UNIVERSITÄT
LEIPZIG

EOOMP

Übung 6

Fragen?

1. **Exceptions**

WIEDERHOLUNG

AUFGABE

Erinnern Sie sich an die Klassen „Tier“, „Hund“, „Katze“ von letzter Übung! Implementieren Sie nun eine Klasse `Mensch`. Wählen Sie passende Sichtbarkeiten für Attribute und Methoden.

```java
public class Mensch {

    //Attribute
    /*
     * Der Mensch hat ein Alter. Dies ist eine Ganzzahl.
     */

    //Konstruktor
    /*
     * Beim Erzeugen einer Instanz von "Mensch",
     * soll direkt das Alter als Parameter übergeben werden und
     * im Attribut gespeichert werden.
     */

    //Methoden
    /*
     * Der Mensch soll sich vorstellen können.
     * Er gibt dabei auch sein Alter an.
     * (z.B. "Hallo ich bin ein Mensch und bin "x" Jahre alt.)
     */
```

AUFGABE

- Implementieren Sie eine weitere Klasse `Seefahrer`,

welche die Klasse `Mensch` spezialisiert.

- Der **Seefahrer** soll nun zusätzlich einen **Namen** haben,

welcher ebenfalls bei der **Erstellung einer Instanz** übergeben werden soll.

- **Überschreiben** Sie die Methode `vorstellen()`, damit der Seefahrer sich wie folgt vorstellen kann:

„Hallo, ich bin *name*. Ahoi!“

- Überlegen Sie sich, wie man **“zählen“** kann, **wie viele Menschen** und **wie viele Seefahrer** in der main-Methode erzeugt wurden.

JAVA.LANG.OBJECT

JAVA.LANG.OBJECT

In Java hat jede Klasse eine Oberklasse, außer `java.lang.Object` .

Das bedeutet alle Klassen sind implizit eine Erweiterung von `java.lang.Object` .

Also kann man alle Methoden von `java.lang.Object` bei allen Objekten verwenden:

z.B. `toString`, `hashCode`, `equals`,...

JAVA.LANG.OBJECT

`public String toString()`

- Gibt eine String-Repräsentation des Objektes zurück
- Klassenname + @ + Hashcode in Hexadezimal

`public int hashCode()`

- Gibt einen Int-Wert zurück, der den Speicherort des Objektes repräsentiert

`public boolean equals (Objekt obj)`

- Vergleicht zwei Objekte auf Gleichheit
- Achtung: Vergleicht Referenzen! (Prüft, ob beide auf dasselbe Objekt zeigen)

EXCEPTIONS

EXCEPTIONS

Ausnahmen (Exceptions) sind Objekte, die Ausnahmesituationen anzeigen und Informationen über diese speichern können.

Man trennt Code von *gewöhnlichen Situation*en von Code von *außergewöhnlichen Situationen*.

(Bsp. Negative Zahlen als Radius – gewöhnliche Situation (positiver Radius) // mögliche Fehler/außergewöhnliche Situationen (z.B. negativer Radius))

Die Frage ist: Was sind mögliche Fehlerquellen/Ausnahmen und was möchte man mit ihnen tun?

- Soll das Programm abgebrochen werden?
- Soll der Fehler ignoriert werden? Soll eine Alternative genutzt werden?
- Soll eine Nachricht mit Informationen ausgegeben werden?

EXCEPTIONS

Exceptions werden mit dem Schlüsselwort `throw` „geworfen“.

Damit man sie fangen kann, muss der Teil des Programms, bei dem eine Ausnahme auftreten kann, mit einem `try`-Block umgeben werden. Dann folgen (mehrere) `catch`-Blöcke für deren Behandlung.

Gibt es mehrere `catch`-Blöcke, dann wird der erste zutreffende `catch`-Block ausgeführt (wie auch bei `if`, `if else`,... `else`).

EXCEPTIONS

Mehrere `catch`-Blöcke funktionieren nur dann, wenn **verschiedene Fehler-Arten/Exception Typen** abgefangen werden sollen. Beispiele:

1. Ich möchte bei 4 verschiedenen Ausdrücken/Berechnungen prüfen, ob *durch 0 geteilt* wird. Das würde man in **einem** catch-Block abfangen (z.B. `ArithmeticException`)
2. Ich möchte auf eine Stelle in meinem Array zugreifen und damit rechnen:
   Ich möchte prüfen, ob ich einen *gültigen Index* erwischt habe und
   ich möchte prüfen, ob meine Berechnung „legal“ ist. z.B. keine *Division durch 0*.
   Das wären **zwei** catch-Blöcke (z.B. `IndexOutOfBoundsException` und `ArithmeticException`)

EXCEPTIONS

Man unterscheidet in Unchecked Exceptions, Checked Exceptions und Error.

Unchecked Exceptions sind Instanzen von `RuntimeException` und deren Unterklassen.

Das sind z.B. `NullPointerExceptions`, `ArrayIndexOutOfBoundsException`.

Diese Exceptions müssen **nicht** in der `throws`-Klausel einer Methode deklariert werden.

Solche Ausnahmen können auch häufig durch eine leichte Änderung im Code einfach vermieden werden.

EXCEPTIONS

```
public static void main (String[] args) {
    Mensch[] menschen = new Mensch[10];    //initial nur null Werte!!!
    /*
    * Irgendein Code-Block in dem man vergisst, die „Menschen“ auch tatsächlich
    * mit Werten zu belegen
    */
    try{
        for( int i = 0; i < menschen.length; i++)
        {
            menschen[i].getName();
        } catch (NullPointerException e) {
            System.out.println(„NullPointerException ist aufgetreten: “ + e.getMessage());
    }
}
```

EXCEPTIONS

Anstelle eines try-catch-Blocks, kann man das Problem auch manchmal einfach “umgehen“.

```
public static void main (String[] args) {
    Mensch[] menschen = new Mensch[10];    //initial nur null Werte!!!
    /*
    * Irgendein Code-Block in dem man vergisst, die „Menschen“ auch tatsächlich
    * mit Werten zu belegen
    */
    for( int i = 0; i < menschen.length; i++)
    {
        if( menschen[i] != null)   //Einfacher Trick um eine NullPointerException zu umgehen
        {
            menschen[i].getName();
        }
    }
}
```

EXCEPTIONS

```
public static void main (String[] args) {

    int[] werte = new int[10];     //initial nur null Werte!!!

    int x = werte[11];


}
```

Wie kann man dieses Problem umgehen?

EXCEPTIONS

Man unterscheidet in Unchecked Exceptions, Checked Exceptions und Error.

Checked Exceptions sind Instanzen von `Exception`, aber nicht von `RuntimeException`.

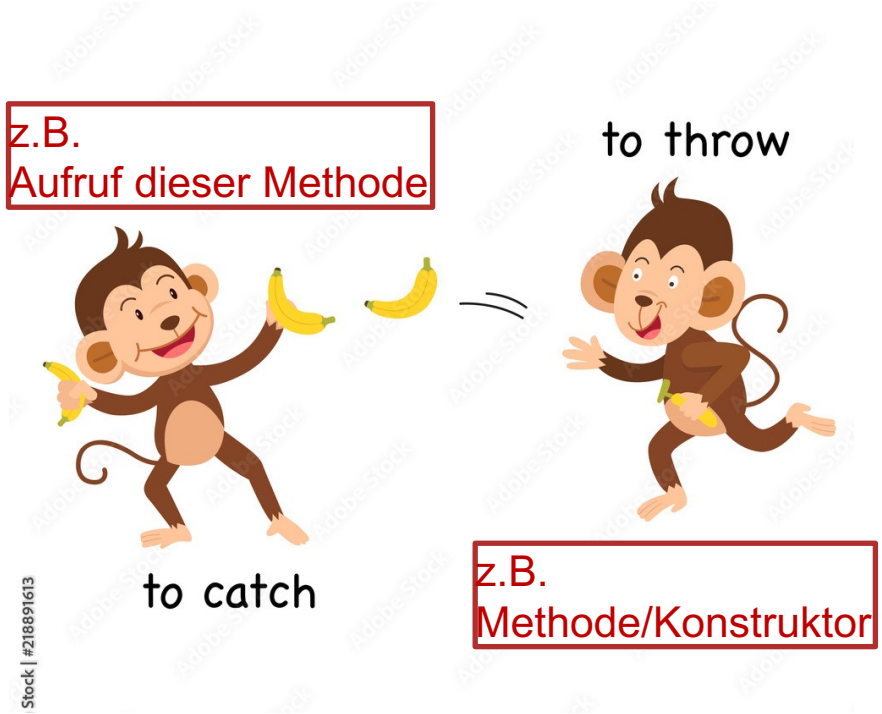
Dies sind „erwartete“ Fehler, die eine Fehlerbehandlung haben sollten.
(negativer Radius beim Erstellen des Kreises, Nutzer gibt falsche Werte ein,...)

Sie **müssen** in der `throws`-Klausel einer Methode deklariert werden, wenn sie bei der Ausführung der Methode ausgelöst werden könnten.

EXCEPTIONS – „IN ANDEREN WORTEN“

Wird eine (neue) Methode oder ein Konstruktor implementiert und man möchte bestimmte Ausnahmen behandeln, dann wird in der Methode/Konstruktor eine `throw`-Klausel eingefügt.

Soll die Methode aufgerufen/ eine Instanz mit dem Konstruktor erstellt werden (z.B. in der Main-Methode), dann schreibt man diesen „Versuch“ in einen `try`-Block. Danach folgt ein `catch`-Block in dem man die Ausnahmen „fängt“.



EXCEPTIONS

```
public Kugel(double radius) throws Exception {
    if(radius < 0) {
        throw new Exception(„Kugel: Radius < 0; Wert: “ + radius);
    }
    this.radius = radius;
}
```

```
public static void main (String[] args) {
    try {
        Kugel k = new Kugel(-1);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

EXCEPTIONS

Man unterscheidet in Unchecked Exceptions, Checked Exceptions und Error.

Error sind Instanzen von `Error` und dessen Unterklassen.

Dies sind Probleme, auf die das Programm wenig Einfluss hat, wie z.B. zu wenig Speicherplatz.

Man sollte sie **weder** werfen noch fangen.

EXCEPTIONS

Man kann auch seine eigenen `throwables` als Unterklasse von `java.lang.Exception` (oder auch `java.lang.RuntimeException`) implementieren.

Das kann man machen, wenn die Standard-Exceptions nicht aussagekräftig genug sind:

- Im Code selbst ist es oft sinnvoll, wenn die Exception einen **aussagekräftigen Namen** hat (z.B. `negativerRadius`, anstelle von `IllegalArgumentException`)
- Die **Art des Fehlers** kann besser abgefangen werden
  (mehrere catch-Blöcke gehen nur, wenn man auch *verschiedene Arten von Fehl*ern hat)
- Fehlermeldungen können **mehr Kontext** mitgegeben bekommen (Bsp. Vorlesung: der „fehlerhafte“ Radius kann mit eingebaut werden)
- Man kann steuern, ob man **Checked oder Unchecked Exceptions** anlegen möchte

INTERFACES

INTERFACES

Ein **Interface** ist eine Sammlung von Methoden **ohne** Implementierung (*). Es kann `public` oder `package private` sein.

Sie dienen zur **Erzwingung** einer bestimmten Struktur/Vorschrift für Klassen, die es implementieren.

Attribute sind erlaubt, aber sie sind automatisch `public`, `static` und `final`.

Die Methoden in einem Interface sind **abstrakte Methoden** und sie sind automatisch immer `public`. Für jede Methode muss die Methoden-Deklaration angegeben werden.

> Im **Interface** steht also *nur*, dass es diese Methode gibt, aber **nicht wie** sie implementiert ist *(„Was sie genau macht“)*. Dies geschieht dann in der Klasse, die das Interface implementiert.

(*) Ausnahme: Soll eine Methode `static` sein, dann muss sie im Interface implementiert werden.

INTERFACES

Erstellung eines Interfaces, seien M1, M2 zwei Methoden:

```
(Sichtbarkeit) interface Interface_Name {
  (Modifikator_M1)  (Rückgabetyp_M1)   (M1_name  (Parameterliste M1)  );
  (Modifikator_M2)  (Rückgabetyp_M2)   (M2_name  (Parameterliste M2)  );
   ...
}
```

Also z.B.

```
public interface Studierende{
  public void lernen();
  public int getMatrikelnummer(String nameStudent);
}
```

INTERFACES

Wenn eine Klasse erweitert werden soll, schreibt man

```
Unterklassen_Name extends Oberklassen_Name
```

Soll nun eine Klasse ein **Interface** implementieren, schreibt man

```
Klassen_Name implements Interface_Name
```

Wenn nun eine Methode implementiert wird, kann man wieder das

Schlüsselwort `@Override` zum Schutz vor Fehlern nutzen.

INTERFACES

Ein Interface kann wieder als Datentyp verwendet werden.

Eine Klasse kann **mehrere** Interfaces implementieren!

```
Klassen_name        implements        Interface_Name1, Interface_Name2
```

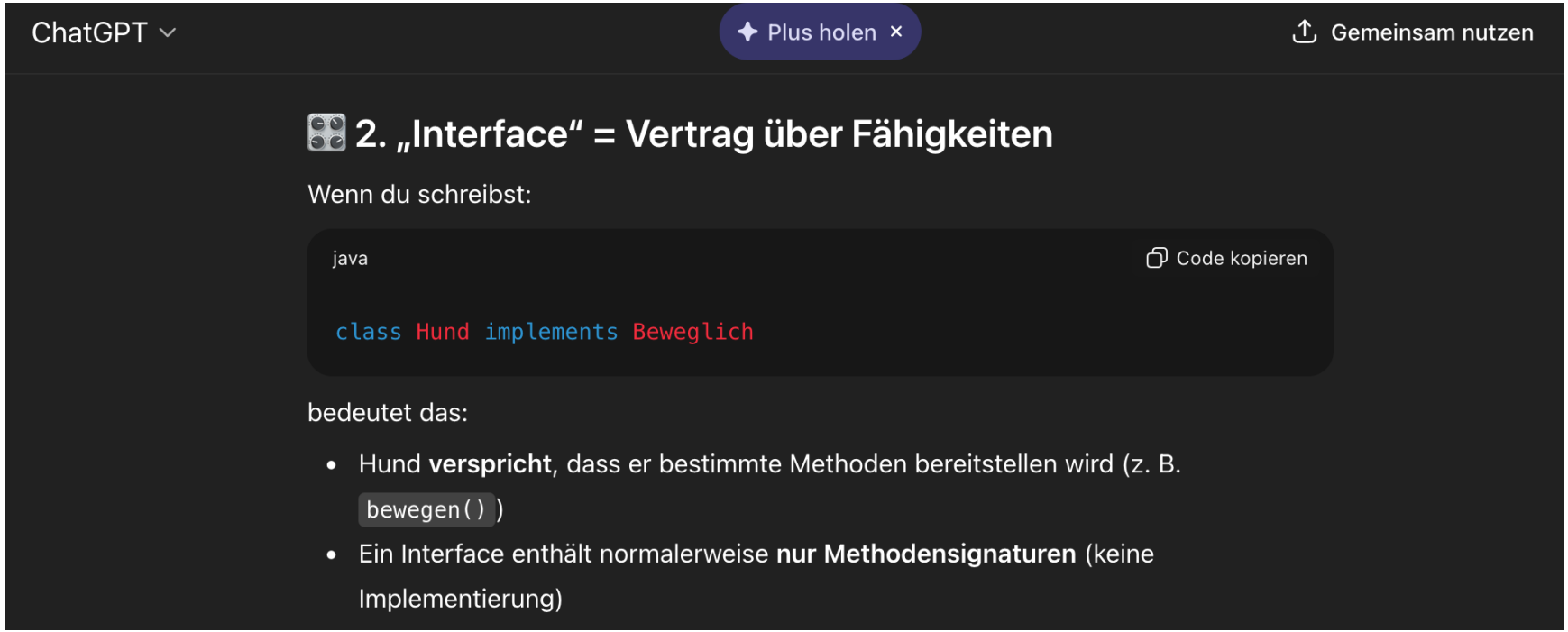
>= 1

Aber eine Klasse kann **nur von einer** Klasse erben!

```
Unterklassen_Name        extends        Oberklassen_Name
```

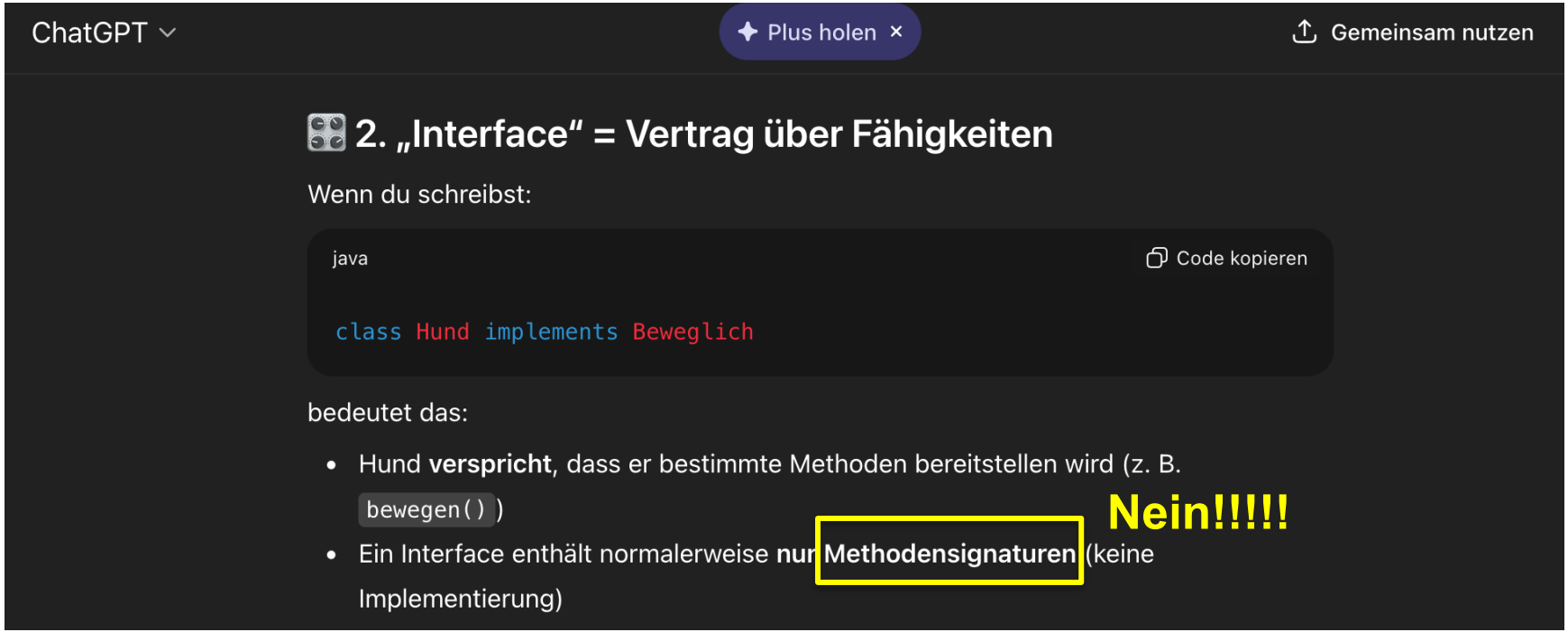
= 1

INTERFACES



Stimmt die folgende Definition für Interfaces von ChatGPT?

INTERFACES



Stimmt die folgende Definition für Interfaces von ChatGPT?

INTERFACES

Signatur einer Methode

- Die **Signatur** einer Methode umfasst folgende Elemente
  1. Den Namen
  2. Die Typen und ihre Reihenfolge in der Parameterliste

- Weitere Elemente der Deklaration einer Methode:
  1. Die Sichtbarkeit (`public`, “package” (kein Schlüsselwort), `protected`, `private`)
  2. Ob die Methode `static` ist, oder nicht
  3. Ob die Methode `final` ist, oder nicht
  4. Der Typ der Rückgabe (erforderlich)

Vorlesung „Klassen und Instanzen“

WEITER IM LEBEWESEN PROJEKT

LEBEWESEN

Wir wollen nun:

1. Tiere und Menschen in **einem** Projekt zusammen führen
   (Wie kann ich packages füllen?, Wie binde ich die packages richtig ein?)
2. Verschiedene Instanzen von Menschen und Tieren erstellen und diese vorstellen. Das soll in nur *einem* Aufruf erfolgen! (Kann man Menschen und Tiere in einem Array speichern???)
3. Eine weitere (Unter-)Klasse `Weihnachtsmann` erstellen. Wie kann man sicher stellen, dass es nur einen Weihnachtsmann gibt?

Fragen bitte immer an

barz@informatik.uni-leipzig.de