

Übungsstunde 10

Einführung in die Programmierung

Probleme bei Übung 8
Keine, Oder?

Zur Bonusaufgabe diese Woche

Mittwoch

17:00 - 19:00

Wird zum ersten Mal so gemacht → sie hoffen, dass alles funktioniert

“Das Zeitfenster ist 17:00 -- 19:00 am Mittwoch, den 30. 11. (bzw. 7. 12.). Wir werden das Template (am Mittwoch vor 17:00) in Ihr git Repository hochladen und dann um 17:00 die Aufgabenstellung auf dem Web publizieren. Bis auf die Deadline gelten die ueblichen Regeln fuer die Bonusaufgaben.”

Advent of Code

<https://adventofcode.com/>

Rekursion und einfache Liste

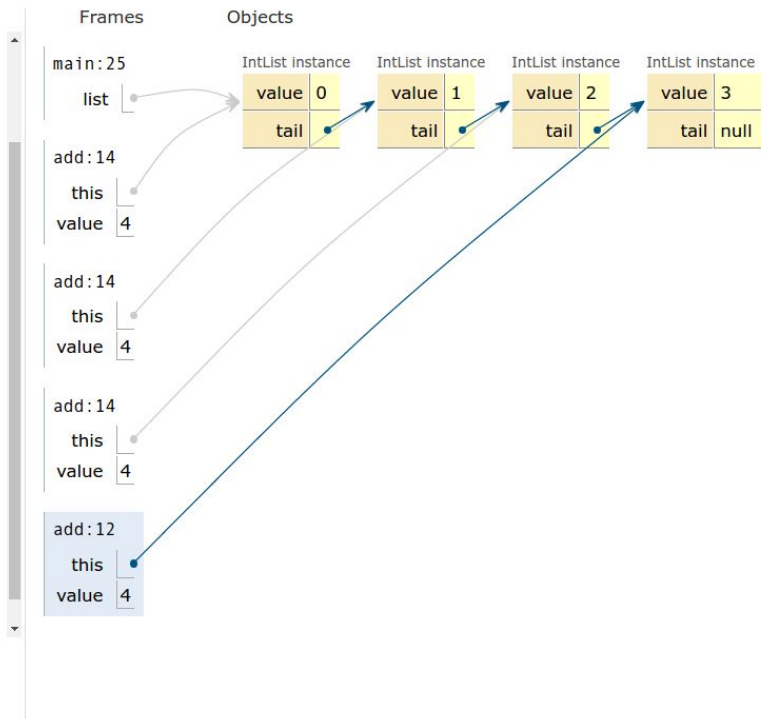
- [Visualisierung](#) (Link)

```
5
6      public IntList(int value) {
7          this.value = value;
8      }
9
10     public void add(int value) {
11         if (tail == null)
12             tail = new IntList(value);
13         else
14             tail.add(value);
15     }
16 }
17
18
19 public class Main {
20     public static void main(String[] args) {
21         IntList list = new IntList(0);
22         list.add(1);
23         list.add(2);
24         list.add(3);
25         list.add(4);
26     }
```

[Edit code](#)

→ line that has just executed

→ next line to execute



Nachbesprechung Übung 9

Was ihr besprechen wolltet

Schwierige Aufgaben: 2 & 3

Bonusaufgabe: <1h

Besprechung: 2 & 4

Aufgabe 1: Bonusaufgabe (Bonus!)

Feedback nach der Korrektur direkt per Git

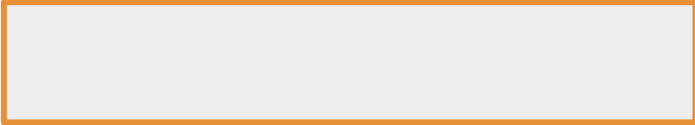
Aufgabe 2: Umkehrung

```
public void reverse() {  
    IntNode previousNode = null;  
    IntNode currentNode = this.first;  
    IntNode nextNode = null;  
  
    while(currentNode != null) {  
        nextNode = currentNode.next;  
        currentNode.next = previousNode;  
  
        // Aktualisiere previousNode und currentNode  
        previousNode = currentNode;  
        currentNode = nextNode;  
    }  
  
    ...  
}
```

Aufgabe 2: Umkehrung

```
public void reverse() {  
    IntNode previousNode = null;  
    IntNode currentNode = this.first;  
    IntNode nextNode = null;  
  
    while(currentNode != null) {  
        nextNode = currentNode.next;  
        currentNode.next = previousNode;  
  
        // Aktualisiere previousNode und currentNode  
        previousNode = currentNode;  
        currentNode = nextNode;  
    }  
  
    //Setze first und last durch einen Swap  
    IntNode newLast = this.first;  
    this.first = this.last;  
    this.last = newLast;  
}
```

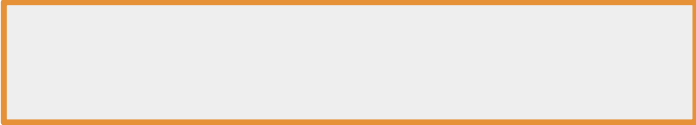
Aufgabe 3: Miles and More

```
public static void calculateMiles(File input, File output) throws FileNotFoundException {  
    Scanner in = new Scanner(input);  
    PrintStream out = new PrintStream(output);  
  
    while (in.hasNext()) {  
        String fullName = in.nextLine();  
  
          
    }  
  
    in.close();  
    out.close();  
}
```

Aufgabe 3: Miles and More

```
public static void calculateMiles(File input, File output)
{
    Scanner in = new Scanner(input);
    PrintStream out = new PrintStream(output);

    while (in.hasNext()) {
        String fullName = in.nextLine();

    }

    in.close();
    out.close();
}
```

```
// now the list of flights starts
int totalMiles = 0;
String flightNum = in.next();
while (!flightNum.equals(".")) {
    String date = in.next();
    String category = in.next();

    int miles = 125;
    if (in.hasNextInt()) {
        miles = in.nextInt();
    }

    if (category.equals("FIRST")) {
        miles = miles * 3;
    } else if (category.equals("BUSINESS")) {
        miles = miles * 2;
    } else {
        assert category.equals("ECONOMY");
    }

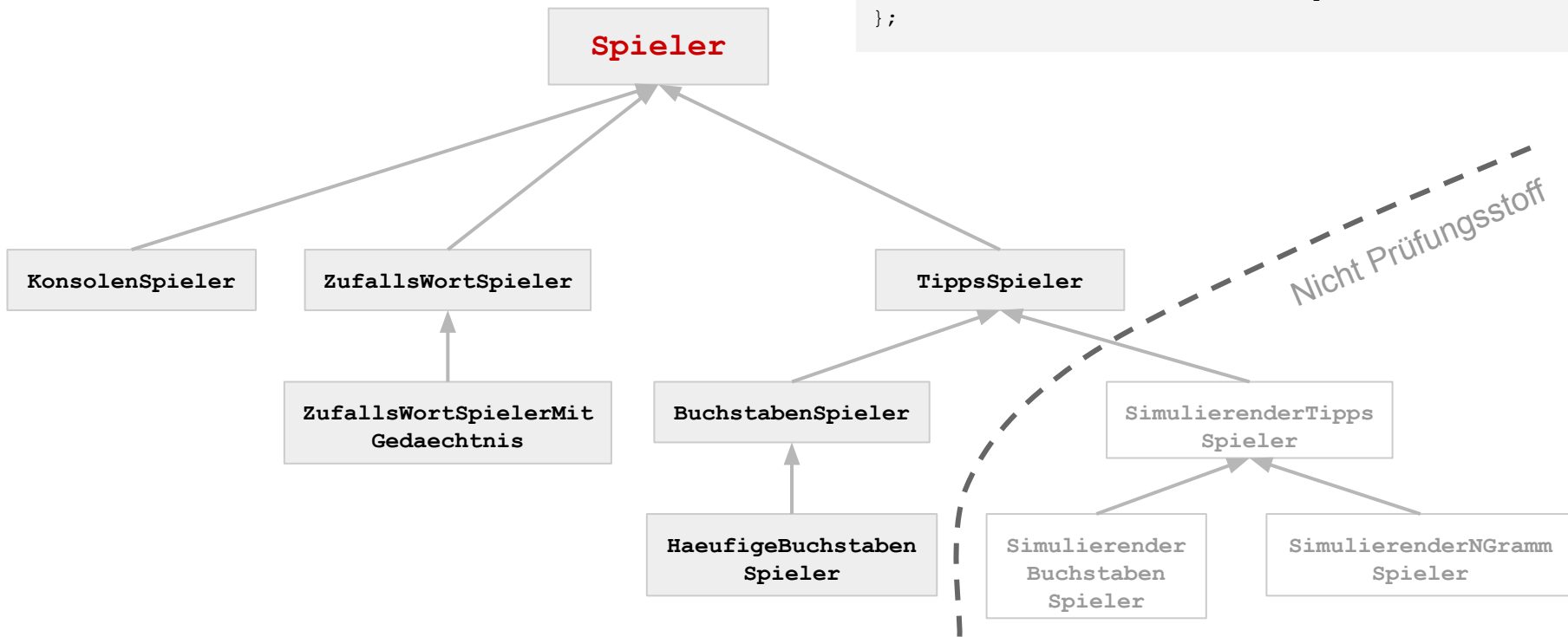
    totalMiles += miles;

    flightNum = in.next();
}

in.nextLine();
out.println(fullName + ": " + totalMiles);
```

Aufgabe 4: Ratespiel

```
Spieler[] spieler = {  
    new ZufallsWortSpieler(),  
    new ZufallsWortSpielerMitGedaechtnis(),  
    new BuchstabenSpieler(),  
    new HaeufigeBuchstabenSpieler(),  
    new SimulierenderBuchstabenSpieler(),  
    new SimulierenderNGrammSpieler()  
};
```



SimulierenderTippsSpieler

Nach einem Tipp alle Wörter entfernen, die nicht mehr möglich sind.

In jeder Runde berechnen: Welcher Tipp ist am wertvollsten?

=> die Anzahl noch möglicher Wörter am meisten

Tipp nicht nur a,b,c... sondern auch N-Gramme, also au, nn, ch...

Beispiel BuchstabenSpieler

```
public class BuchstabenSpieler extends Spieler {
    String[] buchstaben = { "a", ..., "z" };
    String[] woerter;
    boolean[] aussortiert;
    boolean[] tippVerwendet;

    public String gibTipp() {
        int uebrigeW = zaehleFalse(aussortiert);
        int uebrigeT = zaehleFalse(tippVerwendet);

        if (uebrigeW > 2 && uebrigeT > 0) {
            // verwende einen Tipp, den wir noch nicht probiert haben
        } else {
            // probiere eins der noch übriggebliebenen Wörter
        }
    }

    public void bekommeHinweis(String tipp, String hinweis) {
        // setze aussortiert[i] auf 'true', falls das i-te Wort gemäss
        // erhaltenem Hinweis nicht mehr möglich ist.
    }
}
```

- Dieser Spieler probiert zuerst alle Buchstaben a-z (äöü) durch (Tipps)
 - Wenn ≤ 2 Wörter übrig, oder alle Buchstaben ausprobiert: probiere die restlichen Wörter.
- bekommeHinweis:
 - sortiert alle Wörter aus, die gemäss Hinweis nicht mehr möglich sind

Beispiel BuchstabenSpieler

```
public class BuchstabenSpieler extends TippsSpieler {  
    protected String[] buchstaben = {"a", ..., "z"};  
  
    protected String[] tipps(String[] woerter) {  
        return buchstaben;  
    }  
  
    protected int tippIndex() {  
        for(int i = 0; i < tipps.length; i++)  
            if(!tippVerwendet[i])  
                return i;  
    }  
}
```

Die Musterlösung extrahiert einen Teil der Funktionalität in eine Superklasse **TippsSpieler**, die auch von anderen Spielern verwendet wird

Aufgabe 5: Pong

In dieser Aufgabe geht es darum, den Klassiker [Pong](#) von 1972 nach zu bauen. Es sollen zwei Spieler gegeneinander spielen können. Dafür kommt die bereits bekannte Window-Klasse zum Zug. Sie bietet einige Methoden, um Input von Maus und Tastatur aufzunehmen. Eine davon ist `isKeyPressed(String keyName)`, welche `true` zurück gibt, wenn die spezifizierte Taste zum Zeitpunkt des Aufrufs gedrückt ist:

```
if(window.isKeyPressed("up")) {  
    // move something around  
}
```

Das Spiel besteht aus einem Ball und aus zwei Spielern, welche je einen vertikalen Balken kontrollieren und versuchen, den Ball im Spiel zu halten. Wenn der Ball das Spiel seitlich verlässt, erhält der gegenüberliegende Spieler einen Punkt. Wenn der Ball hingegen die Wände oben und unten oder einen Spielerbalken berührt, prallt er ab.



Alle Bälle
verschieben und auf
Kollision testen

PongGame

Spiel-Logik

Methoden

- `move()`
- **`step()`**
- `collides()`

Objekte

- `Players: p1, p2`
- **`LinkedList`**

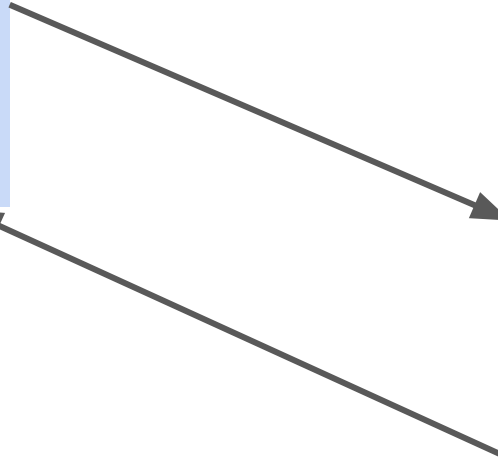
PongGui

Zeichnet

- Spielfeld
- Spieler (Balken, Punktestand)
- **Bälle**

Reagiert auf Tastatur

- up/down, w/s



Theorie

Nur zur Info

Die Themen ab jetzt sind garantiert prüfungsrelevant

→ mit der Theorie müsst damit umgehen können (programmieren)




→ ihr müsst üben

Polymorphism

Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method.

private < (default) < protected < public

<pre>class Tier { public void name() { } } class Hund extends Tier { public void name() { } }</pre> 	<pre>class Tier { private void name() { } } class Hund extends Tier { public void name() { } }</pre> 	<pre>class Tier { public void name() { } } class Hund extends Tier { private void name() { } }</pre> 
--	--	---

Same level

child allows more

child is more restrict

```
class Tier {  
    public void name() {  
        System.out.print("name: ")  
        foo();  
    }  
    public void foo() {  
        System.out.print("Tier");  
    }  
}
```

```
class Hund extends Tier {  
    public void foo() {  
        System.out.print("Hund");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Tier tier = new Tier();  
        Tier tierHund = new Hund();  
        Hund hund = new Hund();  
  
        tier.name(); // name: Tier  
        tierHund.name(); // name: Hund  
        hund.name(); // name: Hund  
    }  
}
```

```
class Tier {  
    int x = 5;  
    public void alter() {  
        System.out.println("alter: " + x);  
    }  
}
```

```
class Hund extends Tier {  
    int x = 10;  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Tier tier = new Tier();  
        Tier tierHund = new Hund();  
        Hund hund = new Hund();  
  
        tier.alter();    // alter: 5  
        tierHund.alter(); // alter: 5  
        hund.alter();   // alter: 5  
    }  
}
```

```
class Tier {
    int x = 5;
    public void alter() {
        System.out.println("alter: " + x);
    }
}

class Hund extends Tier {
    int x = 10;
    public void alter() {
        System.out.println("alter: " + x);
    }
}

public class Main {
    public static void main(String[] args) {
        Tier tier = new Tier();
        Tier tierHund = new Hund();
        Hund hund = new Hund();

        tier.alter();    // alter: 5
        tierHund.alter(); // alter: 10
        hund.alter();   // alter: 10
    }
}
```



```
class Tier {
    int x = 5;
}

class Hund extends Tier {
    double x = 10;
}

public class Main {
    public static void main(String[] args) {
        Tier tier = new Tier();
        Tier tierHund = new Hund();
        Hund hund = new Hund();

        System.out.println("alter: " + tier.x); // alter: 5
        System.out.println("alter: " + tierHund.x); // alter: 5
        System.out.println("alter: " + hund.x); // alter: 10.0
    }
}
```

Unterschiede Attribut vs. Methode

- Bei Zugriff auf ein Attribut bestimmt der Typ der Referenzvariable das Attribut

Es wird immer die Methode des Objekt genommen

se

Es wird immer die Variable des Type genommen

- Für Methodenaufruf entscheidet der aktuelle Typ des Objektexemplars welche Version ausgeführt wird

Ausser private, static oder final Methoden!

- *Immer die Version der aktuellen Klasse*

```
class Tier {  
    public void name() {  
        System.out.print("Tier ");  
        foo();  
    }  
    static void foo() {  
        System.out.println("foo Tier");  
    }  
}  
  
class Hund extends Tier {  
    public void name() {  
        System.out.print( "Hund ");  
        foo();  
    }  
    static void foo() {  
        System.out.println("foo Hund");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Tier tier = new Tier();  
        Tier tierHund = new Hund();  
        Hund hund = new Hund();  
  
        tier.name();      // Tier foo Tier  
        tierHund.name(); // Hund foo Hund  
        hund.name();     // Hund foo Hund  
    }  
}
```

Interfaces

Interfaces

Java kennt nur einfache Vererbung

- Klasse kann nur von einer Superklasse erben

Java kennt aber **mehrfache Implementierung von Interfaces**

Was ist ein Interface

Legt eine Menge von Methoden fest

Jede Klasse, die dieses Interface implementiert, muss für alle Methoden-Header ein Body angeben

- Eine Klasse die ein Interface *implementiert* muss für alle Methoden Code enthalten
 - Methode ist *abstrakt* im Interface
 - Methode ist *konkret* in der Klasse

Java Interface Deklaration

Syntax:

```
public interface name {  
    public type name(type name, ..., type name);  
    public type name(type name, ..., type name);  
    ...  
}
```

“Abstrakte Methoden”



Beispiel:

```
public interface Vehicle {  
    public void start();  
    public void move();  
    public void stop();  
}
```

Gebrauch eines Interfaces

- Eine Klasse deklariert dass sie ein Interface *implementiert*
 - Klasse *muss* Code für *alle* Methoden des Interfaces enthalten. (Sonst kann die Klasse nicht übersetzt werden.)
- Syntax:

```
class name implements interface {  
    ...  
}
```
- Damit erklärt die Klasse dass die *ist-ein* Beziehung gilt
 - Ein Exemplar der Klasse ist-ein (was das Interface bestimmt)
 - Exemplar ist-ein Objekt das Methoden des Interfaces zur Verfügung stellt

Sichtbarkeit und Interfaces

- **Die Sichtbarkeit der im Interface deklarierten Methoden ist immer `public`**
 - Auch wenn wir es nicht hinschreiben (andere Sichtbarkeit verboten)
- **Das Interface selbst kann default (`Package`) oder andere (z.B. `public`) Sichtbarkeit haben**
 - In der Praxis finden Sie oft `public`

```
public interface Personal {  
    public Datum getEintritt();  
    public Datum getAustritt();  
}
```

Was sehr nützlich ist

Mehrfache Implementation von Interfaces

Interface-Erweiterung (Interface-Vererbung)

Referenzvariablen vom Typ eines Interfaces

→ Polymorphismus mit Interfaces

Mehrfache Implementation von Interfaces

- Eine Klasse kann mehr als ein Interface *implementierten*

```
public class name implements interface1, interface2 {  
    ...  
}
```
- Muss dann alle Methoden aus *interface1* und *interface2* implementieren

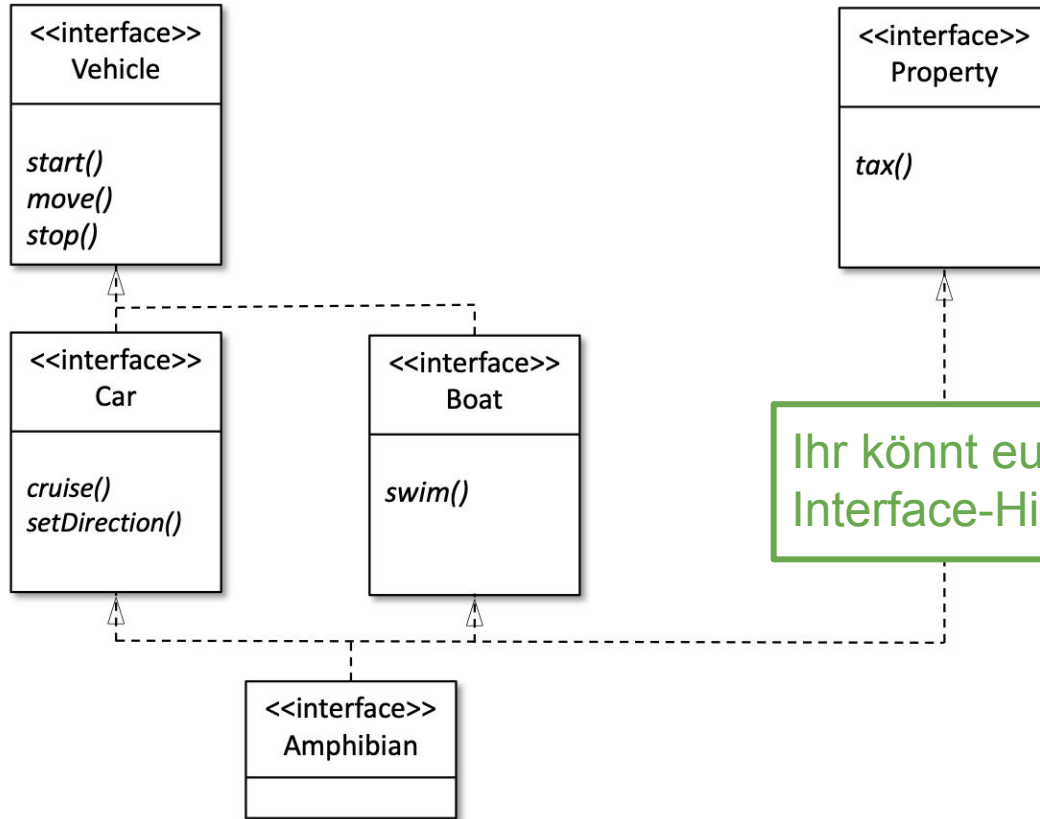
Interface-Erweiterung

- Ein Interface kann mehrere Interfaces erweitern.

```
public interface name extends name1, name2 {  
    public type name(type name, ..., type name);  
}
```

- Beispiel:

```
public interface Boat extends Vehicle {  
    public void swim();  
}  
public interface Property {  
    public double tax();  
}  
public interface Amphibian extends Car, Boat, Property {  
}
```



Ihr könnt euch solche
Interface-Hierarchien bauen

Polymorphismus mit Interfaces

- **Referenzvariable können auf Exemplare verweisen, die ein Interface implementieren**
 - **Definition wie Referenzvariable für Objektexemplare**
 - **Kann als Variable oder als Parameter verwendet werden – wie Referenzvariable für Objektexemplare**
 - **Erlaubt Definition von Methoden die als Parameter Exemplare aller Klassen, die ein Interface implementieren, akzeptieren**
- **Polymorphismus für Klassen die ein Interface implementieren**

Referenzvariable: Typ durch Interface bestimmt

- **Gegeben sei**

```
public interface Shape {  
    public static final double PI_APPROX = 3.14159;  
    public double area();  
    public double perimeter();  
}
```

- **Dann können wir Referenzvariable und Parameter vom Typ Shape deklarieren**

```
Shape s;  
Shape s = new Circle(); //siehe naechste Seite  
void process(Shape myS) { ... }
```

Polymorphismus mit Interfaces in Action

- **Mit dem Interface Shape kann jetzt eine Methode definiert werden die Shape Objekte akzeptiert**
 - **Exemplare *aller* Klassen die das Interface Shape implementieren**

```
public static void printInfo(Shape s) {  
    System.out.println("The shape: " + s);  
    System.out.println("area : " + s.area());  
    System.out.println("perim: " + s.perimeter());  
}
```
- **Jedes Objekt, das das Interface implementiert, kann als Parameter übergeben werden**

- **Auch Arrays von Referenzen sind möglich**

```
Shape[] shapes = {circ, rect};  
println(shapes[0]);  
println(shapes[1]);
```

Exceptions

Exceptions

Was passiert, wenn ihr einen Bug in eurem Programm habt, sodass euer Programm nicht mehr weiterlaufen kann?

→ Exceptions (& Handling)

Exceptions

<https://rollbar.com/blog/java-exceptions-hierarchy-explained/#:~:text=In%20Java%20%E2%80%9Can%20event%20that,run%2Dtime%20in%20application%20code.>

Exception vs Error

Die Klasse **Exception** wird für Ausnahmebedingungen verwendet, die die Anwendung möglicherweise behandeln muss. Beispiele für Ausnahmen (Exceptions) sind **IllegalArgumentException**, **ClassNotFoundException** und **NullPointerException**.

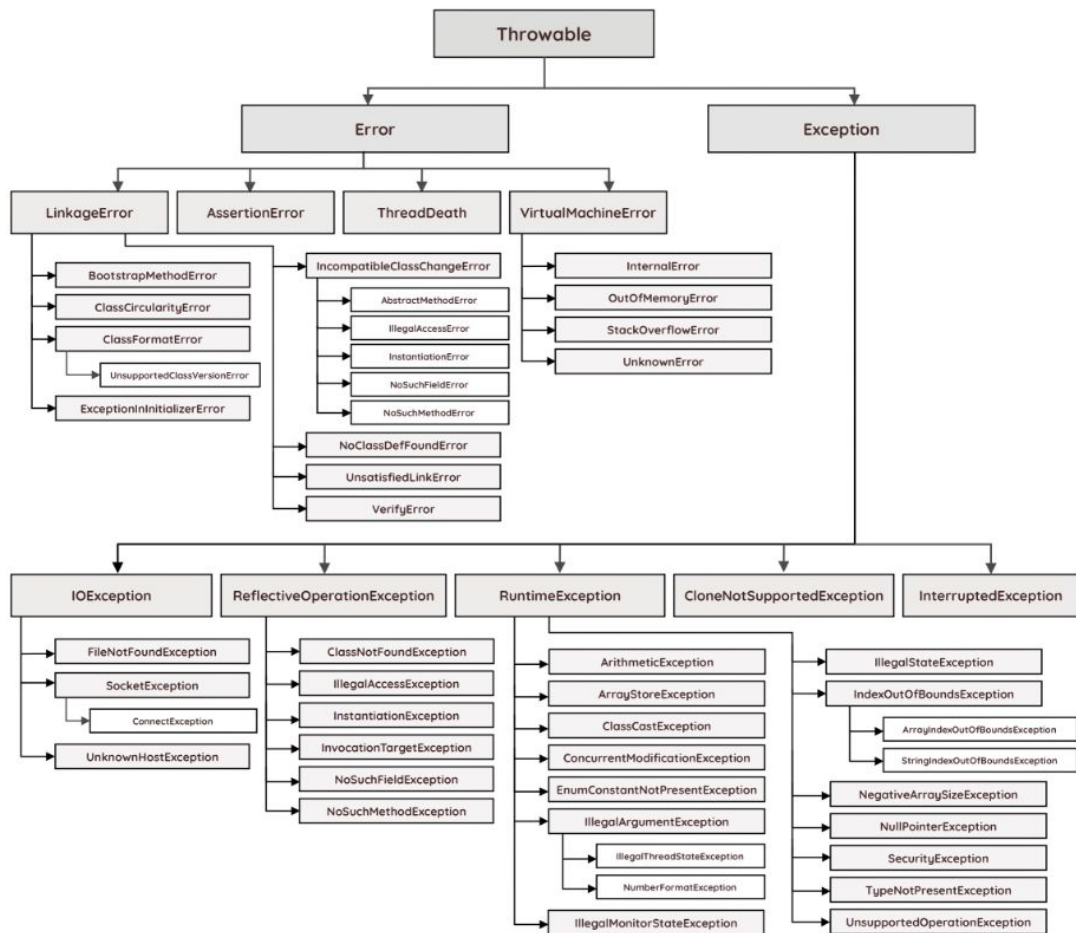
Die Klasse **Error** wird verwendet, um ein schwerwiegenderes Problem in der Architektur anzuzeigen, das nicht im Code behandelt werden sollte. Beispiele für Fehler sind **StackOverflowError**, **OutOfMemoryError** und **AssertionError**.

Unchecked Exceptions

Unchecked Exceptions können "jederzeit" (d. h. zur Laufzeit) ausgelöst werden. Daher müssen Methoden unchecked Exceptions nicht explizit abfangen oder auslösen. Klassen, die **RuntimeException** erben, sind ungeprüfte Ausnahmen, z. B. **ArithmeticException**, **NullPointerException**.

Checked Exceptions

Exceptions, die zur Kompilierzeit auftreten können, werden als checked Exceptions bezeichnet, da sie explizit geprüft und im Code behandelt werden müssen. Klassen, die direkt von **Throwable** erben - außer `RuntimeException` und `Error` - sind geprüfte Ausnahmen, z. B. **`IOException`**, **`FileNotFoundException`** usw.



Exceptions handeln

Mit **try-catch** oder mit **throws** angeben

```
try {  
    // do something  
} catch (ExceptionType name) {  
  
} catch (ExceptionType name) {  
  
} finally {  
    // scanner.close();  
}
```

```
public class Main {  
    public static void main(String[] args) {  
  
        Scanner scanner = new Scanner(new File("input.txt"));  
        int x = scanner.nextInt();  
        scanner.close();  
  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) throws FileNotFoundException {  
  
        Scanner scanner = new Scanner(new File("input.txt"));  
        int x = scanner.nextInt();  
        scanner.close();  
  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
  
        Scanner scanner;  
        int x;  
        try {  
            scanner = new Scanner(new File("input.txt"));  
            x = scanner.nextInt();  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } finally {  
            x = 10;  
        }  
    }  
}
```

mit **throw** selber eine exception werfen

```
throw new IllegalArgumentException();
```

Polymorphismus im Einsatz

```
class Wal {  
    void a() {  
        System.out.print( "Wal a " );  
        b();  
    }  
  
    void b() {  
        System.out.print( "Wal b " );  
    }  
  
    public String toString() { return "Wal"; }  
}  
  
class Zahnwal extends Wal {  
    void b() {  
        System.out.print( "Zahnwal b " );  
    }  
}
```

Was wird von `a()` ausgegeben, wenn `a()` auf einem `Zahnwal`-Objekt aufgerufen wird?

Polymorphismus im Einsatz

```
class Wal {  
    void a() {  
        System.out.print( "Wal a " );  
        b();  
    }  
  
    void b() {  
        System.out.print( "Wal b " );  
    }  
  
    public String toString() { return "Wal"; }  
}  
  
class Zahnwal extends Wal {  
    void b() {  
        System.out.print( "Zahnwal b " );  
    }  
}
```

Was wird von `a()` ausgegeben, wenn `a()` auf einem `Zahnwal`-Objekt aufgerufen wird?

Ausgabe: "Wal a Zahnwal b "

Fortsetzung

```
class Wal {
    void a() {
        System.out.print( "Wal a ");
        b();
    }

    void b() {System.out.print( "Wal b ");}

    public String toString() { return "Wal"; }
}

class Zahnwal extends Wal {
    void b() { System.out.print( "Zahnwal b ");}
}
```

```
class Schweinswal extends Delfinartig {
    void b() { System.out.print( "Schweinswal b "); }
}

class Delfinartig extends Zahnwal {
    void a() {
        System.out.print( "Delfinartig a ");
        super.a();
    }

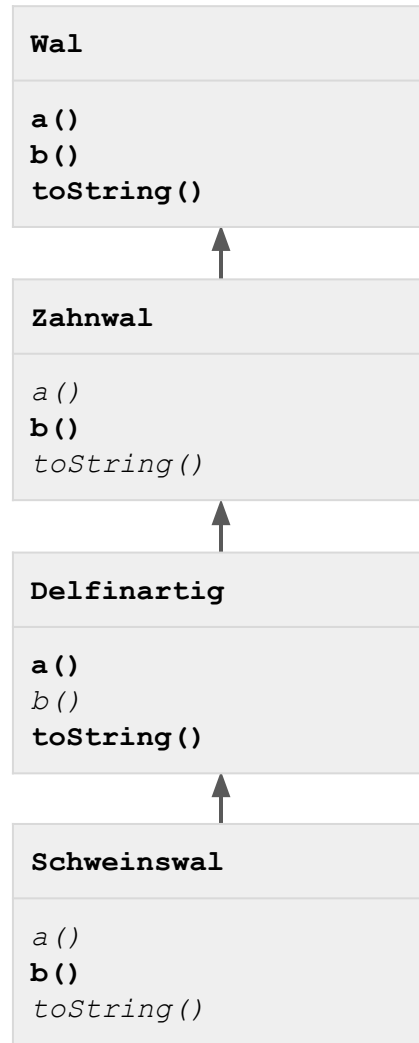
    public String toString() { return "Delfinartig"; }
}
```

```
Wal [] inTheOcean = { new Zahnwal(), new Wal(), new Schweinswal(), new Delfinartig()};
new Zahnwal().b();
for (int i = 0; i < inTheOcean.length; i++) {
    System.out.println( inTheOcean[i] );
    inTheOcean[i].a();
    System.out.println();
    inTheOcean[i].b();
    System.out.println();
    System.out.println();
}
```

Was wird ausgegeben? (für den Zahnwal, Wal, Schweinswal und Delfinartig)

Klassendiagramm

```
class Wal {  
    void a() { ... }  
    void b() { ... }  
    public String toString() { ... }  
}  
  
class Zahnwal extends Wal {  
    void b() { ... }  
}  
  
class Delfinartig extends Zahnwal {  
    void a() { ... }  
    public String toString() { ... }  
}  
  
class Schweinswal extends Delfinartig {  
    void b() { ... }  
}
```



Mit Tabelle

Subtype-Relation

method	Wal	Zahnwal	Delfinartig	Schweinswal
a	Wal a b()	<i>Wal a</i> b()	Delfinartig a Wal a b()	<i>Delfinartig a</i> <i>Wal a</i> b()
b	Wal b	Zahnwal b	Zahnwal b	Schweinswal b
toString	Wal	<i>Wal</i>	Delfinartig	<i>Delfinartig</i>

Vorbesprechung Übung 10

Aufgabe 1: Loop Invariant

Gegeben ist eine Postcondition für das folgende Programm

```
public int compute(String s, char c) {  
    int x;  
    int i;  
  
    x = 0;  
    i = -1;  
  
    // Loop Invariante:  
    while (x < s.length() && i < 0) {  
        if (s.charAt(x) == c) {  
            i = x;  
        }  
        x = x + 1;  
    }  
  
    // Postcondition:  
    // (0 <= i && i < s.length() && s.charAt(i) == c) || count(s, c) == 0  
    return i;  
}
```

Die Methode `count(String s, char c)` gibt zurück wie oft der Character `c` im String `s` vorkommt. Schreiben Sie die Loop Invariante in die Datei "LoopInvariante.txt". **Achtung:** Die Aufgabe ist schwerer als es zuerst scheint. Überprüfen Sie Ihre Lösung sorgfältig.

Aufgabe 2: Cyclic List (Recap!)

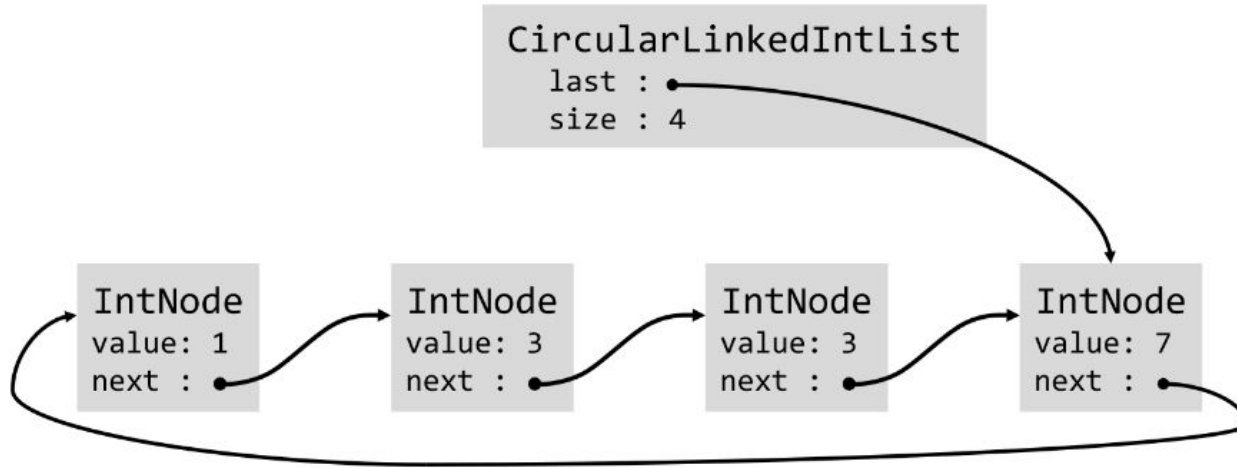


Abbildung 1: Zyklische Liste mit Werten 1, 3, 3, 7.

Aufgabe 3: Exceptions

Ziel: Fehlerbehandlung mit Exceptions

- Was, wenn dataRow sinnlose Werte enthält?
- Person() soll eine checked-Exception
IllegalPersonFormatException werfen

```
public class Person {  
    public int age; // years  
    public int weight; // kg  
    public int height; // cm  
    public boolean isMale;  
  
    public Person(String dataRow) {  
        Scanner scanner = new Scanner(dataRow);  
        age = scanner.nextInt();  
        weight = scanner.nextInt();  
        height = scanner.nextInt();  
        String gender = scanner.next();  
        isMale = gender.equals("m");  
    }  
    public double bodyMassIndex() {  
        return 10000 * weight  
            / (height * height);  
    }  
}
```

```
public class IllegalPersonFormatException extends Exception {  
    public IllegalPersonFormatException(String message) {  
        super(message);  
    }  
}
```

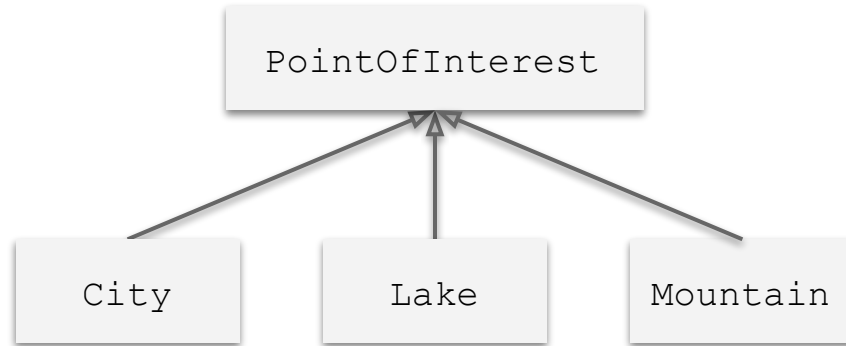
Aufgabe 4: Interaktive Karte

In dieser Übung verwenden Sie die Window-Klasse um eine interaktive Karte zu erstellen. Auf der Karte werden verschiedene **points of interest** (POI) angezeigt. Wenn der Benutzer mit der Maus auf einen solchen POI zeigt, sollen einige Informationen dazu angezeigt werden.



Aufgabe 4: Interaktive Karte

Ausserdem finden Sie die PointOfInterest-Klasse, welche die Basis-Klasse für verschiedene Arten von pois bildet. Ihre Aufgabe ist es, drei Subklassen von PointOfInterest (d.h. Klassen, die von PointOfInterest erben) zu erstellen: City, Lake und Mountain.



Zusatzübungen

Exceptions

Exceptions 1

```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println("main()");  
        int iReturned = new TryCatch().m();  
        System.out.println("m returned " + iReturned);  
    }  
  
    public int m() {  
        int i = 0;  
        try {  
            System.out.println("m(): try");  
            i = 100 / 0;  
        } catch (Exception e) {  
            System.out.println("m(): catch");  
            i = 200;  
        }  
        return i;  
    }  
}
```

Was wird ausgegeben?

Exceptions 1

```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println( "main()" );  
        int iReturned = new TryCatch().m();  
        System.out.println( "m returned " + iReturned );  
    }  
  
    public int m() {  
        int i = 0;  
        try {  
            System.out.println( "m(): try" );  
            i = 100 / 0;  
        } catch (Exception e) {  
            System.out.println( "m(): catch" );  
            i = 200;  
        }  
        return i;  
    }  
}
```

Was wird ausgegeben?

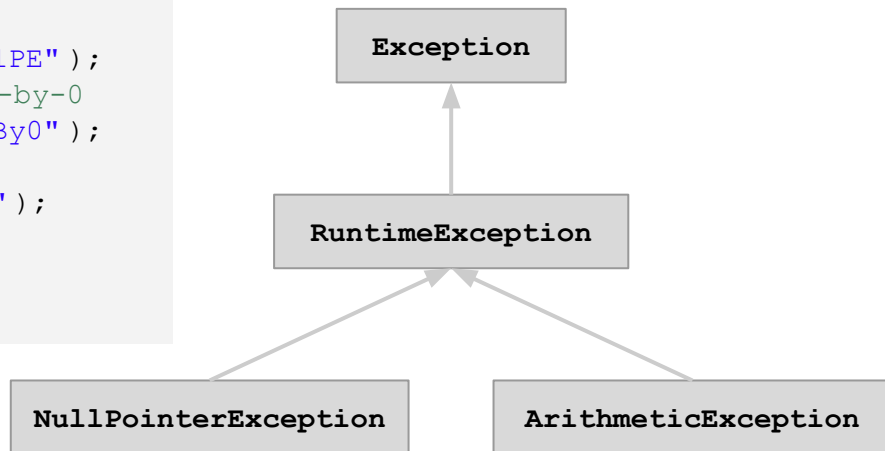
Ausgabe:
main()
m(): try
m(): catch
m returned 200

Exceptions 2

```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println( "main()" );  
        new TryCatch().m();  
    }  
  
    public void m() {  
        try {  
            System.out.println( "m(): try" );  
            int i = 1 / 0;  
        } catch (NullPointerException e) {  
            System.out.println( "m(): catch NullPE" );  
        } catch (ArithmeticException e) { // Div-by-0  
            System.out.println( "m(): catch DivBy0" );  
        } catch (Exception e) {  
            System.out.println( "m(): catch Exc" );  
        }  
    }  
}
```

Was ist die Reihenfolge der Catch-Clauses?

Was wird ausgegeben?



Exceptions 2

```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println( "main()" );  
        new TryCatch().m();  
    }  
  
    public void m() {  
        try {  
            System.out.println( "m(): try" );  
            int i = 1 / 0;  
        } catch (NullPointerException e) {  
            System.out.println( "m(): catch NullPE" );  
        } catch (ArithmeticException e) { // Div-by-0  
            System.out.println( "m(): catch DivBy0" );  
        } catch (Exception e) {  
            System.out.println( "m(): catch Exc" );  
        }  
    }  
}
```

Reihenfolge der Catch-Clauses:

- Div-by-0 ist keine NullPE
- Div-by-0 ist eine
ArithmeticException
- catch(Exception) fängt den
Rest

Ausgabe:

main()
m(): try
m(): catch DivBy0

Exceptions 2: catch-Blöcke vertauscht

```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println( "main()" );  
        new TryCatch().m();  
    }  
  
    public void m() {  
        try {  
            System.out.println( "m(): try" );  
            int i = 1 / 0;  
        } catch (NullPointerException e) {  
            System.out.println( "m(): catch NullPE" );  
        } catch (Exception e) {  
            System.out.println( "m(): catch Exc" );  
        } catch (ArithmeticException e) { // Div-by-0  
            System.out.println( "m(): catch DivBy0" );  
        }  
    }  
}
```

Was wird ausgegeben?

Exceptions 2: catch-Blöcke vertauscht

```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println( "main()" );  
        new TryCatch().m();  
    }  
  
    public void m() {  
        try {  
            System.out.println( "m(): try" );  
            int i = 1 / 0;  
        } catch (NullPointerException e) {  
            System.out.println( "m(): catch NullPE" );  
        } catch (Exception e) {  
            System.out.println( "m(): catch Exc" );  
        } catch (ArithmeticException e) { // Div-by-0  
            System.out.println( "m(): catch DivBy0" );  
        }  
    }  
}
```

Ausgabe:

Compiler-Error: "Unreachable catch block for ArithmeticException. It is already handled by the catch block for Exception"

Fängt nie etwas → Fehler

Exceptions 3

```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println( "main()" );  
        try {  
            new TryCatch().m();  
        } catch (RuntimeException e) {  
            System.out.println( "main(): catch Exc" );  
        }  
    }  
  
    public void m() {  
        try {  
            System.out.println( "m(): try" );  
            int i = 1 / 0;  
        } catch (RuntimeException e) {  
            System.out.println( "m(): catch Exc" );  
            throw e;  
        }  
    }  
}
```

Was wird ausgegeben?

Exceptions 3

```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println( "main()" );  
        try {  
            new TryCatch().m();  
        } catch (RuntimeException e) {  
            System.out.println( "main(): catch Exc" );  
        }  
    }  
  
    public void m() {  
        try {  
            System.out.println( "m(): try" );  
            int i = 1 / 0;  
        } catch (RuntimeException e) {  
            System.out.println( "m(): catch Exc" );  
            throw e;  
        }  
    }  
}
```

Was wird ausgegeben?

Ausgabe:

main()

m(): try

m(): catch Exc

main(): catch Exc

Exceptions 4

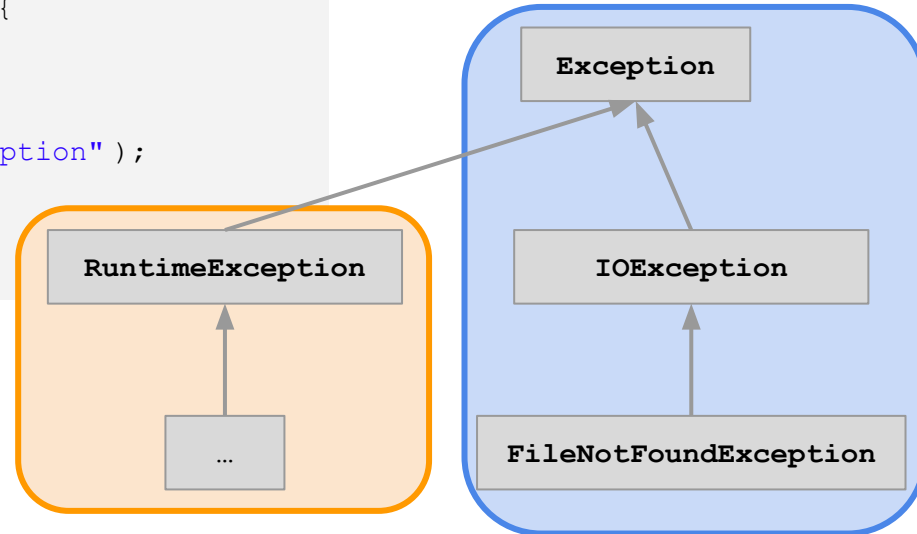
```
public class TryCatch {  
    public static void main(String[] args) {  
        System.out.println("main()");  
        try {  
            new TryCatch().m();  
        } catch (Exception e) {  
            System.out.println("caught exception");  
        }  
    }  
  
    public void m() throws FileNotFoundException {  
        try {  
            throw new FileNotFoundException();  
        } catch (RuntimeException e) {  
            System.out.println("unchecked exception");  
        }  
    }  
}
```

Fängt checked und
unchecked Exceptions

Wenn checked Exception in der
Methode nicht abgefangen wird,
throws.

checked Exceptions

unchecked Exceptions



Noch mehr Exceptions

Ein weiteres Beispiel

- Wir wollen zwischen grossen Dateien (> 1000 Bytes) und gigantischen Dateien (>2000 Bytes) unterscheiden.
- Wir wollen auch Feedback geben, wenn das File zu gross ist.
 - Sollte die Datei nicht existieren, so fragen wir nur nach einem anderen Namen.

Ein weiteres Beispiel

```
public class Exception2 {  
  
    public static void main (String[] args)  {  
  
        Scanner console = new Scanner(System.in);  
        Scanner rd = null;  
  
        while (rd == null) {  
            try {  
                try {  
                    System.out.println("Please enter a file name:");  
                    String name = console.next();  
                    rd = getScanner(name);  
                } catch (MyBigException ex) {  
                    System.out.println("This file is HUGE!");  
                } catch (MyException ex) {  
                    System.out.println("This file is large!");  
                }  
  
                // continued on next slide
```

Ein weiteres Beispiel (Fortsetzung 1)

```
// in innermost "try" block
if (rd == null) {
    System.out.println("Please enter name of a small file");
}
// end innermost "try" block -- large file
} catch (IOException ex) { // outer "try", no such file
    System.out.println("File does not exist");
}
// end while loop
}

// use rd to read from small file ...
int i = rd.nextInt();
System.out.println("here " + i);

} // end main
```

Ein weiteres Beispiel (Fortsetzung 2)

```
public static Scanner getScanner(String n) throws
    FileNotFoundException, MyException {
    File f = new File(n);
    if (f.exists() && f.length() > 1000) {
        if (f.length() > 2000) {
            throw new MyBigException();
        } else {
            throw new MyException();
        }
    }
    return new Scanner(f);
} // Exception2

public class MyException extends Exception {
}

public class MyBigException extends MyException {
}
```


■ Die Dateien

```
ls -alt *txt
```

```
8 -rw-----. 1 trg inf      6 Nov 24   2016 data.txt
8 -rw-----. 1 trg inf 2804 Oct 24   2016 hoehe.txt
8 -rw-----. 1 trg inf 1067 Oct 21   2015 hs.txt
```

■ Ausführung

Please enter a file name:

no-such-file.txt

File does not exist

Please enter a file name:

hs.txt

This file is large!

Please enter name of a small file

Please enter a file name:

hoehe.txt

This file is HUGE!

Please enter name of a small file

Please enter a file name:

data.txt

here 33333

Frage

Was passiert wenn wir die Reihenfolge von

```
} catch (MyBigException ex) {  
    System.out.println("This file is HUGE!");  
} catch (MyException ex) {  
    System.out.println("This file is large!");  
}
```

in

```
} catch (MyException ex) {  
    System.out.println("This file is large!");  
} catch (MyBigException ex) {  
    System.out.println("This file is HUGE!");  
}
```

ändern?

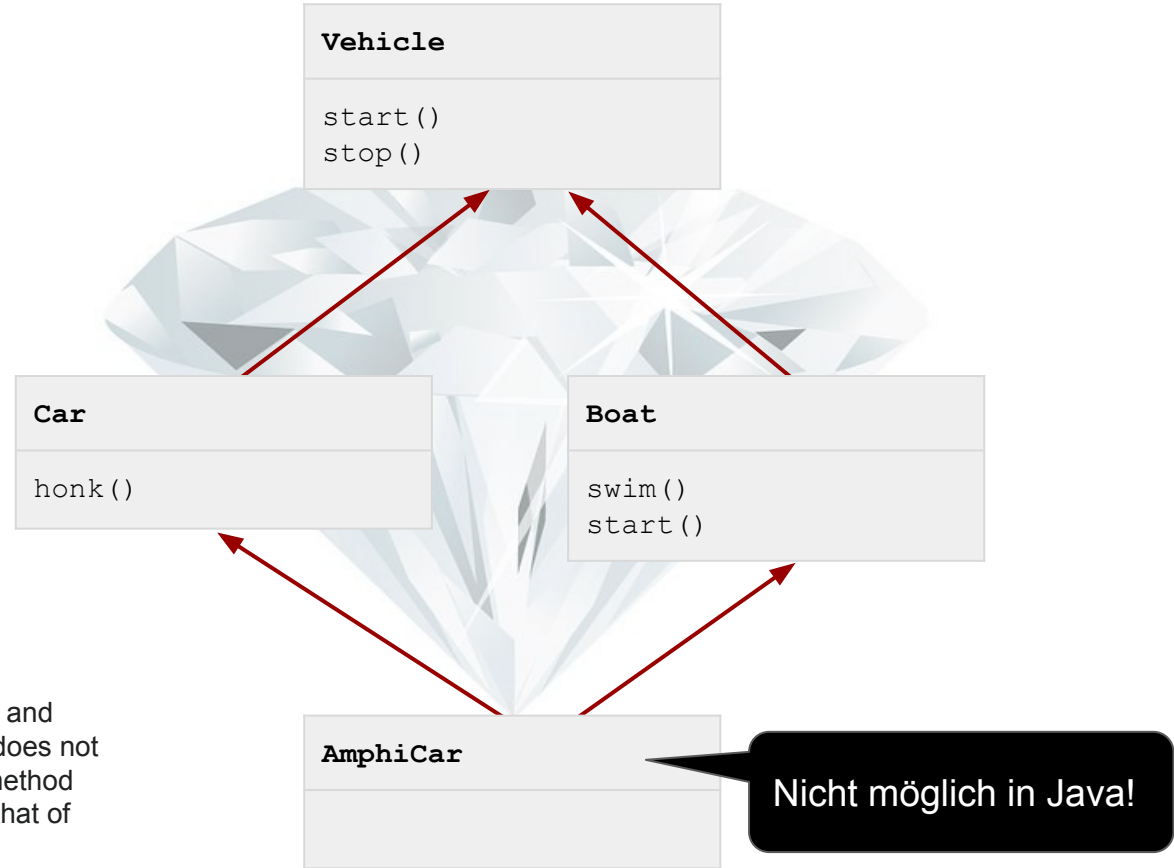
Vererbung, Interfaces

Diamanten

```
class Vehicle {  
    public void start() {System.out.println("start");};  
    public void stop() {System.out.println("stop");};  
}  
  
class Car extends Vehicle {  
    public void honk() {System.out.println("*toot*");};  
}  
  
class Boat extends Vehicle {  
    public void swim() {System.out.println("swim...");}  
    public void start() {System.out.println("depart");};  
}  
  
class AmphiCar extends Car, Boat {}
```

Nicht möglich in Java!

Diamanten

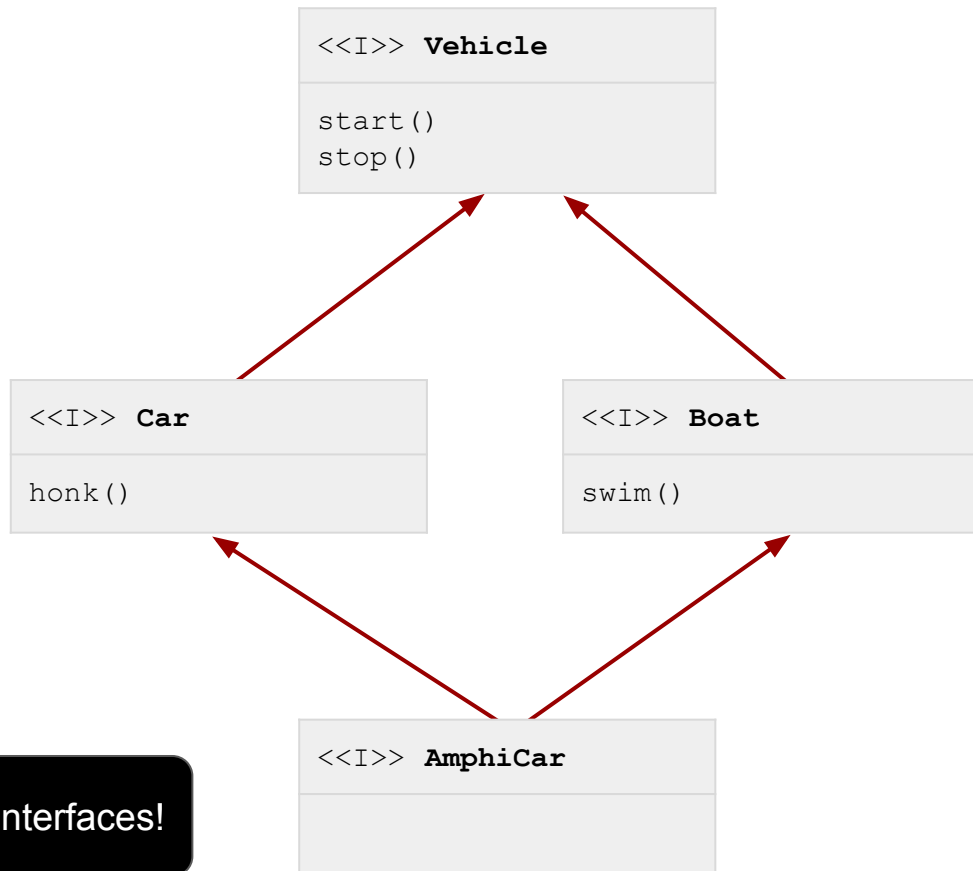


If there is a method in **Vehicle** that **Car** and **Boat** have overridden, and **AmphiCar** does not override it, then which version of the method does **AmphiCar** inherit: that of **Car**, or that of **Boat**?

Interfaces!

```
interface Vehicle {  
    public void start();  
    public void stop();  
}  
  
interface Car extends Vehicle {  
    public void honk();  
}  
  
interface Boat extends Vehicle {  
    public void swim();  
}  
  
interface AmphiCar extends Car, Boat {}
```

Möglich mit Interfaces!



Interfaces!

```
interface Vehicle {  
    public void start();  
    public void stop();  
}  
  
interface Car extends Vehicle {  
    public void honk();  
}  
  
interface Boat extends Vehicle {  
    public void swim();  
}  
  
interface AmphiCar extends Car, Boat {}
```

```
class AbstractVehicle implements Vehicle {  
    public void start() { System.out.println("start"); }  
    public void stop() { System.out.println("stop"); }  
}  
  
class BMWCar extends AbstractVehicle implements Car {  
    public void honk() { System.out.println("*honk*"); }  
}  
  
class SpeedBoat extends AbstractVehicle implements Boat {  
    public void swim() { System.out.println("swim..."); }  
    public void start() { System.out.println("depart"); }  
}  
  
class AmphiCarM770 extends AbstractVehicle implements  
    AmphiCar {  
    public void honk() { System.out.println("*toot*"); }  
    public void swim() { System.out.println("swim..."); }  
}
```

- Interfaces für Subtyping
- Inheritance für Code-Sharing

