fehler aufgetreten.", tr);  
}
```

Stacktrace wird in die
Logausgabe eingefügt.



ANDROID RUNTIME



- Seit Android 5.0 werden Apps bei der Installation auf dem Mobilgerät kompiliert und nicht mehr (wie sonst in Java) in einer virtuellen Maschine gestartet
- Die Darstellung ist verkürzt. Nicht aufgeführt ist das Ein- und Auspacken der dex-Files und Ressourcen als apk-File

REFERENZEN

- Java ist auch eine Insel:
<http://openbook.rheinwerk-verlag.de/javainsel/>
- Java SE 8 Standard-Bibliothek:
<http://openbook.rheinwerk-verlag.de/java8/>
- Java API Documentation: <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>
- Java Tutorial:
<https://docs.oracle.com/javase/tutorial/>
- Thomas Künneht: Android 8.
Rheinwerk Computing Verlag, 2018



Module	Package	Class	Use	Tools	Platform	API	Help
<p>Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification</p> <p>This document is divided into two sections:</p> <p>Java SE The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with <code>java</code>.</p> <p>JDK The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with <code>jdk</code>.</p>							
All Modules	Java SE	JDK	Other Modules				
Module	Description						
java.base	Defines the fundamental APIs of the Java SE Platform.						
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.						
java.datatransfer	Defines the API for transferring data between and within applications.						
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, locale, imaging, printing, and JavaFX.						
java.instrument	Defines services that allow agents to instrument programs running on the JVM.						
java.logging	Defines the Java Logging API.						