



TECHNISCHE
UNIVERSITÄT
WIEN
Vienna | Austria



Institut für
Computertechnik
Institute of
Computer Technology

Java Einstieg

Thomas Rathfux
Ralph Hoch

Überblick (1/5)

- Java – Ein erster Überblick
- Primitive Datentypen
- Main-Methode
- Ablaufsteuerung
- Schleifen
- Listen
- Fehlerbehandlung & Stacktrace



Überblick (2/5)

■ Klassen

- Der Konstruktor
- Objekte und Objekterzeugung
- Garbage Collection

■ Methoden

- Methodenaufruf
- Methodensignatur
- Parameterübergabe
- Statische Methoden
- Overloading



Überblick (3/5)

- Abstrakte Klassen
- Vererbung
 - Allgemeines
 - Sichtbarkeiten
- Konstruktoren und Vererbung
- Attribute und Vererbung
 - Sichtbarkeit
 - Überdeckung von Attributen
- Methoden und Vererbung
 - Overriding



Überblick (4/5)

- Interfaces
- Typen in Java
- Dynamisches Binden
- Casting / Typumwandlung
 - Type-Casting
 - Implizite Typumwandlung
 - Explizite Typumwandlung
 - Sichere Typumwandlung (Typesafe-Casting)



Überblick (5/5)

- Exceptions
 - Exceptions werfen
 - Fehlerbehandlung
- Enumerations
- Die *Object*-Klasse
 - *Object*-Methoden
 - *equals*-Methode
 - *toString*-Methode





Java – Ein erster Überblick

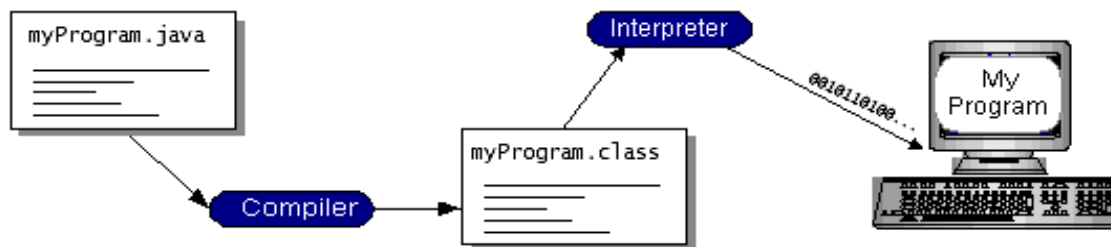
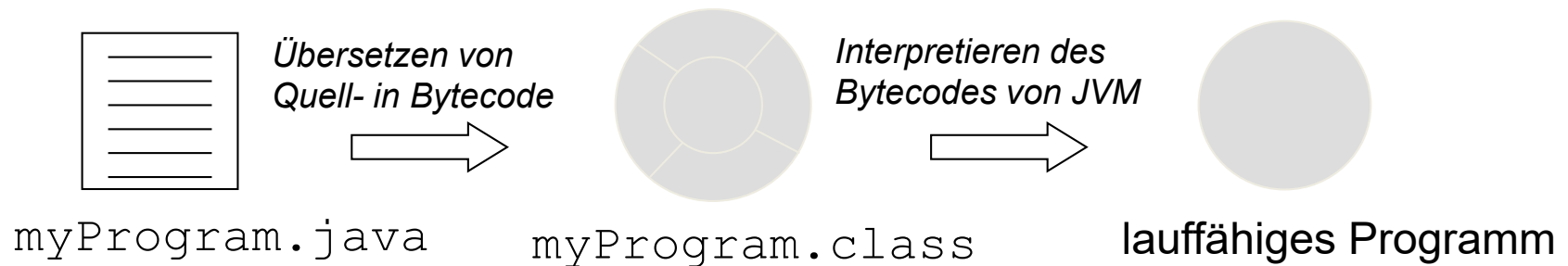
Java

- Statisch typisierte objektorientierte Programmiersprache
- Laufzeitumgebung ist eine virtuelle Maschine (JVM)
- Java-Laufzeitumgebungen führen Bytecode aus



Java Runtime-Environment

- Java-Code wird in einen **plattform-unabhängigen Bytecode** übersetzt.
- Bytecodes werden von einer **plattform-abhängigen Java Virtual Machine (JVM)** zur Laufzeit interpretiert.



Auszug von Java Schlüsselwörtern

Schlüsselwort	Verwendung
class	Deklariert eine Klasse.
new	Schlüsselwort für die Objekterzeugung.
final	Verhindern von Variablenüberdeckung oder Ableitung von Klassen.
this	Zugriff auf eigene Objektinstanz.
super	Zugriff auf Eigenschaften der Superklasse.
static	Klassenvariable, statische Methode.
extends	Vererbung zwischen Klassen oder Interface.
implements	Implementierung eines Interface.
abstract	Deklariert eine Klasse oder Methode als abstrakt.
...	...



Auszug von Operatoren

Operator	Verwendung
!, &&, , ^	Logische Operatoren NOT, AND, OR, XOR
==	Vergleichsoperator
<, <=, >, >=	Numerische Vergleiche
++, --	Inkrement und Dekrement
+=, /=, *=	Zuweisung mit numerischer Operation
+=	Zuweisung mit String-Konkatenation



Java Code–Konventionen

- Klassen
 - Variablennamen mit semantischer Bedeutung.
 - Klassennamen sollten Substantive sein und mit Großbuchstaben beginnen.
- Schnittstellen
 - Konventionen wie bei Klassen.
- Automatische Code-Formatierung in Eclipse:
`<STRG> + <SHIFT> + <F>`

Oracle Code-Conventions:

<http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>

Java Code–Konventionen

■ Attribute

- Variablennamen mit semantischer Bedeutung wählen.
- Variablennamen beginnen mit einem Kleinbuchstaben, interne Wörter mit Großbuchstaben getrennt (Camel Case).

■ Methoden

- Methodennamen beginnen mit einem Verb.
- Der erste Buchstabe wird kleingeschrieben, interne Wörter mit Großbuchstaben getrennt (z.B. `addAll(...)`).

■ Konstanten

- Nur Großbuchstaben, Worte werden durch „_“ getrennt.

Packages

- Ein Package enthält Typen (Klassen, Interfaces,...), die sich einen Geltungsbereich (Namespace) teilen.
- Ein Package erlaubt Zugriffsbeschränkungen.
- Verwendung um die Zusammengehörigkeit von Typen festzulegen und Namenskonflikte zu vermeiden.
- Package–Namen werden in Kleinbuchstaben festgelegt.



Primitive Datentypen

Primitive Datentypen

- Primitive Datentypen sind keine Klassen.
- Für jeden primitiven Datentyp gibt es eine eigene, so genannte, Wrapperklasse.
- Autoboxing/Unboxing ermöglicht automatische Konversion zwischen primitiven Datentyp und Wrapperklasse.

Basistyp	Wrapper-Klasse
int	Integer
long	Long
float	Float
boolean	Boolean



Main-Methode

main-Methode

- Die main-Methode dient als Einsprungspunkt für einen Programmstart.
- Eine main-Methode ist innerhalb einer Klasse definiert.
- Eine Applikation kann mehrere main-Methoden haben.
- Die konkrete main-Methode wird beim Programmstart der Applikation ausgewählt.
- In einer Klasse kann es nur eine main-Methode geben.

main-Methode

- Die main-Methode ist eine statische Methode.
- Über die main-Methode können Argumente an die Applikation übergeben werden.
- Der Rückgabotyp ist *void*.
- Am Ende der main-Methode terminiert das Programm.

```
public class Application {  
  
    public static void main( String[] args ) {  
        Storage storage = new Storage();  
        storage.store("New Element");  
    }  
}
```

main-Methode

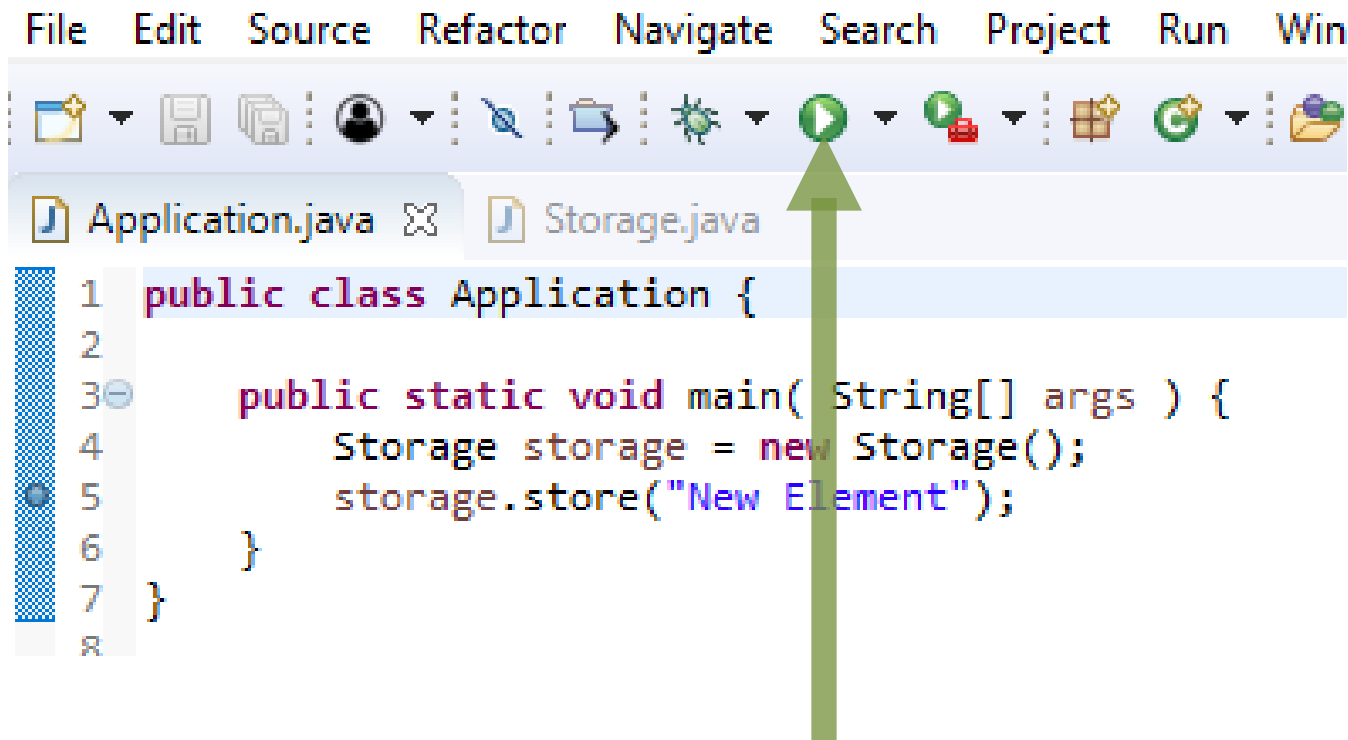
- Argumente können wie folgt an eine Java-Anwendung übergeben werden.

```
public class SayHello {  
  
    public static void main( String[] args ) {  
        if ( args.length > 0 )  
            System.out.println( "Hallo " + args[0] );  
        else  
            System.out.println( "Hallo" );  
    }  
}
```

```
C:> javac SayHello.java  
C:> java SayHello Thomas  
Hallo Thomas  
C:>
```

Ausführen eines Programms

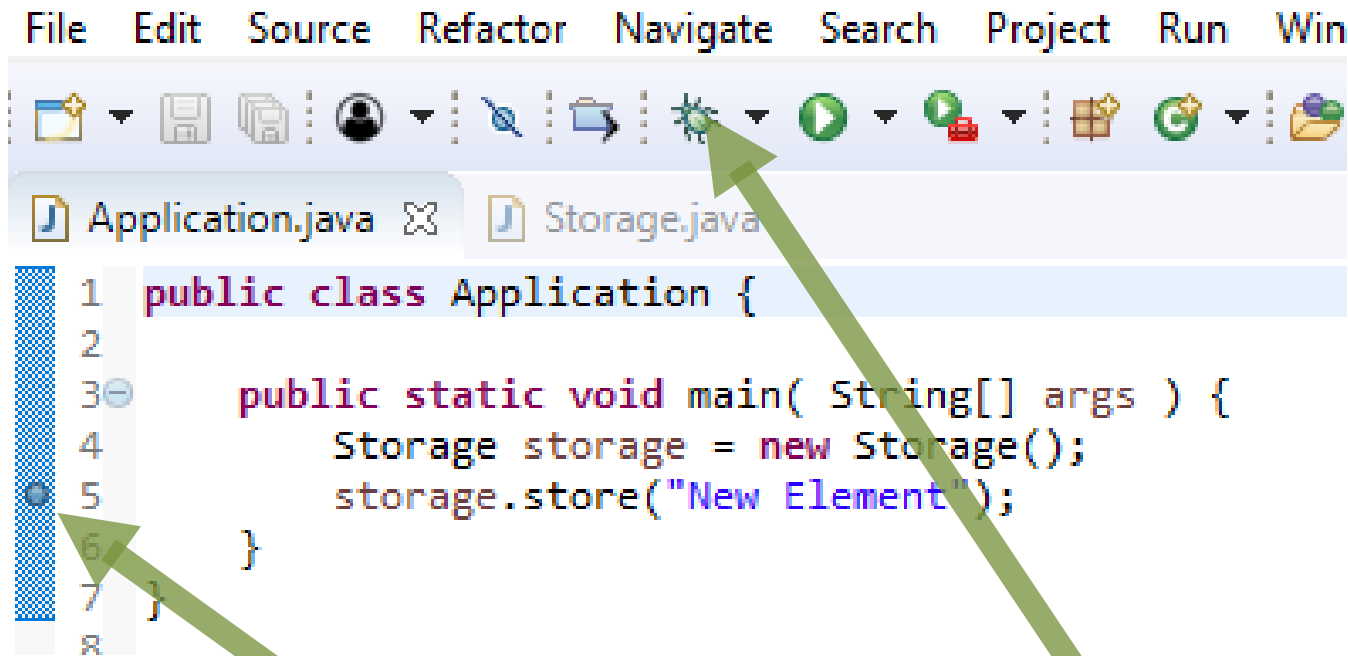
- Ausführen einer *main*-Methode mittels Eclipse:



Run As > Java Application

Debuggen eines Programms

- Debuggen eines Programms mittels Eclipse:



Breakpoint

Debug As > Java Application



Ablaufsteuerung

if – Anweisung

- Die *if*-Anweisung ist eine Kontrollstruktur, die mehrere potentielle Ausführungspfade hat.
- Sie erlaubt die konditionale Ausführung von Anweisungsblöcken.
- Konditionen können über logische Operatoren miteinander verbunden werden.
- Es können beliebig viele *else-if*-Zweige in einer *if*-Anweisung vorhanden sein.



if – Anweisung

```
if ( __CONDITION-1__ ) {  
    // Anweisungen des Blocks  
  
} else if ( __CONDITION-2__ ){  
    // Anweisungen des Blocks  
    // der Block kann auch entfallen  
  
} else {  
    // Anweisungen des Blocks  
    // der Block kann auch entfallen  
}
```

```
if ( x < 100) {  
    // Ausführung falls x kleiner als 100 ist  
} else if ( x == 100 || x == 200){  
    // Ausführung falls x gleich 100 oder 200 ist  
} else {  
    // Ausführung in allen anderen Fällen  
}
```

switch – Anweisung

- Nützlich, um zwischen einer Anzahl von möglichen Werten zu wählen.
- Die *switch*–Anweisung ist eine Kontrollstruktur, die mehrere potentielle Ausführungspfade hat.
- Unterstützte Typen der *switch*–Variable ab Java Version 1.8
 - ENUMs
 - String
 - Character
 - Byte
 - Short
 - Integer, integer

switch – Anweisung

```
switch ( SWITCH_VARIABLE ) {  
  
    case CONST1 :  
        // Anweisungen des Blocks  
        ausdruck1;  
        break;  
    case CONST2 :  
        // Anweisungen des Blocks  
        // der Block kann auch entfallen  
        ausdruck2;  
        break;  
    default :  
        // Anweisungen des Blocks  
        // der Block kann auch entfallen  
        ausdruck3;  
        break;  
}
```



Bedingungs – Operator

- Der Bedingungsoperator „?:“ ist ein ternärer Operator (siehe auch ANSI-C).
- Kurzform der *if-else*-Anweisung.

```
<CONDITION> ? <COMMAND_1> : <COMMAND_2>
```

```
int bigger = (a > b) ? a : b;  
// Der Wert von bigger ist a wenn a größer als b ist,  
// in allen anderen Fällen ist der Wert von bigger b
```



Schleifen

schleifen – Anweisung

- Erlaubt die wiederholte Ausführung von Anweisungsblöcken.
- Die Anzahl der Wiederholungen wird über Bedingungen gesteuert.
- Java unterstützt mehrere Schleifen-Konstrukte.
- Innerhalb von Schleifen können kontrollierte Sprünge durchgeführt werden.
 - Schleifen können während der Ausführung abgebrochen werden.
 - Während der Ausführung einer Schleifeniteration kann diese abgebrochen und mit der nächsten fortgesetzt werden.



while – Schleife

- Prüfung der Schleifenbedingung vor jedem Schleifeneintritt.

```
while ( BOOL_CONDITION ) {  
    // Anweisungen des Blocks  
}
```

```
int i = 0;  
while ( i < 10 ) {  
    System.out.println( "i= " + i );  
    i++;  
}
```



do-while – Schleife

- Prüfung der Schleifenbedingung nach jedem Schleifendurchlauf.

```
do {  
    // Anweisungen des Blocks  
} while ( BOOL_CONDITION );
```

```
int i = 0;  
do {  
    System.out.println( "i= " + i );  
    i++;  
} while ( i < 10 );
```


for – Schleife

- Schleife mit Zählvariable
- Der Rumpf wird ausgeführt solange die Schleifenbedingung wahr ist.

```
for (int i = 0; i < 10; i++) {  
  
    // Anweisungen des Blocks  
    System.out.println( "i= " + i );  
  
}
```



Kontrollierte Sprünge

- Möglichkeit den Ablauf von Schleifen zu beeinflussen.

Schlüsselwort	Funktion
<code>continue</code>	Beendet den aktuellen Schleifendurchlauf.
<code>break</code>	Beendet die Schleife. Bei verschachtelten Schleifen wird nur die innerste Schleife beendet.





Listen

Collection in Java

- *Collection* ist ein Interface das Methoden auf Sammlungen von Objekten definiert.
- Unterschiedliche Implementierungen von Collections sind verfügbar (z.B. *ArrayList*, *LinkedList*,...).
- Für Details zu Collections, siehe auch Foliensatz Generics.

```
Collection<String> list = new ArrayList<String>();
```

Typ der Elemente in der Sammlung



Listen in Java

- *List* ist ein Interface das Methoden auf Listenobjekten definiert.
- *List* leitet vom Interface *Collection* ab.
- Unterschiedliche Listen sind verfügbar (*ArrayList*, *LinkedList*,...).
- Für Details zu Listen, siehe auch Foliensatz Generics.

```
List<String> list = new ArrayList<String> ();
```

Typ der Listenelemente

Listen – Beispiel

```
Collection<String> greekAlphabet = new ArrayList<String>();  
// hinzufügen von Elementen  
greekAlphabet.add( „ALPHA“ );  
greekAlphabet.add( „BETA“ );  
  
List<String> list2 = Arrays.asList( „GAMMA“, „DELTA“ );  
greekAlphabet.addAll( list2 );  
  
System.out.println( greekAlphabet );  
// Konsole: [ALPHA, BETA, GAMMA, DELTA]  
System.out.println( greekAlphabet.size() );  
// Konsole: 4  
  
// löschen von Elementen  
greekAlphabet.remove( „ALPHA“ );  
// Konsole: [BETA, GAMMA, DELTA]  
  
if ( greekAlphabet.contains( „GAMMA“ ) ) {  
    System.out.println( „Contains the symbol GAMMA“ );  
}
```

Verkürzte *for* – Schleife

- Sehr kompakte Schreibweise von Schleifen.
- Die verkürzte *for*–Schleife kann verwendet werden, wenn das *Iterable* Interface implementiert wird.
- Das Interface *Collection* bzw. *List* implementieren das Interface *Iterable*.
- **Hinweis:** Wird die Liste beim Durchlaufen verändert kommt es zu einer *ConcurrentModificationException*!

Verkürzte for – Schleife

```
Collection<String> list = new ArrayList<String>();  
// Einfügen von Elementen in die Liste  
  
// Verkürzte for-Schleife setzt Iterable voraus.  
for ( String element : list ) {  
    System.out.println( element );  
}
```

```
// Modifikation der Liste während des Durchlaufens  
list.add(„Element to remove“);  
for ( String element : list ) {  
    list.remove(„Element to remove“); // EXCEPTION  
  
    // Änderung der Liste: ConcurrentModificationException  
}
```


Iteratoren

- *Iterator* definiert ein Interface mit dem Listen durchlaufen werden können.
- Das Interface *Collection* bzw. *List* implementieren das Interface *Iterator*.
- **Hinweis:** Erfolgt keine korrekte Abfrage ob es noch weitere Elemente gibt (*hasNext()*) wird bei Zugriff auf das nächste Element eine *NoSuchElementException* geworfen falls keines existiert.

Iteratoren - Beispiel

```
Collection<String> list = new ArrayList<String>();  
// Einfügen von Elementen in die Liste  
  
Iterator<String> iteratorObject = list.iterator();  
  
// solange es noch ein nächstes Element gibt  
while ( iterator.hasNext() ) {  
  
    // liefert das nächste Element  
    // springt danach um eine Stelle weiter  
    String currentElement = iteratorObject.next();  
}
```





Fehlerbehandlung & Stacktrace

Fehlerbehandlung mit Exceptions

- Java-Methoden können eine Ausnahme (Exception) auslösen, wenn sie aus irgendeinem Grund versagen.
- Ausnahmebehandlung durch sogenannte Exceptions:
 - Auslösen (**throw**) einer Ausnahme sobald ein Fehler eintritt.
 - Abfangen (**catch**) einer Ausnahme in einer übergeordneten Methode.

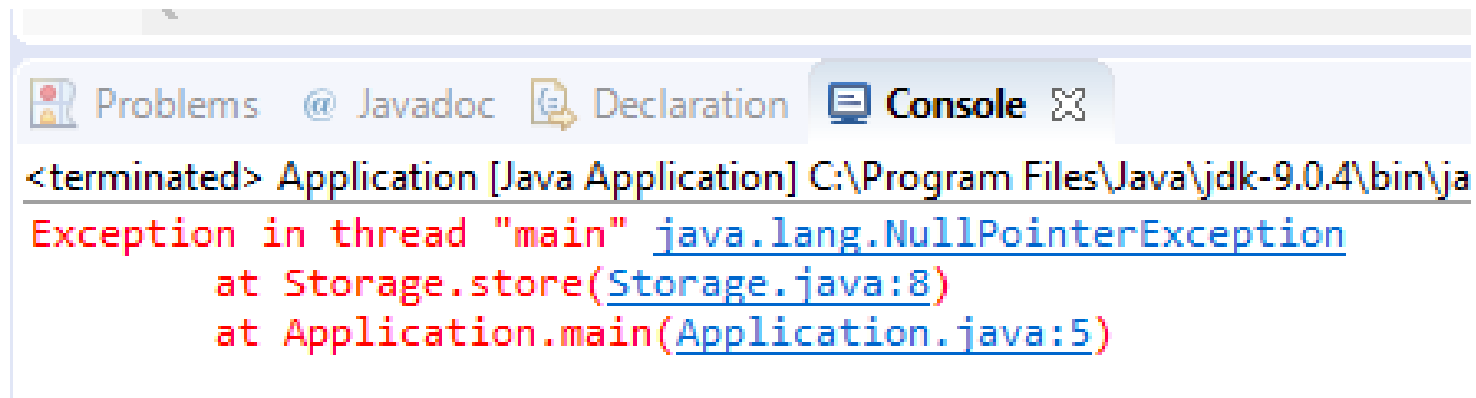
Fehlerbehandlung

- Wird eine aufgetretene Exception nicht behandelt dann fängt in letzter Instanz die Laufzeitumgebung die Exception und es erfolgt eine Fehlerausgabe mittels Stacktrace.
- Der Stacktrace kann verwendet werden um die Fehlerursache einzugrenzen.
- Details finden Sie im Kapitel Exceptions.



Stacktrace

- Stacktrace: Ausgabe und Interpretation des Inhalts des Stacks.
- Der Stack spiegelt die Aufrufreihenfolge von Methoden wieder und kann verwendet werden um den Grund einer Exception einzugrenzen.
- Die Zeile, die den Fehler auslöst, steht an oberster Stelle.




```
<terminated> Application [Java Application] C:\Program Files\Java\jdk-9.0.4\bin\ja
Exception in thread "main" java.lang.NullPointerException
    at Storage.store(Storage.java:8)
    at Application.main(Application.java:5)
```



Reihenfolge der Aufrufe

Stacktrace Nachvollziehen



```
Application.java Storage.java
1 public class Application {
2
3 public static void main( String[] args ) {
4     Storage storage = new Storage();
5     storage.store("New Element");
6 }
7 }
8
```

Aufrufe entsprechend
dem Stacktrace

```
Application.java Storage.java
1 import java.util.List;
2
3 public class Storage {
4
5     private List<String> list = null;
6
7 public void store(String element) {
8     list.add(element);
9 }
10 }
```

NullPointerException in Zeile 8 da die Variable list nicht initialisiert ist.

Fehlersuche mittels Debugger

- Debuggen erlaubt das Schrittweise durchgehen eines Programms.
- Es können Breakpoints gesetzt werden. Diese erlauben das Programm an einer bestimmten Stelle im Code anzuhalten.
- Breakpoints können auch konditional sein.
- Eclipse bietet einen Überblick der Variablen und ihrer Werte während des Debuggens.



Klassen

Was ist eine Klasse?

- Jede Klasse definiert einen Typ.
- Klassen sind „Baupläne“/Schablonen für Objekte (Instanzen).
- Klassen bieten Konstruktoren zur Objekterzeugung (Instanziierung) an.
- Von einer Klasse können mehrere Objekte erzeugt werden.

Was ist eine Klasse?

- Klassen spezifizieren Attribute, die gemeinsam den Zustand des Objekts (Eigenschaften) repräsentieren.
- Klassen spezifizieren ihr Verhalten über Methoden.
- Klassen bieten öffentliche Schnittstellen um den Zustand eines Objektes von außen ändern zu können.

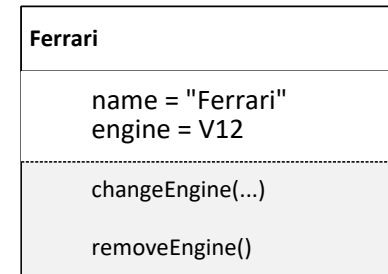
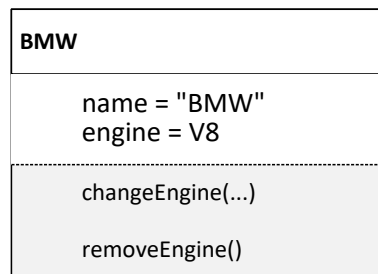
Beispiel für eine Klasse

```
public class Vehicle {  
    private Motor engine = null; // Attribut, Instanzvariable  
    private String name = null; // Attribut, Instanzvariable  
  
    public Vehicle() { // Konstruktor zur Objekterzeugung  
        this.name = new String("");  
    }  
  
    // Methoden die Verhalten der Klasse definieren  
    public void changeEngine(Motor engine) {  
        this.engine = engine;  
    }  
    // Methoden für Zugriff auf „geschützte“ Attribute  
    public void removeEngine() {  
        this.engine = null;  
    }  
}
```



Was ist ein Objekt?

- Objekte (auch Instanzen genannt) sind konkrete Realisierungen von Klassen.
- Objekte haben einen konkreten Zustand (Instanzvariablen der Klasse) und ein Verhalten (Methoden der Klasse).
- Jedes Objekt hat zur Laufzeit eine Repräsentation im Speicher.
- Manipulationen des Objektzustands erfolgt über die in der zugehörigen Klasse definierten Methoden.



Die Java *Object*-Klasse

- Alle Klassen leiten von *Object* ab.
- Alle selbst erstellten Klassen erben implizit auch von *Object*.
- *Object*-Methoden können mit eigener Implementierung überschrieben werden.



Konstruktor

- Konstruktoren werden verwendet um Objekte von Klassen zu erzeugen.
- Aufruf beim Erzeugung von Objekten einer Klasse.
- Initialisierung des Objektzustands.
- Name des Konstruktors muss gleich sein mit dem Klassennamen.
 - Wird kein Konstruktor explizit definiert, erstellt der Compiler einen Default-Konstruktor (= mit leere Parameterliste). Ist einer definiert, wird kein Default-Konstruktor erstellt.
 - Mehrere Konstruktoren mit unterschiedlicher Signatur sind möglich.

Konstruktor

- Verwendung von set-Methoden im Konstruktor sollte vermieden werden.
- Durch das Überschreiben der set-Methoden könnte es zu ungewolltem Verhalten kommen (siehe Overriding).

“Constructors must not invoke overridable methods, directly or indirectly. If you violate this rule, program failure will result. The superclass constructor runs before the subclass constructor, so the overriding method in the subclass will get invoked before the subclass constructor has run. If the overriding method depends on any initialization performed by the subclass constructor, the method will not behave as expected.”

[Effective Java 2nd Edition, Joshua Bloch, Item 17, p89]

Constructor Chaining

- *Constructor Chaining* bedeutet das in einem Konstruktor ein anderer Konstruktor aufgerufen wird:
 - a) Ein anderer Konstruktor der eigenen Klasse wird aufgerufen.
 - b) Ein Konstruktor der direkten Superklasse wird aufgerufen.
- Der Aufruf des anderen Konstruktors muss an erster Stelle (vor jeder anderen Operation) stehen.

Konstruktor Beispiel

```
public class Vehicle {
    private Motor engine = null;
    private String name = null;

    public Vehicle() {
        this.name = new String(" ");
    }

    public Vehicle(Motor engine) {
        this(); // constructor chaining
        this.engine = engine;
    }

    public Vehicle(String name, Motor engine) {
        // Aufruf eines anderen Konstruktors möglich
        this.name = name;
        this.engine = engine;
    }
}
```



Objekterzeugung - Ablauf

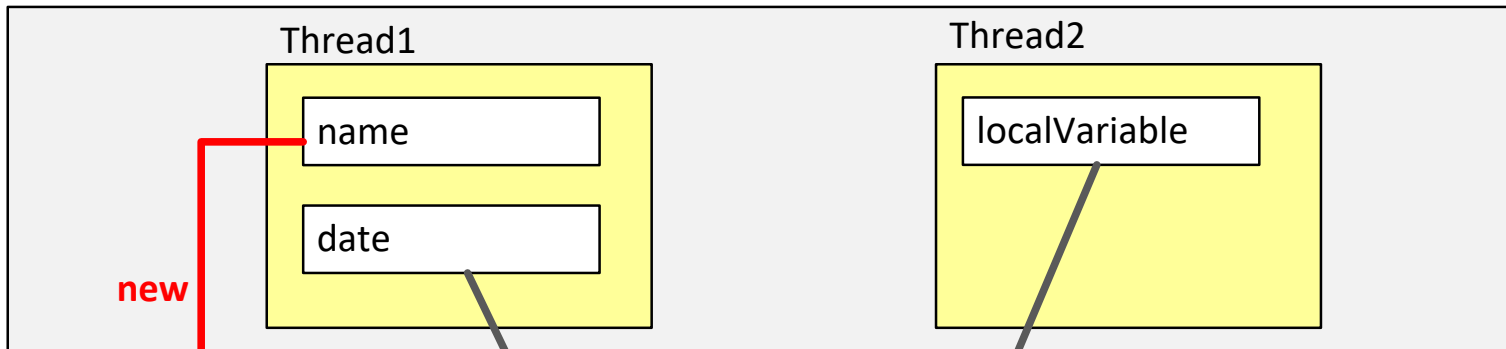
```
Vehicle car = new Vehicle( "BMW" , engineVariable );
```

1. Die Referenzvariable *car* wird am Stack angelegt und mit dem Wert *null* belegt.
2. Der *new*-Operator der Klasse *Vehicle* wird aufgerufen. Auf dem Java-Heap wird der Platz für ein Objekt der Klasse *Vehicle* zur Verfügung gestellt. Es wird **instanziert**.
3. Die Datenfelder des Objekts werden **initialisiert**.
4. Der Methodenblock des entsprechenden Konstruktors wird ausgeführt.
5. Der *new*-Operator gibt die Referenz auf das erzeugte Objekt zurück.
6. Die Referenz wird in der Variable *car* gespeichert.

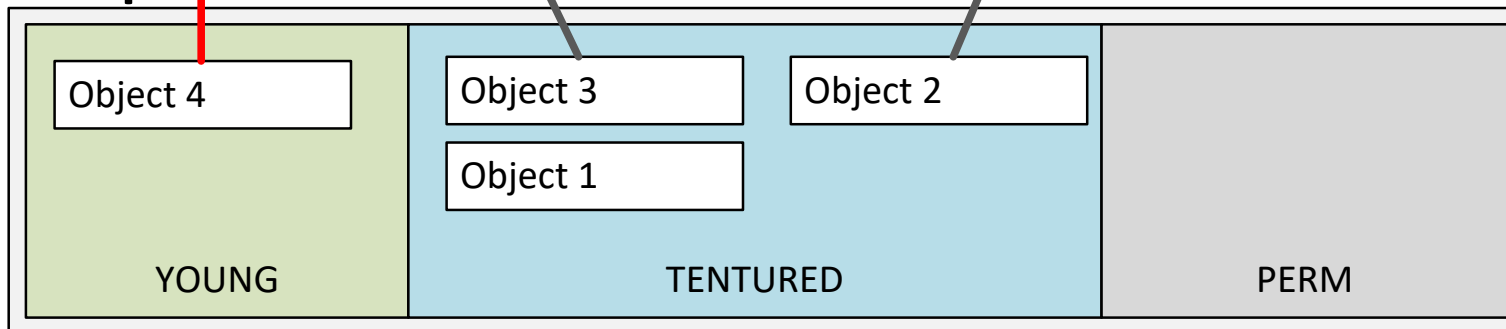
Objekterzeugung

```
String name;  
name = new String("Erzeuge Objekt" );
```

Stack



Heap



Garbage Collection

- Speicher von Objekten, die auf dem Heap liegen, wird automatisch freigegeben wenn nicht mehr darauf referenziert wird.
- Der Garbage Collector erkennt auch zyklische referenzierte Objekte die nicht mehr verwendet werden.
- Memory Leaks sind weiterhin möglich, wenn die Referenzen zu den Objekten nicht entfernt werden.
- finalize-Methode wird vor dem Zerstören des Objekts aufgerufen. Der Aufrufzeitpunkt ist aber nicht deterministisch.

Rekursive Klassen

- Klassen können Attribute beliebigen Typs haben.
- Hat ein Attribut denselben Typ (oder einen Obertyp) wie die Klasse, so spricht man von einer rekursiven Klasse.
- Klassen die Listen desselben Typs (oder eines Obertyps) als Attribut haben, bilden ebenfalls eine rekursive Klasse.
- Rekursive Klassen werden häufig in komplexen, verketteten Datenstrukturen eingesetzt (z.B.: Graphen, verkettete Listen, Baumstrukturen...).

Beispiel – Rekursive Klasse

```
public class Node {
    private Node leftChild = null; // Attribut des gleichen Typs
    private Node rightChild = null; // Attribut des gleichen Typs
    private int value = 1;
    // Konstruktor: initialisieren der Attribute.
    ...

    public int getSumOfNodes() {
        return value + leftChild.getValue() + rightChild.getValue();
    }

    // Typischerweise werden hier noch Überprüfungen auf NULL
    // durchgeführt.
    public int getSumOfAllChildNodes() {
        return value + leftChild.getSumOfAllChildNodes() +
            rightChild.getSumOfAllChildNodes();
    }

    public int getValue() {
        return this.value;
    }
}
```





Methoden

Methoden

- Methodenaufruf
- Methodensignatur
- Parameterübergabe
- Statische Methoden
- Overloading



Methodenaufruf

- Aufruf einer Instanzmethode
- Aufruf einer statischen Methode

```
// Aufruf einer Instanzmethode auf dem Objekt myObject  
myObject.methodName(param1, param2 ,...);
```

```
// Aufruf einer statischen Methode der Klasse ClassName  
ClassName.methodName(param1, param2 ,...);
```

Methodenaufruf

- Innerhalb einer Methode können Methodenaufrufe erfolgen.
- Der Methodenaufruf kann wieder auf demselben Objekt erfolgen.
- Ist der Methodenaufruf auf demselben Objekt und derselben Methode, so spricht man von einem rekursiven Aufruf.



Methodenaufruf- Rekursion

```
public class Car {  
    private Speedometer speedometer = new Speedometer(this);  
  
    public void accelerate(int targetValue) {  
        this.increaseSpeed();  
  
        // Bei Erreichung des Zielwerts stoppt die Rekursion  
        if(speedometer.getSpeed() < targetValue) {  
            // Hier erfolgt ein rekursiver Aufruf  
            this.accelerate(targetValue);  
        }  
    }  
}
```

Methodensignatur

- Methodensignatur definiert die formale Schnittstelle einer Methode.
- Namen der Funktion und der Anzahl und Reihenfolge der kompatiblen Parameterdatentypen.
- Bei Java gehört der Rückgabotyp nicht zur Signatur. Bei streng strukturierten Sprachen ist er Teil davon.

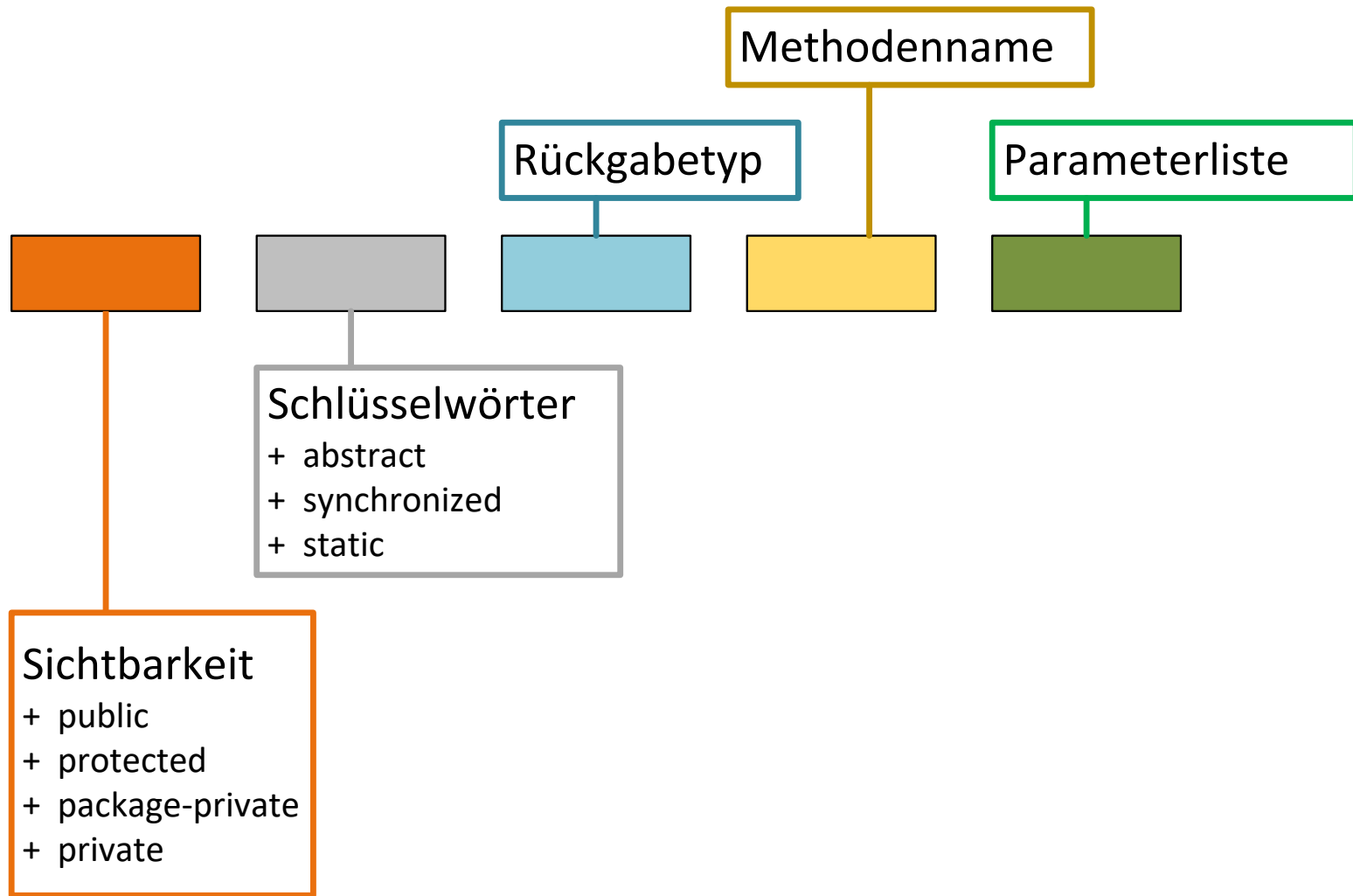
```
public Integer add(Integer a, Integer b) {...}
```

```
public Integer add(Integer a, Integer b, Integer c) {...}
```

```
// Signatur: add(Integer, Integer) => nicht eindeutig!
```

```
public Float add(Integer a, Integer b) {...}
```

Methodensignatur



Methodensignatur - Beispiele

public abstract String multiply (Integer a, Integer b);

public String toLower (String str){...};

protected Integer getAge (String str){...};

protected static synchronized void setAge (Integer str){...};

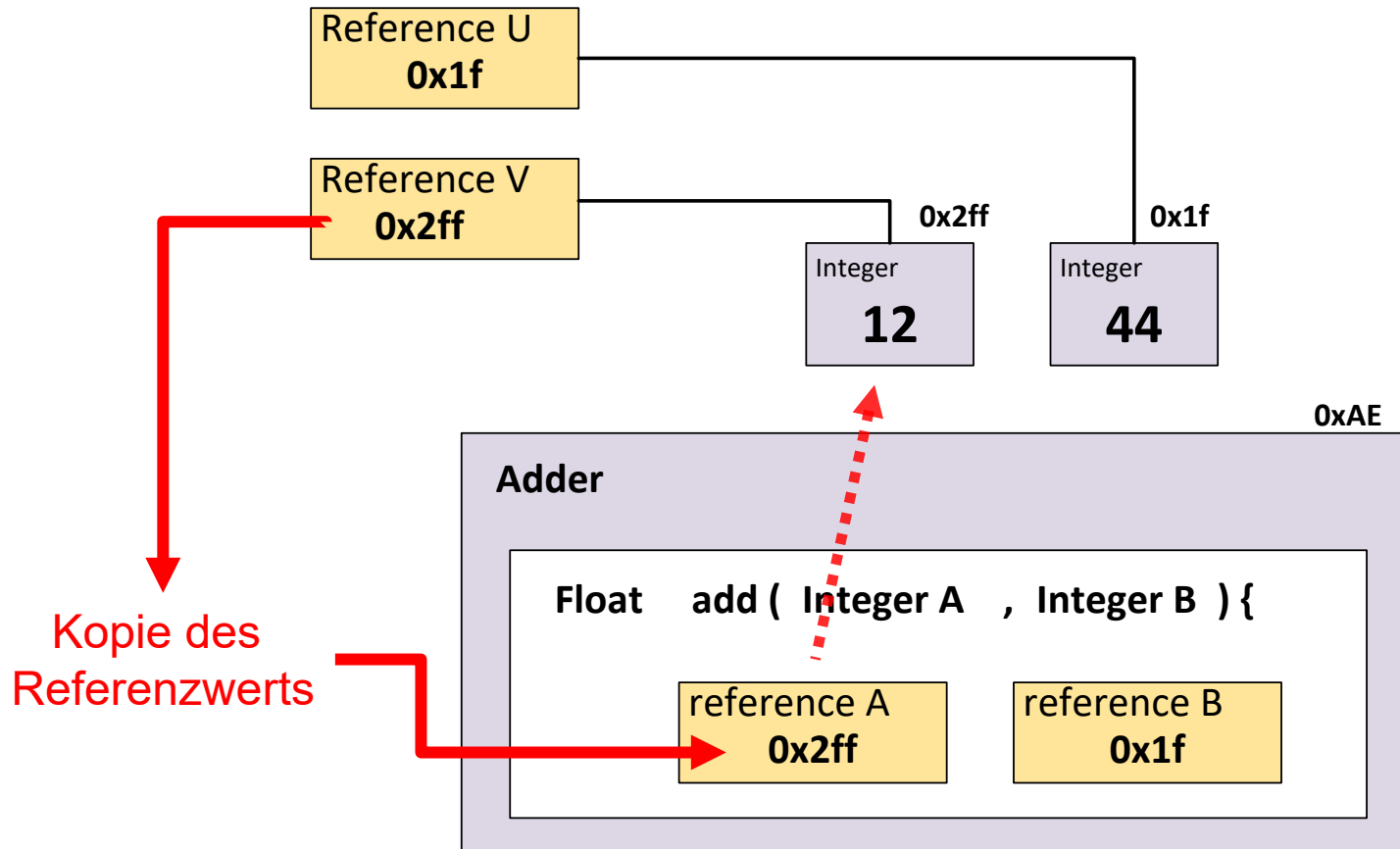
Parameterübergabe

- Primitive Datentypen werden als Kopie übergeben.
- Bei Objekten wird eine Kopie der Referenz auf das Objekt übergeben (*Copy-By-Value-Reference*).



Parameterübergabe

- Parameterübergabe: Copy-By-Reference-Value



Parameterübergabe

```
public class ParameterPassing {  
  
    public static void main(String[] args){  
        // creating objects of type Integer (class)  
        Integer U = new Integer(12);  
        Integer V = new Integer(44);  
  
        // creating object of type Adder (class)  
        Adder adder = new Adder();  
  
        // passing the objects  
        float operationResult = adder.add(u,v);  
    }  
}
```



Overloading

- Wenn zwei oder mehr Methoden, innerhalb einer Klasse, den exakt gleichen Methodennamen haben.
- Alle Methoden haben aber unterschiedliche Parameterlisten.
- Java berücksichtigt dabei den Rückgabetyp nicht.
- Ist ein Mechanismus der sich zur Compile-Time zeigt.

```
public class Overloading {  
    public Integer add(Integer a, Integer b) {...}  
    public Integer add(Integer a, Integer b, Integer c) {...}  
    public Float add(Integer a, Integer b, Float c) {...}  
}
```

Statische Methode

- Eine statische Methode ist nicht mit einem Objekt verbunden.
- Keine Instanz der Klasse notwendig um die Methode aufzurufen.
- Der Aufruf sollte über den Klassennamen erfolgen.

```
public class StaticExample {  
    public static Integer add(Integer a, Integer b) { ... }  
}
```



Aufruf über Klassennamen

```
int result = StaticExample.add(12, 44);
```



Abstrakte Klassen

Abstrakte Klassen

- Abstrakte Klassen definieren einen Typ.
- Von abstrakten Klassen können keine Instanzen erzeugt werden.
- Von einer abstrakten Klasse kann abgeleitet werden.
- Abstrakte Klassen können (müssen aber nicht) abstrakte Methoden beinhalten.
- Enthält eine Klasse eine abstrakte Methode muss die Klasse abstrakt sein.

Abstrakte Methoden

- Abstrakte Methoden legen nur die Methodensignatur fest.
- Es gibt keine Implementierung der Methode.
- Erbt eine nicht abstrakte Klasse von einer abstrakten Klasse müssen alle abstrakten Methoden implementiert werden.

```
public abstract class Rectangle {  
    // Attribute, Konstruktoren, ...  
  
    // abstrakte Methode - kein Methodenblock  
    public abstract void draw(Graphics drawArea);  
}
```

Beispiel für eine abstrakte Klasse

```
public abstract class Rectangle {  
    private int width = 0;    // Instanzvariable  
    private int height = 0;   // Instanzvariable  
  
    public Rectangle(int width, int height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    // Methode die ein Teilverhalten der Klasse definieren  
    public int calculateArea() {  
        return width * height;  
    }  
    // abstrakte Methode  
    public abstract void draw(Graphics drawArea);  
}
```




Vererbung

Vererbung

- Vererbung dient zur Erweiterung/Einschränkung einer bestehenden Klasse.
- Zugriffsmöglichkeit auf Eigenschaften/Verhalten einer Superklasse ist abhängig von deren Sichtbarkeiten.
- Expliziter Zugriff auf Methoden oder Attribute einer direkten Superklasse (Schlüsselwort *super*).



Vererbung

```
class [SUBCLASS] extends [SUPERCLASS] { }
```

- Vererbung bei Klassen und Interfaces möglich.
- Vererbung erzwingt keine Untertypenbildung im Sinne von OOP.
 - Vererbung nützt Ähnlichkeiten im Source-Code aus.
 - Echte Untertypen im Sinne von OOP haben ersetzbares Verhalten!

Vererbung

```
class [SUBCLASS] extends [SUPERCLASS] { }
```

- Zusicherungen können in Java nur dokumentiert werden (z.B. JavaDoc).
- **Kontrollfrage:** Warum ist Ersetzbarkeit wichtiger als Vererbung?



Vererbung

```
class [SUBCLASS] extends [SUPERCLASS] { }
```

- Instanzvariablen werden vererbt.
- Instanzmethoden werden vererbt.
- Neue Variablen können angelegt werden.
- Neue Methoden können angelegt werden.
- Konstruktoren werden nicht vererbt.



Vererbung

```
class [SUBCLASS] extends [SUPERCLASS] { }
```

- Instanzvariablen können überdeckt werden.
- Private Instanzmethoden können in Subklasse überdeckt werden.



Beispiel Vererbung - Spezialisierung

```
public class Tokenizer {  
    // Zerlegt einen Satz in Wörter. Trennzeichen  
    // sind Leerzeichen und Tabstop.  
    public List<String> explode (String sentence) {  
        String[] token = sentence.split("\\s+");  
        return Arrays.asList(tokens);  
    }  
}
```

```
public class AdvancedTokenizer extends Tokenizer {  
  
    // Ersetzt alle Punkte im Satz durch ein Leerzeichen  
    // und zerlegt den Satz danach in einzelne Wörter.  
    public List<String> explode (String sentence) {  
        String dotFreeSentence =  
            sentence.replaceAll("\\.", " ");  
        return super.explode(dotFreeSentence);  
    }  
}
```

Beispiel Vererbung - Erweiterung

```
public class SuperClass {  
    protected Integer lastResult = 0;  
  
    protected Integer multiply (Integer a, Integer b) {  
        lastResult = a * b;  
        return lastResult;  
    }  
}
```

```
public class SubClass extends SuperClass {  
  
    // Subklasse erbt die Methode multiply der Superklasse.  
    // Aufruf der multiply Methode aus der Subklasse möglich.  
  
    public Integer tenTimes (Integer a, Integer b) {  
        return 10 * this.multiply(a,b);  
    }  
}
```


Sichtbarkeiten

- Sichtbarkeit von Methoden
- Sichtbarkeit von Attributen

Schlüsselwort	Sichtbarkeit
public	Öffentlicher Zugriff explizit gewährt.
protected	Zugriff nur innerhalb der Klasse, aus Klassen des gleichen Package oder von einer abgeleiteten Klasse möglich.
private	Zugriff nur innerhalb der Klasse
-----	Package privat - Zugriff nur von Klassen möglich, die im gleichen Package liegen.

Sichtbarkeiten

Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
private	Yes	No	No	No






Konstrukturen und Vererbung

Konstruktoren und Vererbung

- Konstruktoren werden nicht vererbt.
- Ein Konstruktor der Subklasse muss als erste Anweisung einen Konstruktor der Superklasse aufrufen.
- Expliziter Aufruf auf des Konstruktors der direkten Superklasse mittels *super(...)*.
- Existiert nur der Default-Konstruktor in der Superklasse kann dieser explizite Aufruf entfallen.

Constructor Chaining & Vererbung

```
public class HardwareComponent {  
    // Deklaration von Attributen  
    private Float price;  
  
    public HardwareComponent(Float price) {  
        this.price = price;  
    }  
}
```



```
public class Resistor extends HardwareComponent {  
    private Float resistanceValue;  
  
    public Resistor (Float price, Float value) {  
        super(price);  
        resistanceValue = value;  
    }  
}
```



Attribute und Vererbung

Attribute

- Sichtbarkeit regelt Zugriffsmöglichkeit auf die Attribute.
- Methoden werden für einen geregelten Zugriff auf Attribute verwendet.
- Attribute mit gleichem Namen und Typ in Superklasse und Subklasse werden nicht überschrieben, sondern überdecken einander (Englisch: hiding field). Expliziter Zugriff auf das Attribut der Superklasse mit dem Schlüsselwort **super** möglich.

Beispiel – Sichtbarkeit

```
public class Person {  
    // Deklaration von Attributen  
    private String firstName;  
    protected String lastName;  
    public Integer age = 0;  
}
```

```
public class AccessAge {  
    // Zugriff auf ein sichtbares Attribut eines Objekts  
    private String streetName;  
    private String houseNumber;  
  
    public static void main (String[] args) {  
        // Zugriff auf ein sichtbares Attribut eines Objekts  
        Person p = new Person();  
        p.age = 23;  
        p.firstName = ""; // FEHLER - Kein Zugriff möglich  
    }  
}
```


Beispiel – Sichtbarkeit

```
public class Person {  
    // Deklaration von Attributen  
    private String firstName;  
    protected String lastName;  
    public Integer age = 0;  
}
```

```
public class PersonWithAddress extends Person {  
    // Deklaration von Attributen  
    private String streetName;  
    private String houseNumber;  
  
    public void access (String text) {  
        // Zugriff auf Attribute über this oder super möglich  
        this.lastName = text;  
    }  
}
```

Beispiel – Sichtbarkeit

```
public class Person {  
    // Deklaration von Attributen  
    private String firstName;  
    protected String lastName;  
    public Integer age = 0;  
}
```

```
public class PersonWithAddress extends Person {  
    // Deklaration von Attributen  
    private String streetName;  
    private String houseNumber;  
  
    public void access () {  
        this.houseNumber = „12/2/1“;  
        // Zugriff auf Attribute der Superklasse ?  
        super.firstName = „Max“; // FEHLER !  
    }  
}
```

Beispiel - Überdeckung

```
public class SuperAdder {  
    // Deklaration von Attributen  
    protected Integer lastResult = 0;  
    protected Integer totalSum = 0;  
  
    public Integer add (Integer a, Integer b) {  
        lastResult = a + b;  
        totalSum = totalSum + lastResult;  
        return this.lastResult;  
    }  
}
```

Beispiel - Überdeckung

```
public class SubAdder extends SuperAdder {  
    // Attribute werden nicht überschrieben sondern überdeckt!  
    protected Integer lastResult = 0;  
  
    public Integer add (Integer a, Integer b) {  
        Integer lastResult = a + b;  
        super.lastResult = this.lastResult;  
        this.lastResult = lastResult;  
        super.totalSum = this.totalSum + lastResult;  
        return this.lastResult;  
    }  
  
    public static void main(String[] args) {  
        SuperAdder adder = new SubAdder();  
        adder.add(10, 4);  
        System.out.println(adder.toString());  
        adder.add(14, 3);  
        System.out.println(adder.toString());  
    }  
}
```

Beispiel - Überdeckung

Tragen Sie die Attributwerte nach jeder Zuweisung (Zeile) in der Methode ein.

super		this	
lastResult	totalSum	lastResult	totalSum

Ihre Lösung können Sie anhand der Code-Beispiele aus TUWEL im Package `at.ac.tuwien.ict.hidingfields` überprüfen.



Methoden und Vererbung

Methoden und Vererbung

- Erweiterung des Verhaltens
 - Neue Methoden in Klasse einfügen
 - Overloading bestehender Methoden
- Veränderung des Verhaltens
 - Overriding bestehender Methoden

Overriding

- Methode in Subklasse hat die gleiche Signatur wie eine Methode in der Superklasse.
- Ist ein Mechanismus der sich zur Runtime zeigt.
- Zugriff auf überschriebene Methode nur mit **super** aus der Subklasse möglich.

```
public class OverrideSuper {  
    public Integer add(Integer a, Integer b) { ... }  
}
```

```
public class OverrideSub extends OverrideSuper {  
    public Integer add(Integer a, Integer b) { ... }  
}
```


Beispiel - Overriding

```
public class SuperClass {  
    protected Integer lastResult = 0;  
  
    public Integer add (Integer a, Integer b) {  
        lastResult = a + b;  
        return lastResult;  
    }  
}
```

```
public class SubClass extends SuperClass {  
  
    public Integer add (Integer a, Integer b) {  
        // Kontrollfrage: wo muss super verwendet werden,  
        // wo kann auch this geschrieben werden?  
        super.lastResult = a + b + super.add(a,b);  
        return this.lastResult;  
    }  
}
```



Interfaces

Interfaces

- Interfaces deklarieren einen Typ.
- Ein Interface ist eine Schnittstellendefinition.
- Legt die verfügbaren Operationen auf einem Objekt dieses Typs fest.
- Verhaltensspezifikation mittels JavaDoc.
- Keine Aussage über die konkrete Implementierung.



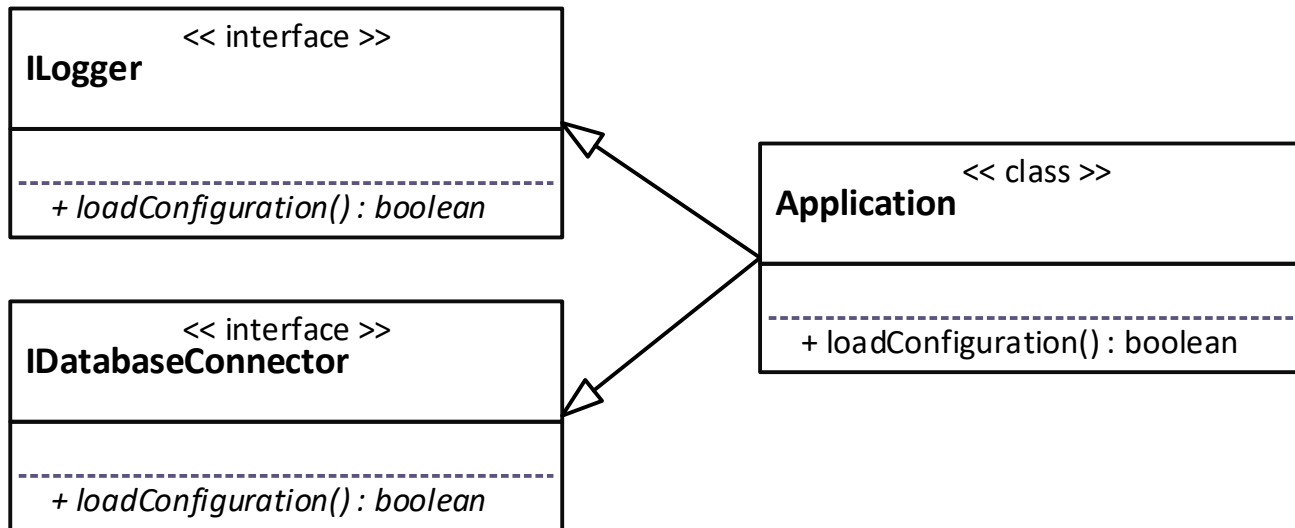
Interfaces

- Die Methoden sind abstrakt (explizit durch Schlüsselwort *abstract* oder implizit).
 - **Sonderstellung:** Ab Java 1.8 sind in einem Interface statische Methoden, *default*-Methoden sowie private Methoden mit Implementierung erlaubt.



Interfaces

- Werden mehrere Interfaces implementiert und haben diese eine Operation mit der gleichen Signatur, muss der Programmierer Überlegungen bzgl. der Kompatibilität der Operationen vornehmen. Zusicherungen werden in Java nur dokumentiert.



Interface

```
public interface IRecord {  
    // Keine Deklaration von Attributen möglich!  
    public String getIdentifizier ();  
}
```

```
public interface IProduct extends IRecord {  
    // Keine Deklaration von Attributen möglich!  
  
    // Methode explizit als abstrakt deklariert.  
    public abstract String getProductName ();  
  
    // Methoden eines Interface sind implizit abstrakt.  
    public float getPrice ();  
}
```



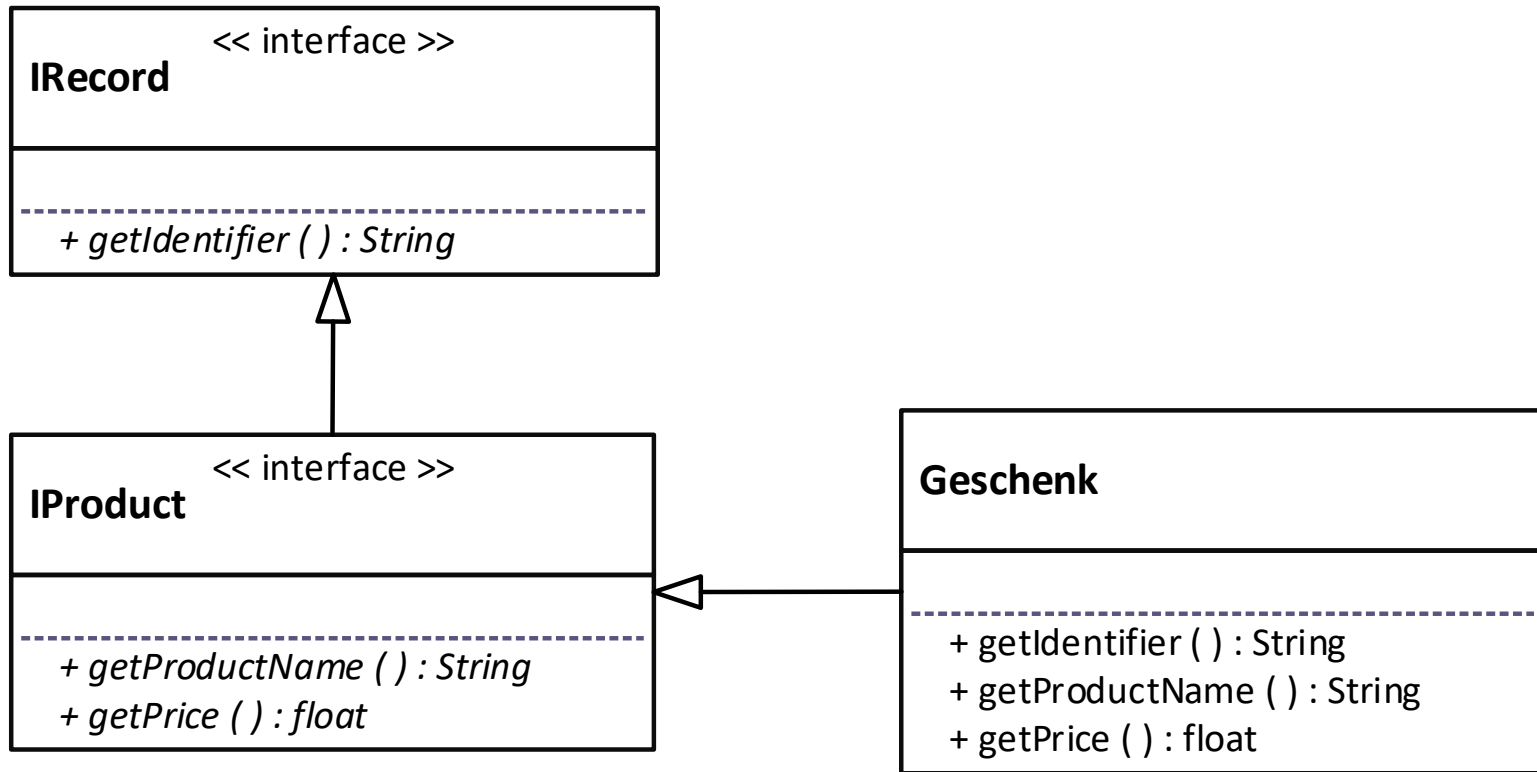
Interface

```
public interface IRecord {  
    // Keine Deklaration von Attributen möglich!  
    public String getIdentifizier ();  
}
```

```
public class Geschenk implements IProduct {  
  
    private String identifer;  
    private String productname;  
    private float price;  
  
    // Methoden des Interface müssen implementiert werden!  
    public String getIdentifizier () { ... }  
    public String getProductName () { ... }  
    public float getPrice () { ... }  
}
```

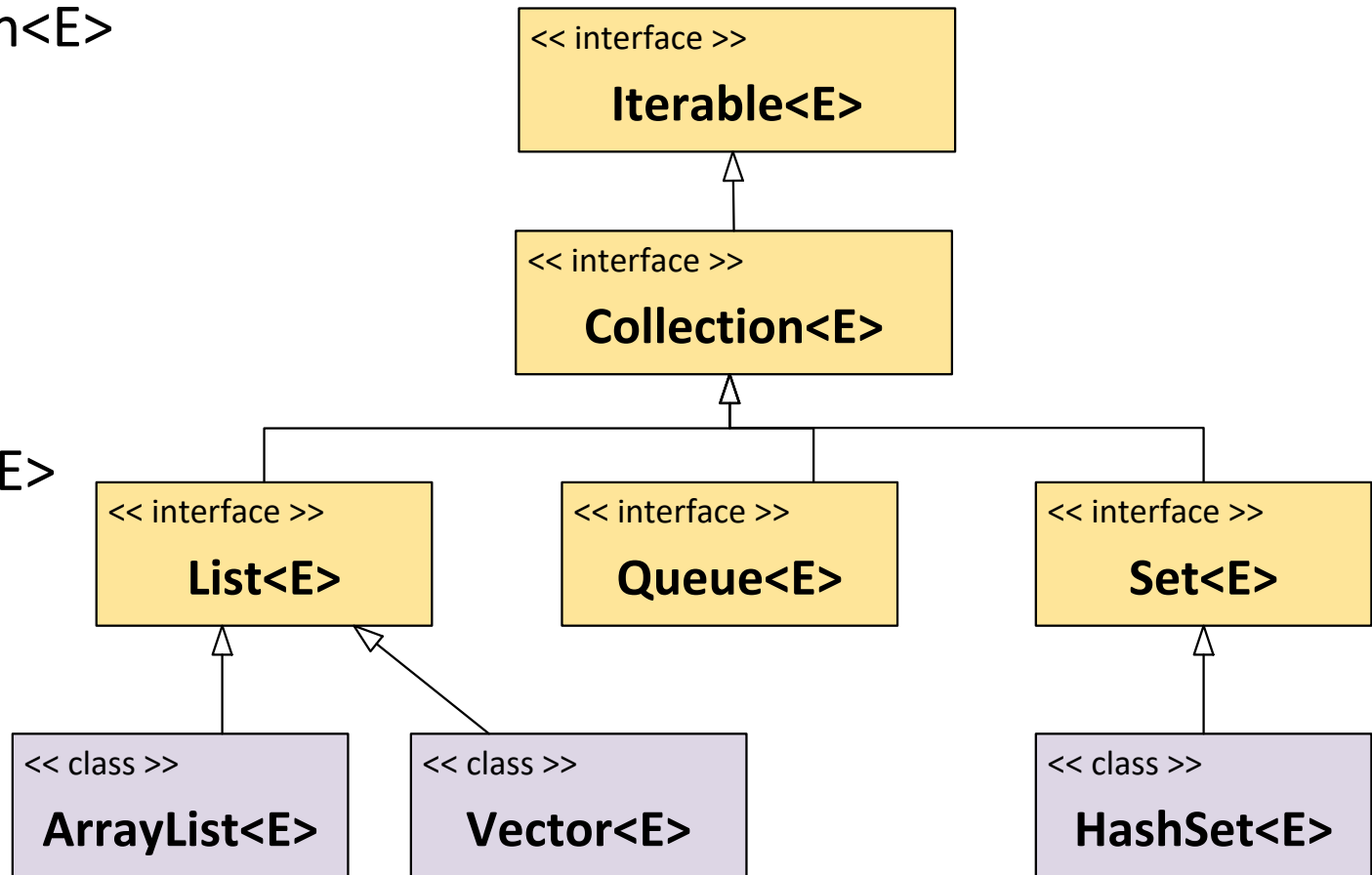


Interface - Vererbung

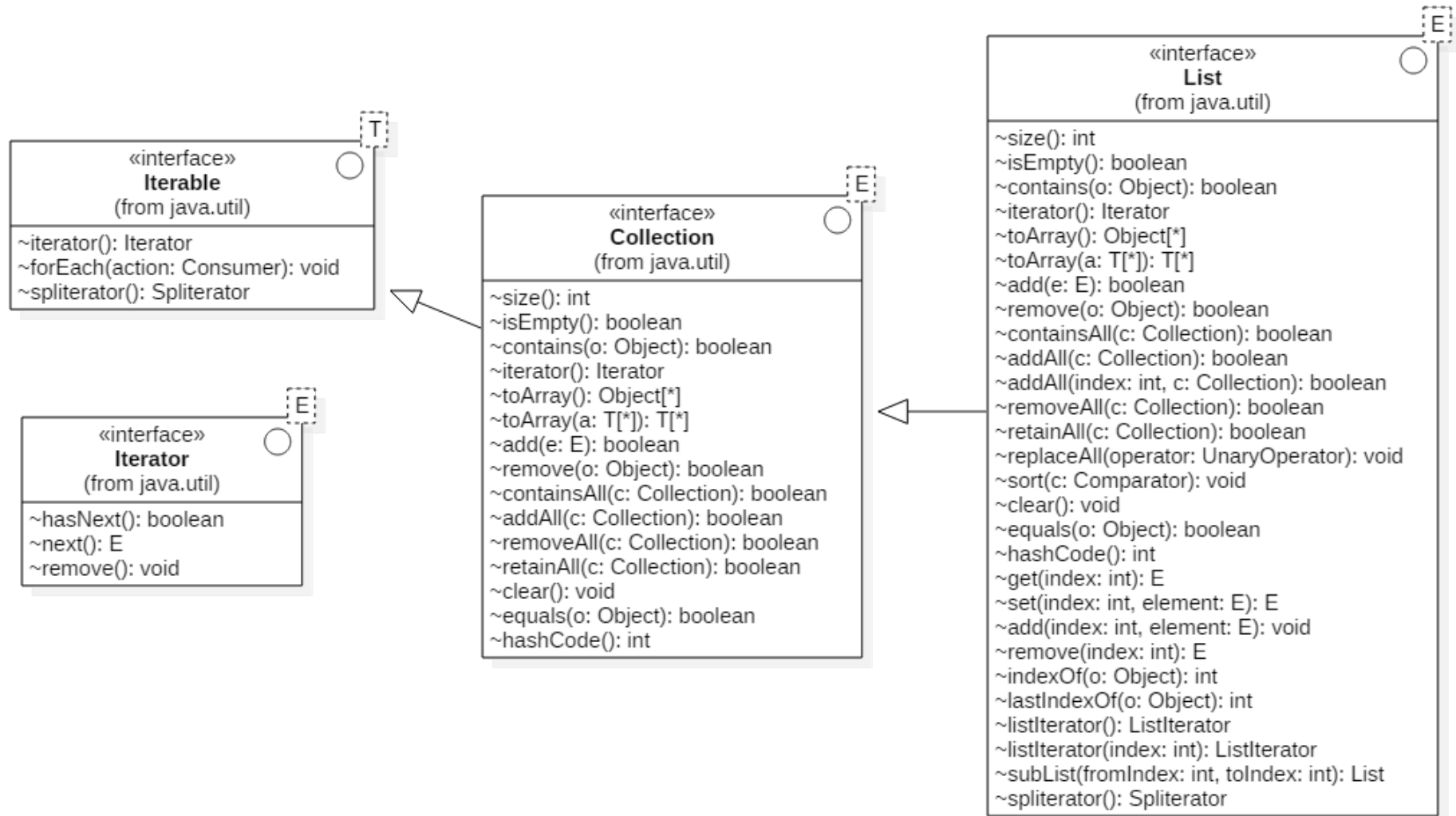


Interfaces der Java Bibliothek

- Collection<E>
- List<E>
- Set<E>
- Iterable<E>



Wichtige Interfaces





Typen in Java

Wiederholung: Typen

- Generalisierung/Spezialisierung führt zu einer Typhierarchie.
- Spezialisierung führt zu einem Subtypen.
- Jede Instanz eines Subtyps kann verwendet werden, wo auch immer eine Instanz des Supertyps erwartet wird (siehe „Behavioral Compability“ und Ersetzbarkeitsprinzip).

Typen in Java

- Klassen, abstrakte Klassen, Interfaces und Enumerations definieren Typen.
- Vererbungsbeziehung zwischen Klassen und Interfaces sind möglich.
- Achtung: Vererbung in Java **erzwingt keine** Subtypenbildung im Sinne des Ersetzbarkeitsprinzip!
- Erfüllung des Ersetzbarkeitsprinzips muss in OOP berücksichtigt werden.

Subtypen

- Bei der Erstellung von Subtypen müssen Bedingungen beachtet werden
 - Kompatible Schnittstellen
 - Kompatible Resultate
- Rückgabetypen müssen Invariant oder Kovariant sein.
- Parametertypen müssen Invariant oder Kontravariant sein.



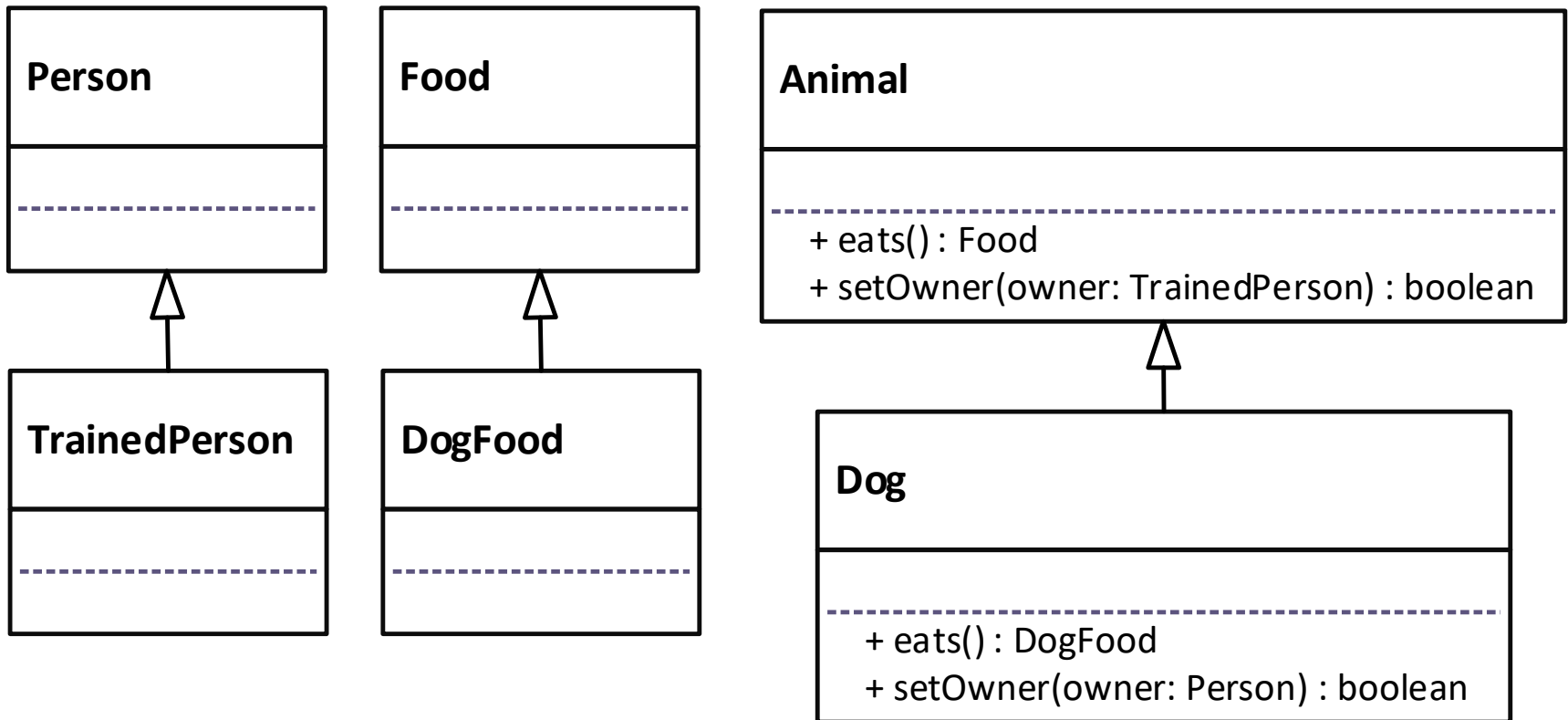
Rückgabetypen

- Bei Bildung eines Subtypen darf der Rückgabetyt einer überschriebenen Methode entweder invariant oder kovariant sein.
- Invariant bedeutet, dass keine Änderung des Rückgabetyten erfolgt.
- Kovariant bedeutet, dass der Rückgabetyt der überschriebenen Methode auf einen vom Rückgabetyt der Superklasse abgeleiteten Typen eingeschränkt ist.


Parametertypen

- Bei Bildung eines Subtypen dürfen die Parametertypen einer überladenen Methode entweder invariant oder kontravariant sein.
- Invariant bedeutet, dass keine Änderung der Parametertypen erfolgt (= Overriding).
- Kontravariant bedeutet, dass mindestens ein Parametertyp der überladenen Methode auf dessen Supertyp festgelegt wird.

Beispiel Subtyping



Beispiel Subtyping



```
public class Animal {  
    public Food eats () { return new Food(); }  
  
    public boolean setOwner (TrainedPerson owner) {...}  
}
```

kovariant

kontravariant

```
public class Dog extends Animal {  
    public DogFood eats () { return new DogFood(); }  
  
    public boolean setOwner (Person owner) {...}  
}
```

Beispiel Subtyping

```
// Invariante oder kovariante Rückgabetypen
Food food = (new Animal()).eats();
food = (new Dog()).eats();

// Invariante oder kontravariante Parametertypen
Animal animal = new Animal();
animal.setOwner( new TrainedPerson() );

animal = new Dog();
animal.setOwner( new TrainedPerson() ); // OK
animal.setOwner( new Person() ); // ERROR

Dog dog = new Dog();
dog.setOwner( new Person() ); // OK
```



Dynamisches Binden

Polymorphie und Dynamisches Binden

- Anwendung von Generalisierung/Spezialisierung führt zu einer Typhierarchie.
- Jede Instanz eines Subtyps kann verwendet werden, wo auch immer eine Instanz des Supertyps erwartet wird (siehe „*Behavioral Compability*“).

Polymorphie und Dynamisches Binden

- Variable vom Typ A können auch Instanzen von Untertypen von A beinhalten.
- Man spricht von dynamischer Bindung, wenn ein Methodenaufruf zur Laufzeit anhand des tatsächlichen (dynamischen) Typs eines Objektes aufgelöst wird.

Beispiel Polymorphie

```
public class Mammal {  
    public String info () { return „I am a mammal“; }  
}
```

```
public class Lion extends Mammal {  
    public String info () { return „I am a lion“; }  
}
```

```
public class Elephant extends Mammal {  
    public String info () { return „I am an elephant“; }  
}
```



Beispiel Polymorphie

```
// Liste von Säugetieren
List<Mammal> mammals = new ArrayList<>();

mammals.add( new Lion());
mammals.add( new Elephant());
mammals.add( new Mammal());
for ( Mammal m : mammals ){
    // Zur Laufzeit wird die Methodenimplementierung
    // anhand der Instanz gewählt.
    System.out.println( m.info() );
}
```

Console:

```
I am a lion
I am an elephant
I am a mammal
```


Beispiel Polymorphie

```
public class Person {  
    public String info () { return „I am a person“; }  
}
```

```
public class Teacher extends Person {  
    public String info () { return „I am a teacher“; }  
}
```

```
Person person = new Person();  
Teacher teacher = new Teacher();  
  
Person p1 = teacher;  
// statischer Typ = Person; dynamischer Typ = Teacher  
  
Teacher t1 = person; // FEHLER - Downcast  
// statischer Typ = Teacher; dynamischer Typ = Person
```





Casting / Typumwandlung

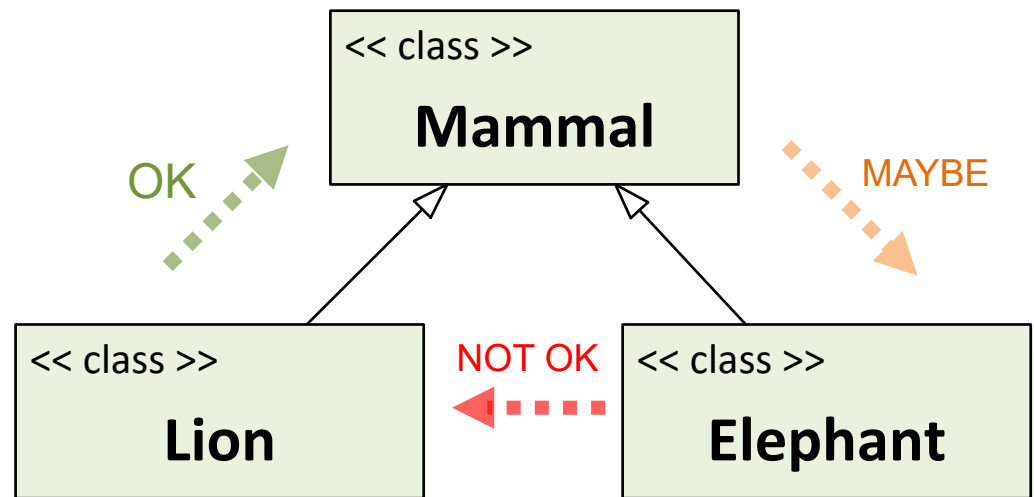
Casting

- Typumwandlung (Type-Casting)
- Implizites Casting
- Explizites Casting
 - Downcast
 - Casting Operator
 - Typesafe Casting



Typumwandlung (Type-Casting)

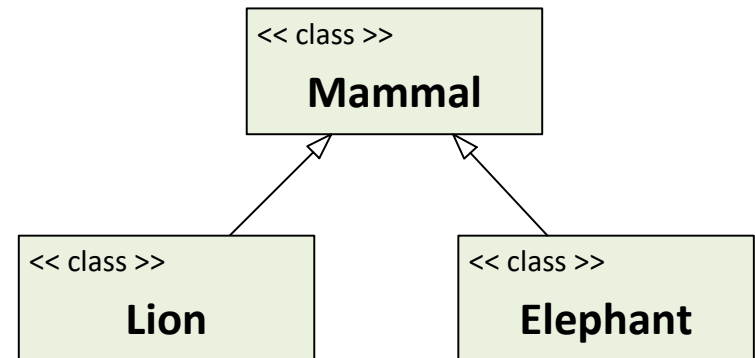
- Eine Typenhierarchie legt Obertyp – Untertyp Beziehungen fest.
- Entlang eines Astes einer Typenhierarchie können Typen ineinander umgewandelt werden.



Typhierarchie: Säugetier

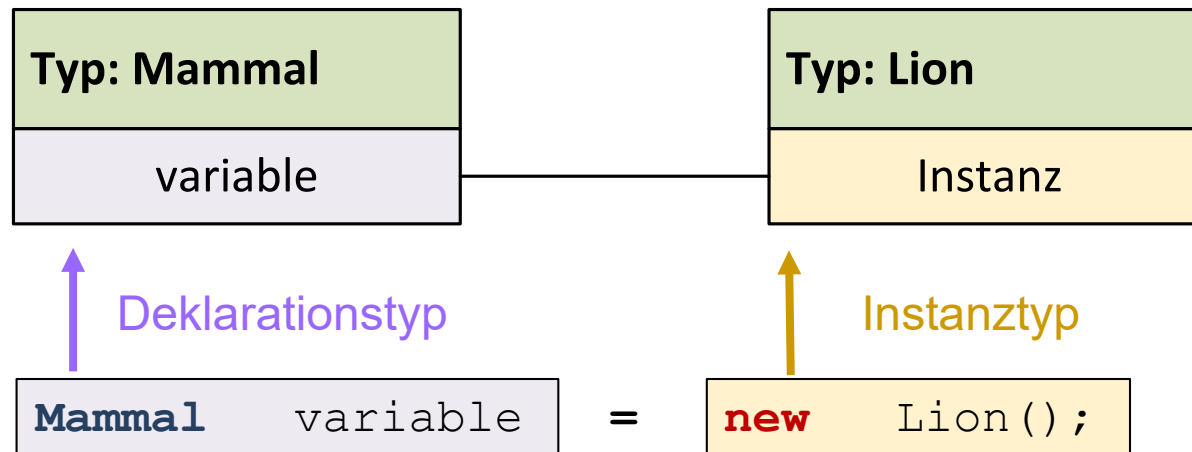
Typhierarchie - Beispiel

- Der Obertyp ist Säugetier (Mammal). Untertypen davon sind Löwe und Elefant.
- Jeder Löwe und jeder Elefant ist in dieser Typhierarchie auch ein Säugetier.
- Nicht jedes Säugetier ist auch ein Löwe bzw. ein Elefant.



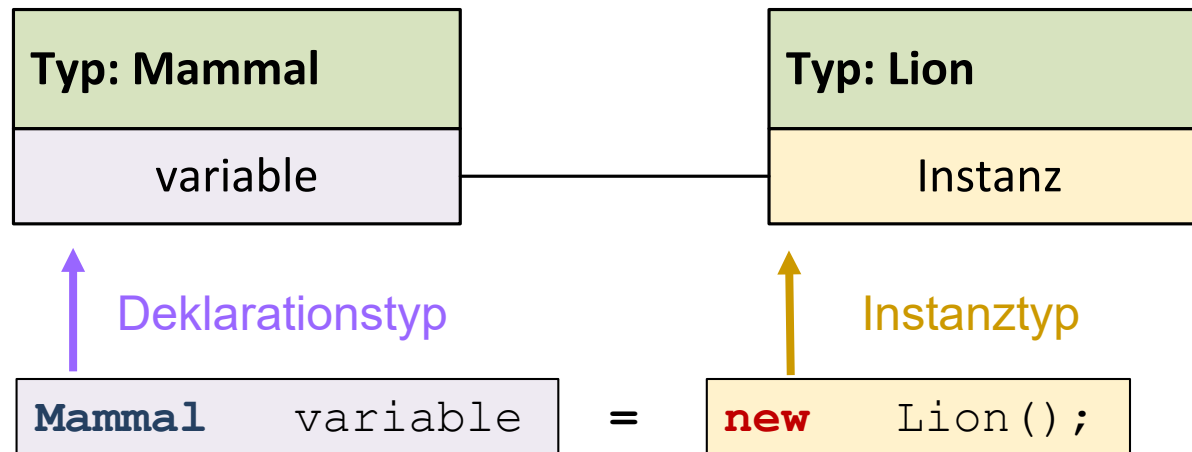
Typen in Java

- Jede Variable hat zwei Typen:
 1. Deklarationstyp (= Sicht des Compiler)
 2. Instanztyp (= Typ des konkreten Objekts zur Laufzeit)



Typen in Java

- Diese Deklarationstyp und Instanztyp können sich unterscheiden.
- Eine Einschränkung dabei ist, dass diese Typen in einer Typhierarchie stehen müssen.



Typen in Java

- Der Deklarationstyp legt fest, welche Operationen auf der Variable ausgeführt werden können (Sicht des Compiler).
- Der Instanztyp der Variable wird zur Laufzeit durch die Zuweisung eines Objekts festgelegt (new-Operator, Zuweisung, ...).
- Der Instanztyp legt zur Laufzeit fest, welche Implementierung einer Operation (Methodenaufruf auf Objekt) verwendet wird.



Beispiel Typauflösung

```
public class Mammal {  
    public String info () { return „I am a mammal“; }  
}
```

```
public class Lion extends Mammal {  
    public String info () { return „I am a lion“; }  
    public void roar () { }  
}
```

```
public class Elephant extends Mammal {  
    public String info () { return „I am an elephant“; }  
}
```



Beispiel Typauflösung

```
// Deklarationstyp: Mammal
// Instanztyp: Mammal
Mammal mammal = new Mammal ();
System.out.println( „T1:“ + mammal.info() );

// Deklarationstyp: Mammal
// Instanztyp: Lion
mammal = new Lion();
System.out.println( „T2:“ + mammal.info() );
// Der Instanztyp legt fest, welche Implementierung
// der Methode info() ausgeführt wird.
// Obwohl der Instanztyp Lion ist, kann die Methode
// roar() nicht ausgeführt werden, da der Deklarationstyp
// die möglichen Methodenaufrufe festlegt.
```

Console:

```
T1: I am a mammal
T2: I am a lion
```

Implizite Typumwandlung

- **Implizite Typumwandlung** – Die Typumwandlung ist nicht extra im Quelltext angewiesen, sondern wird automatisch vom Compiler durchgeführt.

```
// Implizite Typenumwandlung (Upcast)
Mammal mammal_1 = new Lion ();
// Explizite Typenumwandlung (Upcast)
Mammal mammal_2 = (Mammal) new Lion ();
```

Explizite Typumwandlung

- **Explizite Typumwandlung** – bei der im Quellcode die Typumwandlung ausdrücklich angewiesen wird.
- Ist der Cast nicht möglich wird eine `ClassCastException` geworfen.

```
// Explizite Typenumwandlung (Downcast)
Lion lion = (Lion) mammal_1;

// Explizite Typenumwandlung (Downcast)
Elephant elephant = (Elephant) mammal_1;
// Fehler - wird zu einer ClassCastException führen.
```

Sichere Typumwandlung (Typesafe-Casting)

- Prüfung des konkreten Instanztyps im Zuge eines potentiellen Downcasts um ClassCastExceptions zu vermeiden.
 - *instanceof*-Operator
 - *getClass*-Methode

```
Elephant elephant = null;  
if ( mammal_1 instanceof Elephant ) {  
    elephant = (Elephant) mammal_1;  
}
```

Der *instanceof*-Operator

- Dieser binäre Operator wird verwendet um zu Prüfen, ob ein Objekt zuweisungskompatibel zu einer Klasse ist.
- Er stellt zur Laufzeit fest, ob eine Referenz ungleich `null` und von einem bestimmten Typ ist. Untertypenbeziehungen werden dabei beachtet.

```
"Compare String" instanceof String ; // true
new String("OOP") instanceof String; // true
new Lion() instanceof Mammal;         // true

null instanceof Lion;                 // false
new Mammal() instanceof Lion;         // false
new Elephant() instanceof Lion;       // false
```

Die *getClass*-Methode

- Liefert ein Objekt der Klasse *Class* typisiert mit dem Instanztyps des Objekts (= Klassenobjekt).
- Das Klassenobjekt ist zur Laufzeit eindeutig.
- Wird verwendet um zu prüfen, ob zwei Objekte von derselben Klasse sind. Untertypbeziehungen werden nicht berücksichtigt.
- Diese Methode ist in der Klasse *Object* als *final* deklariert.

Die *getClass*-Methode

```
Mammal mammal = new Mammal();
Mammal lion = new Lion();
Lion lionKing = new Lion();

String myString = new String();
String testString = new String();
Class<?> clazz = myString.getClass();

if ( clazz == testString.getClass() ) {
    // is NOT executed
}
if ( mammal.getClass() == lion.getClass() ) {
    // is NOT executed
}
if ( lion.getClass() == lionKing.getClass() ) {
    // is executed
}
if ( lionKing.getClass() == Lion.class ) {
    // is executed
}
```


Typumwandlung (Type-Casting)

- Explizite Typumwandlung – bei der im Quellcode die Typumwandlung ausdrücklich angewiesen wird.
- Ist der Cast nicht möglich wird eine ClassCastException geworfen.

```
// Implizite Typenumwandlung (Upcast)
Mammal lion = new Lion ();
lion.roar(); // Fehler - Deklarationstyp hat Methode nicht

// Typsichere explizite Typenumwandlung (Downcast)
if ( lion instanceof Lion ) {
    // Ändern des Deklarationstyps durch Typecasting
    ((Lion) lion).roar(); // OK
    lion.roar(); // Fehler
}
```



Exceptions

Exceptions

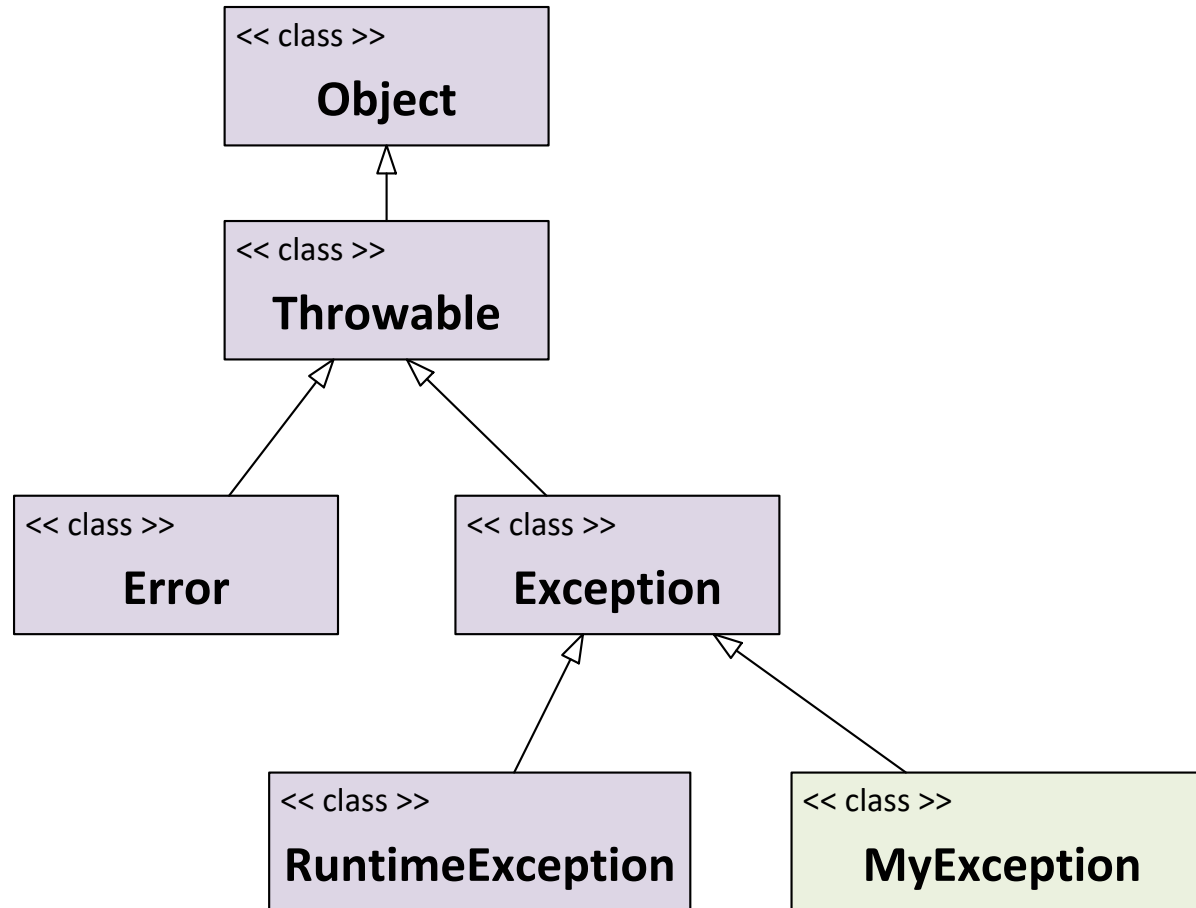
- Verwendung zur Behandlung von Laufzeitfehlern (Ausnahmebehandlung).
- Exceptions sind Objekte und werden wie diese erzeugt.
- Exceptions können in einer Klassenhierarchie stehen.
- Geworfene Exceptions unterbrechen den Programmfluss. Wird eine Exception geworfen, so wird der aktuelle Programmfluss an dieser Programmstelle abgebrochen!
- Der Programmfluss wird bei der entsprechenden Fehlerbehandlung fortgesetzt.

Exceptions

- Ausnahmebehandlung durch die aufrufende Methode.
- Eine Exception kann auch weiter gereicht werden.
- Wir betrachten nur *Checked Exceptions*.

Schlüsselwort	Verwendung
throws	Deklaration, dass eine Methode potentiell eine Exception wirft.
throw	Werfen einer Exception.
try-catch-finally	Fangen von Exceptions, Fehlerbehandlung.

Exception Hierarchie



Exceptions werfen und Fehlerbehandlung

```
public class ExceptionDemo {  
    public Double getCurrentNoXValue() throws MyException {  
        ...  
        if ( __CONDITION__ ) throw new MyException();  
        // weiterer Programmcode wird nicht ausgeführt  
        // falls die Exception geworfen wurde!  
    }  
}
```

```
ExceptionDemo demo = new ExceptionDemo();  
try {  
    demo.getCurrentNoXValue();  
    // Programmcode wenn keine Exception aufgetreten ist.  
} catch ( MyException e ) {  
    // Handle geworfene MyException .  
} finally {  
    // Optionaler finally-Block kann auch entfallen  
    // finally-Block wird immer ausgeführt  
}
```

Exceptions Beispiele

```
public class ExceptionDemo {  
  
    public Double getCurrentNoXValue() throws MyException {  
    }  
  
    public Double getCurrentTemperature() throws MyException {  
    }  
  
    public Double doCalculation() throws MyException {  
        this.getCurrentTemperature(); // may throw exception  
        this.getCurrentNoXValue();    // may throw exception  
        // do some calculation and return result  
        return ...;  
    }  
  
    public void run() {  
        try {  
            this.doCalculation();  
        }  
        catch (MyException ex) { // exception handling  
        }  
    }  
}
```

Exceptions Beispiele

```
public static returnExceptions() {  
    try {  
        System.out.println( „try block“ );  
        throw new Exception();  
    }  
    catch (Exception ex) {  
        System.out.println( „catch block“ );  
        return „return catch“;  
    }  
    finally {  
        System.out.println( „finally block“ );  
        return „return finally“;  
    }  
}  
  
// Kontrollfrage: Welches return-Statement  
// wird ausgeführt?  
System.out.println(CLAZZ.returnExceptions());
```

Console:

```
try block  
catch block  
finally block  
return finally
```


Exceptions Beispiele

```
public static returnExceptions() {  
    try {  
        System.out.println( „try block“ );  
        return „return catch“;  
    }  
    catch (Exception ex) {  
        System.out.println( „catch block“ );  
        return „return catch“;  
    }  
    finally {  
        System.out.println( „finally block“ );  
        return „return finally“;  
    }  
}  
  
// Kontrollfrage: Welches return-Statement  
// wird ausgeführt?  
System.out.println(CLAZZ.returnExceptions());
```

Console:

```
try block  
finally block  
return finally
```



Enumerations

Enumerations

- Die Enum Deklaration definiert einen Typ.
- Enumerations werden verwendet um ein fixes Set an Konstanten abzubilden.
- Instanzen einer Enum Konstante werden nur einmal erzeugt. Die Referenzen einer Enum Konstante sind daher eindeutig.

```
if (Language.GERMAN == Language.GERMAN) {  
    // Die Bedingung ist immer TRUE  
}
```

Konstruktor und Methoden

- Ein Enum kann Konstruktoren haben, diese sind aber immer *private* und können nur aus dem Enum selbst aufgerufen werden.
- Die Parameter müssen dem Konstruktor übergeben werden, wenn die Konstante erzeugt wird.
- Eine Java Enumeration kann Methoden und Attribute besitzen.

Enumeration Beispiele

```
public enum Language {  
    GERMAN,  
    ENGLISH,  
    SPANISH;  
    public boolean method (int num) {  
        if ( num < Language.values().length ) {  
            return true;  
        }  
        return false;  
    }  
}
```

```
Language lang = Language.GERMAN;  
if (lang == Language.SPANISH) {  
    // Programmcode  
    System.out.println( lang.method(4) );  
}
```

Enumeration Beispiele

```
public enum Planet {  
    // Verwendung eines Konstruktors um Werte zu setzten.  
    MERCURY (3.303e+23, 2.4397e6),  
    EARTH   (5.976e+24, 6.37814e6);  
  
    private final double mass;    // in kilograms  
    private final double radius; // in meters  
  
    Planet(double mass, double radius) {  
        this.mass = mass;  
        this.radius = radius;  
    }  
  
    public double mass() { return mass; }  
    public double radius() { return radius; }  
}
```





Die *Object*-Klasse

Die *Object*-Klasse

- Die *Object*-Klasse ist an der Spitze der Klassenhierarchie.
- Jede Klasse ist ein direkter oder indirekter „Nachkomme“ von *Object*.
- Die Methoden von *Object* können mit klassenspezifischem Verhalten überschrieben werden.

Methode	JavaDoc
<code>equals(Object obj)</code>	Indicates whether some other object is "equal to" this one.
<code>getClass()</code>	Returns the runtime class of this Object.
<code>toString()</code>	Returns a string representation of the object.
...	...

Die *equals*-Methode

- Wird die *equals*-Methode nicht überschrieben, dann ist das Ergebnis eines Vergleichs mit `o1.equals(o2)` gleich mit dem Vergleich der Referenzen der Variablen `o1 == o2`.

```
String varA = new String("Am i equal");
String varB = new String("Am i equal");

if ( varA == varB ) {
    System.out.println("References are identical");
}
if ( varA.equals(varB) ) {
    // String overrides equals
    System.out.println("Objects are equal");
}
```

Console:

Objects are equal

Die *equals*-Methode überschreiben

- Die *equals*-Methode wird bei vielen Operationen der Java-Bibliothek verwendet (z.B. `list.contains(..)`).
- Überschreiben der *equals*-Methode wenn ein logischer Vergleich sinnvoller ist, d.h. ein Vergleich der Attribute erwünscht ist.
- Einhaltung des *equals*-Anforderungen im Zusammenhang mit Vererbung ist mit Herausforderungen verbunden!

“It turns out that this is a fundamental problem of equivalence relations in object-oriented languages. There is no way to extend an instantiable class and add a value component while preserving the equals contract, unless you are willing to forgo the benefits of the object-oriented abstraction.”

[Effective Java 2nd Edition, Joshua Bloch, Item 8, p38]

Anforderungen an die *equals*-Methode

- Wird Sie überschrieben muss die *equals*-Methode eine binäre, reflexive, symmetrische, transitive Operation (Äquivalenzrelation R) sein.

1. Reflexivität: $X \sim X$

2. Symmetrie: $X \sim Y \Rightarrow Y \sim X$

3. Transitivität: $X \sim Y \wedge Y \sim Z \Rightarrow X \sim Z$

Äquivalenzklasse:

Die Äquivalenzklasse eines Objektes a ist die Klasse der Objekte, die äquivalent zu a sind.

$$[a]_R = \{x \in M \mid x \sim_R a\} \subseteq M \quad R \subseteq M \times M$$

Die Klasse Point

```
public class Point {  
    public int x;  
    public int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        // Different implementations ...  
    }  
}
```



Die Klasse ColorPoint

```
public class ColorPoint extends Point {  
    // the color is defined by an enum  
    public Color color;  
  
    public ColorPoint(int x, int y) {  
        super(x,y);  
        this.color = color;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        // Different implementations ...  
    }  
}
```



Codegerüst mit *instanceof*-Operator

```
@Override
public boolean equals(Object obj) {

    if ( obj == null )
        return false;

    if ( obj == this )
        return true;

    if ( obj instanceof _CLASSNAME_ ) {
        if ( _CONDITION_ ) {
            // Evaluate whether the objects (this,obj)
            // are equal
            return true;
        }
    }
    return false;
}
```

Beispiel mit *instanceof*-Operator

```
// Implementation of class Point
public boolean equals(Object obj) {
    // Previous code - see code skeleton

    if ( obj instanceof Point ) {
        Point point = (Point) obj;
        return this.x == obj.x && this.y == point.y ;
    }
    return false;
}
```

```
// Implementation of class ColorPoint
public boolean equals(Object obj) {
    // Previous code - see code skeleton

    if ( obj instanceof ColorPoint ) {
        ColorPoint point = (ColorPoint) obj;
        return super.equals() && this.color == point.color ;
    }
    return false;
}
```

Beispiel mit *instanceof*-Operator

- Diese Implementierung verletzt die Symmetrie-Eigenschaft!
- Diskutieren Sie die Ursache des Problems und folgende Aussage:

Würde die *equals*-Methode von `ColorPoint` eine Fallunterscheidung machen, und für ein Vergleichsobjekt vom Typ `Point` den Vergleich entsprechend der Implementierung von `Point` machen, verletzen Sie die Transitivität.

```
Point point = new Point(1,2);
ColorPoint colorPoint = new ColorPoint(1,2, Color.RED);

// will return true
point.equals( colorPoint );

// will return false
colorPoint.equals( point );
```

} Verletzung der Symmetrie!

Codegerüst mit *getClass-Methode*

```
@Override
public boolean equals(Object obj) {

    if ( obj == null )
        return false;

    if ( obj == this )
        return true;

    if ( obj.getClass() == this.getClass() ) {
        if ( __CONDITION__ ) {
            // Evaluate if the objects (this,obj) are equal
            return true;
        }
    }
    return false;
}
```

Beispiel mit *getClass*-Methode



```
// Implementation of class Point
public boolean equals(Object obj) {
    // Previous code - see code skeleton

    if ( this.getClass() == obj.getClass() ) {
        Point point = (Point) obj;
        return this.x == obj.x && this.y == point.y ;
    }
    return false;
}
```

```
// Implementation of class ColorPoint
public boolean equals(Object obj) {
    // Previous code - see code skeleton

    if ( this.getClass() == obj.getClass() ) {
        ColorPoint point = (ColorPoint) obj;
        return super.equals() && this.color == point.color ;
    }
    return false;
}
```

Beispiel mit *getClass*-Methode

- Diese Implementierung erfüllt die Anforderungen an eine Äquivalenzrelation.
- Verletzt das Liskovsche Substitutionsprinzip (Ersetzbarkeitsprinzip)

Vergleich nur zwischen gleichen Klassen – Es wird kein Vergleich mit einer Subklassen ausgeführt.

```
Point point = new Point(1,2);
ColorPoint colorPoint = new ColorPoint(1,2, Color.RED);

// will return false, but based on the substitution
// principle one would expect true.
point.equals( colorPoint );

// will return false
colorPoint.equals( point );
```

Verletzung des
Ersetzbarkeitsprinzips!

Zusammenfassung: *equals*-Methode

- Die Lösung muss abhängig vom konkreten Anwendungsfall wohlüberlegt sein, beide Lösungen bieten Vor-/Nachteile.
- Verletzung der Anforderungen einer Äquivalenzrelation kann zu schwer auffindbaren Problemen führen.
- Aufbrechen der Ersetzbarkeit kann zum Verlust der Vorteile der Typenersetzbarkeit führen.
- **Empfehlung:** Versuchen Sie die Äquivalenzrelation und die Ersetzbarkeit einzuhalten und, wenn nicht anders möglich, verzichten Sie auf die Ersetzbarkeit im Bezug auf die *equals*-Methode. Seien Sie sich der Implikationen bewusst und dokumentieren Sie diese im Source-Code und der JavaDoc.

Die *toString*-Methode

- Der zurückgegebene String soll eine prägnante, leicht zu erfassende, Darstellung aller wichtigen Informationen über ein Objekt sein.
- Verwendung von `toString` für z. B. Diagnoseausgaben oder im Debugger.
- Auf Informationen, die in der Rückgabe von `toString` enthalten sind, soll von Außen zugegriffen werden können.
- **Empfehlung:** Keine manuellen Zeilenumbrüche in der Rückgabe von `toString` einfügen.

Beispiel einer toString-Methode

```
public class Point {  
    public int x;  
    public int y;  
  
    public Point(int x, int y) { ... }  
  
    public String toString() {  
        return "Point [x=" + x + ", y=" + y + "]";  
    }  
}
```

Console:

```
-- toString of Objects  
Point [x=10, y=30]  
ColorPoint [x=2, y=7, color=RED]  
  
-- toString of a List  
[Point [x=10, y=30], ColorPoint [x=2, y=7, color=RED]]
```

Bücher und Web-Links

Bücher:

Java ist auch eine Insel, Galileo Press

ISBN 978-3-8362-1802-3

Programmieren in Java, Pearson Studium; Auflage: 2 (1. August 2010)

ISBN: 978-3868940312

Effective Java (2nd Edition): A Programming Language Guide

ISBN: 978-0321356680

Weblinks:

<https://docs.oracle.com/javase/tutorial/java/index.html>

<http://openbook.rheinwerk-verlag.de/javainsel>

Kapitel 1 bis 7

