# Vorkurs Programmieren

HTW Berlin
SoSe 2016

# 1.) Wiederholung

## 1.) Daten- und Referenztypen

# Arrays

- Gruppiert identische Datentypen
- Lesen / Schreiben mit Index (Start: 0)
- Feste Anzahl Elemente / Länge

```
public class Main {
    public static void main(String[] args) {
        int[] coordinate = new int[] {0,3,1};
//          int[] point = new int[3];
//          point[0] =  3; ...
        printCoordinates(coordinate);
    }

    private static void printCoordinates(int[] p) {
       System.out.println("Coordinates:"+p[0]+","+p[1]+","+p[2]);
    }
}
```

# von primitiven Datentypen zu Referenztypen

- primitive Datentypen
  - speichern Wert direkt vor Ort (call-by-value)
  - i.d.R. nur so groß wie ein Speicherbereich (heute: 64 bit)
- Referenztypen:
  - speichern Referenz auf eigentlichen Werte (call-by-reference)
  - siehe: Array und ArrayList

# Datentypen: Zusammenfassung

- werden mit **`new`** erzeugt
- keine Referenz: **`null`**
- wenn keine Referenz mehr im System: Garbage Collection

- Datenstrukturen und andere Referenztypen nutzen wie **`String`**, **`BigDecimal`** und **`ArrayList`**
- primitive Datentypen wie **`int`** nutzen mittels Wrapper-Klassen wie **`Integer`** bei generischen Datentypen

# Generische Typen

```
public class Main {
    public static void main(String[] args) {
        int[] pointA = new int[3];
        pointA[0] =  3;

        ArrayList<Integer> pointAL = new ArrayList<Integer>();
        pointAL.add("3");
        int x = pointAL.get(0);

    }
}
```

1.) Wiederholung

2.) Klassen

# Klassen und Objekte

- Klassen (abstrakt, allgemein) und
  Objekte (Instanz einer Klasse, konkret)

- Bekannt aus: Java API (Bsp.: ArrayLists)
  Referenz auf Inhalt

- Organisation in Packages

- Modifier wie `public`, `private`, `protected` und `getters()`/`setters()`

- Vererbung mit Ober- und Unterklasse,
  Methoden überschreiben

# Inhalt Vorlesung 3

1. ~~Wiederholung~~
2. Interfaces
3. Fehlerbehandlung
4. Design Patterns

# 2.) Interfaces

# Objekte und Eigenschaften

- Zusammenfassend:
  - Jedes Objekt hat eine Identität
  - Jedes Objekt hat einen Zustand
  - Jedes Objekt zeigt ein Verhalten

- Problem: Wie lässt sich das standardisieren?

http://openbook.rheinwerk-verlag.de/javainsel/javainsel_03_001.html#dodtp394ff2f8-9134-4eb3-a6b4-469ed4572a74

# Objekte und Eigenschaften

- Vererbung: Einfach-Klassenvererbung (daher beschränkt)
- Idee: Mehrfach Schnittstellenvererbung
- Frage: Was kennzeichnet eine Schnittstelle (Interface)?

- Bsp.:

```
class Car extends Vehicle implements Lockable, Refillable

Lockable lockableCar = (Lockable) myCar;
```

# Sinn und Zweck

- Interfaces: Neben Vererbung (via [`abstract`] `Class`)
  - Trennen von *“Was ist implementiert”* und *“Wie ist es Implementiert”*.
  - *Meist* Eigenschaften eines Objektes nach außen z.B.: `Resizable`, `Comparable`, `Sortable`
  - Eine Art Schablone: Interfaces verdecken alle Methoden, außer für die Eigenschaft notwendigen

# Interface-Deklaration

```
public interface Lockable {
    boolean isLocked();

    void registerKey(String key);
    void lock();
    void unlock(String key);
}
```

- Implementierung abhängig von implementierender Klasse

# Interface-Implementierung

```
class Car extends Vehicle implements Lockable, Refillable {
    private int numberOfDoors;
    public Car(String color, int numberOfDoors) {
       // ...
    }
    public boolean isFamilyFriendly() {
        return numberOfDoors > 3;
    }

    public boolean isLocked() {
        return false;
    }
    public void registerKey(String key) {}

    public void lock() {}

    public void unlock(String key) {}
}
```

# Interface-Implementierung

```
private String key;
private boolean isLocked;

public boolean isLocked() {
    return isLocked;
}

public void registerKey(String key) {
    if(key != null) {
        return;
    }
    this.key = key;
    isLocked = false;
}

public void lock() {
    isLocked = true;
}

public void unlock(String key) {
    if(this.key != null && this.key.equals(key)) {
        isLocked = false;
    }
}
```

# Interfaces: Zusammenfassung

- Legt Eigenschaften nach außen fest
- Mehrere Eigenschaften via Interface implementierbar
- Trennt Implementierung von Deklaration
- Warum Einfach-Klassenvererbung, aber Mehrfach-Schnittstellenvererbung?

# 3.) Fehlerbehandlung

# Bsp.: ungültiger Zugriff

```java
public class ErrorHandling {
    public static void main(String[] args) {
        int[] point = {3, 1};
        point[3] = 2;
    }
}
```

Ausgabe:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 3
  at de.htw.ErrorHandling.main(ErrorHandling.java:6)
```

# Bsp.: Fehler auffangen

```
public class ErrorHandling {
    public static void main(String[] args) {
        int[] point = {3, 1};

        try {
            point[3] = 2;
            System.out.println("Punkt in 3D");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Nicht möglich");
        }
    }
}
```

# Was sind Fehler (Exceptions)?

- Neben **`Return`** eine Möglichkeit eine Methode zu verlassen.

- Zeigt undefinierten Zustand an -
  Programm weiß nicht, wie es weiter geht:
  *Vollbremsung* bis nach außen oder aufgefangen

  - `NullPointerException`
  - `IndexOutOfBoundException`
  - `IllegalStateException`
  - `IllegalArgumentException`

# Exception-Deklaration bei Methoden

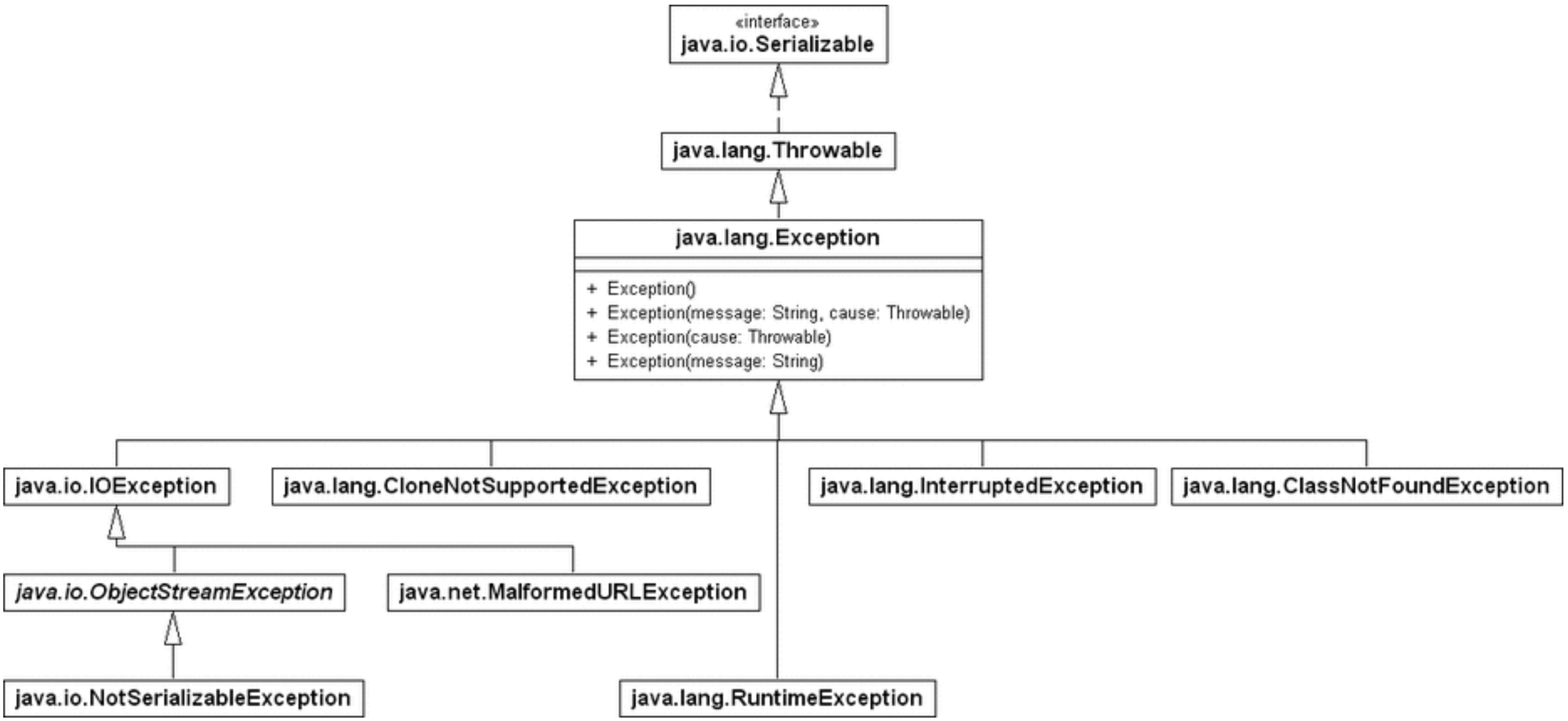
```
// Integer.java
public static int parseInt(String s)
  throws NumberFormatException {
    return parseInt(s,10);
}

// ErrorHandling.java
public class ErrorHandling {
    public static void main(String[] args) {
        int vatRate;
        try {
            vatRate = Integer.parseInt("19");
        } catch(NumberFormatException e) {
            vatRate = 0;
        }
    }
}
```

# Arten von Exceptions

- Zur Kompilierzeit bekannt:
  - geprüfte (checked) Exceptions, müssen deklariert werden (via `throws`)
  - abgeleitet von `Throwable`
  - bekannt, dass diese geworfen werden können
  - z.B.: `IOException`
- Zur Laufzeit
  - ungeprüfte (unchecked) Exception, können deklariert werden (via `throws`)
  - abgeleitet von `RuntimeException`
  - abhängig vom Zustand, sollten i.d.R. nicht auftreten
  - z.B.: `NumberFormatException`

# Arten von Exceptions



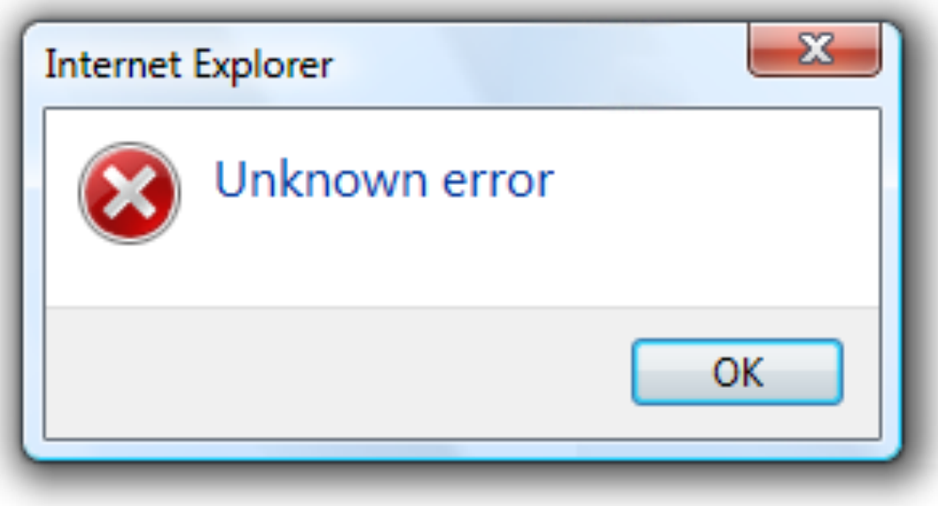

http://openbook.rheinwerk-verlag.de/javainsel9/bilder/exceptionsuml.gif

# Eigene Exceptions

- Ableiten von `RuntimeException`, `Exception`, oder `Throwable`

- z.B. für ungültigen Zustand bei eigenen Programm-Modulen

- Hilft beim Fehler finden und beheben



# try-catch: Alle Fehler

```
// Integer.java
public static int parseInt(String s)
  throws NumberFormatException {
    return parseInt(s,10);
}

// ErrorHandling.java
public class ErrorHandling {
    public static void main(String[] args) {
        int vatRate;
        try {
            vatRate = Integer.parseInt("19");
        } catch(Exception e) {
            vatRate = 0;
        }
    }
}
```

# try-catch: mehrere Fehler

```
// Integer.java
public static int parseInt(String s)
  throws NumberFormatException {
    return parseInt(s,10);
}

// ErrorHandling.java
public class ErrorHandling {
    public static void main(String[] args) {
        int vatRate;
        try {
            vatRate = Integer.parseInt("19");
        } catch(NullPointerException | NumberFormatException e) {
            vatRate = 0;
        }
    }
}
```

# finally: Abschlussbehandlung

```
// Integer.java
public static int parseInt(String s)
  throws NumberFormatException {
    return parseInt(s,10);
}

// ErrorHandling.java
public class ErrorHandling {
    public static void main(String[] args) {
        int vatRate;
        try {
            vatRate = Integer.parseInt("19");
        } catch(NullPointerException | NumberFormatException e) {
            // TODO: try to parse in a different way?
            vatRate = 0;
        } finally {
            System.out.println("MwSt ist" + vatRate);
        }
    }
}
```

# Fehlerbehandlung: Zusammenfassung

- Stoppt/Verlässt aktuellen Programmcode sofort (vgl. `Return`)
- Undefinierten Zustand beheben
- `try-catch-finally` möglichst genau
- Eigene Fehler erzeugen bei eigenen Modulen *oder* klare Fehlermeldungen erzeugen

# 4.) Design Patterns

# Literatur

- Titel: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*
  (Originaltitel: *Design Patterns. Elements of Reusable Object-Oriented Software*)

- 4 Autoren: Viererbande (engl.: Gang of Four, GoF)

- Klassifizierung von Design Patterns

# Was sind Design Patterns?

- Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software

- Nicht nur wichtig bei OOP

  - Bsp.: definierter Such-Algorithmus

- *Design* Pattern

  - Muster Zusammenarbeiten mehrerer Klassen

  - *Software-Architektur-Design*

# Eigenschaften

- Erprobt
  - auf Fehler und Nebeneffekte
  - auf Nutzen
- Bekannt
  - Entwickler verstehen Code schneller
- Effizienz
  - sparen Zeit
  - Code lässt sich leichter wiederverwenden

# Abgrenzung

- Nur Vorlage (Klassen- / Objektnamen, Verwendung)
- Menschenverstand nutzen
- Anti-Pattern: Muster von schlechtem Code

- Bsp.: `“Hello” == “Hello”` vs.
  `“Hello”.equals(“Hello”)`

- Tipp: neues Feature entdeckt?
  > Google nach *Patterns / Anti-Patterns*

# Bsp.: Factory (Fabrik)

- Idee:
  - Fabrik-Klasse kennt Implementierungen
  - gibt auf Anfrage gewünschtes Objekt zur weiteren Verwendung
- Grund:
  - Trennung von Was und Wie
  - Implementierung interessiert uns nicht

# Factory: gemeinsame Schnittstelle

```java
// SortAlgorithm.java
public interface SortAlgorithm {
    void  doSort(int[] data);

    String getName();
}
```

```java
// MyDemoSort.java
public class MyDemoSort implements SortAlgorithm {

    public void doSort(int[] data) {
        Arrays.sort(data);
    }

    public String getName() {
        return "demo";
    }
}
```

# Factory: Umsetzung

```
// SortFactory.java
public class SortFactory {

    static SortAlgorithm[] algos = new SortAlgorithm[] {
            new MyDemoSort()
    };

    static SortAlgorithm getAlgorithm(String name) {
        for (SortAlgorithm a : algos) {
            if(a.getName().equals(name)) {
                return a;
            }
        }

        throw new IllegalArgumentException("Algorithm with name "
            + name + " not found.");
    }
}
```

# Factory: Verwendung

```
// MyProgram.java
public class MyProgram {

    public static void main(String[] args) {
        SortAlgorithm demo = SortFactory.getAlgorithm("demo");

        int[] data = {7, 4, 2, 6, 3, 2, 6, 8, 1};

        System.out.println(Arrays.toString(data));

        demo.doSort(data);

        System.out.println(Arrays.toString(data));
    }
}
```

# Bsp. aus der Java-API

- `javax.xml.parsers.DocumentBuilderFactory`
- `javax.xml.transform.TransformerFactory`
- `javax.xml.xpath.XPathFactory`

https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries/2707195#2707195

# Zweck: Factory (Fabrik)

- Idee:
  - Fabrik-Klasse kennt Implementierungen
  - gibt auf Anfrage gewünschtes Objekt zur weiteren Verwendung
    - *Bsp.: Dokumenten-Erstellung, Datei-Einlesen*
- Grund:
  - Trennung von Was und Wie
  - Implementierung interessiert uns nicht

# Patterns: Zusammenfassung

- Nutzen von Erfahrung bei Problemlösung und Bereitstellen von besser lesbarem Code

- Immer eine Suche wert ;-)

- Anti-Patterns zeigen wie es nicht gemacht wird
  > Lernen aus Fehlern (von anderen)