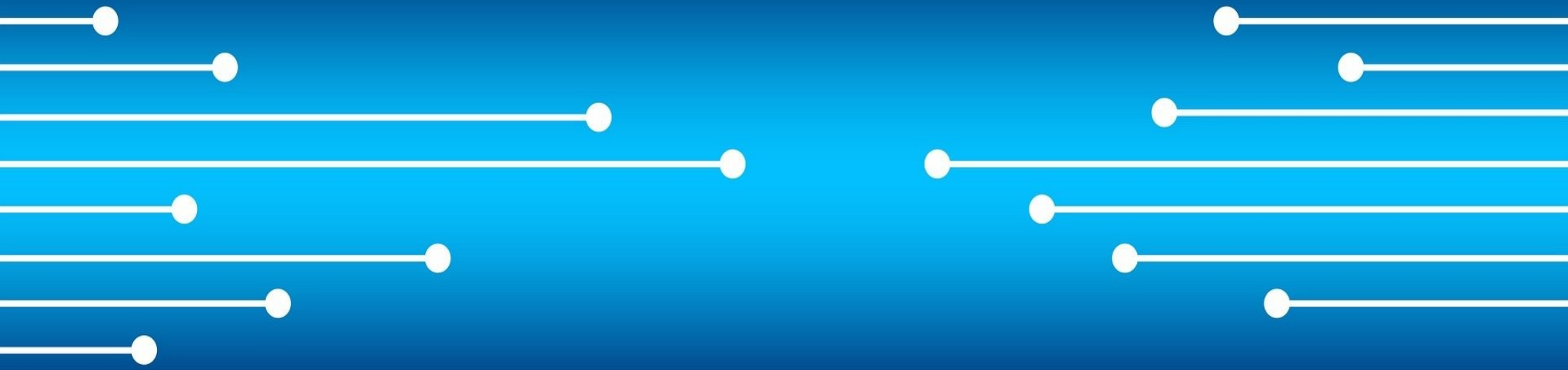
Datenverarbeitung

Teil des Moduls 5CS-DPDL-20



Prof. Dr. Deweß

Thema 5

„Generics“ und „Wildcards“

- Was sind „Generics“ und warum sind sie sinnvoll?
- „Generics“ in Klassen, Schnittstellen und Methode
- „Wildcards“ als nötiges Sahnehäubchen

Fehler ... passieren

„Es ist ein großer Vorteil im Leben, die Fehler, aus denen man lernen kann, möglichst früh zu begehen."

- Sir Winston Churchill

Quelle: Hadeler, Thorsten (2000): Zitate für Manager: Für Reden, Diskussionen und Papers immer das treffende Zitat. S. 250.

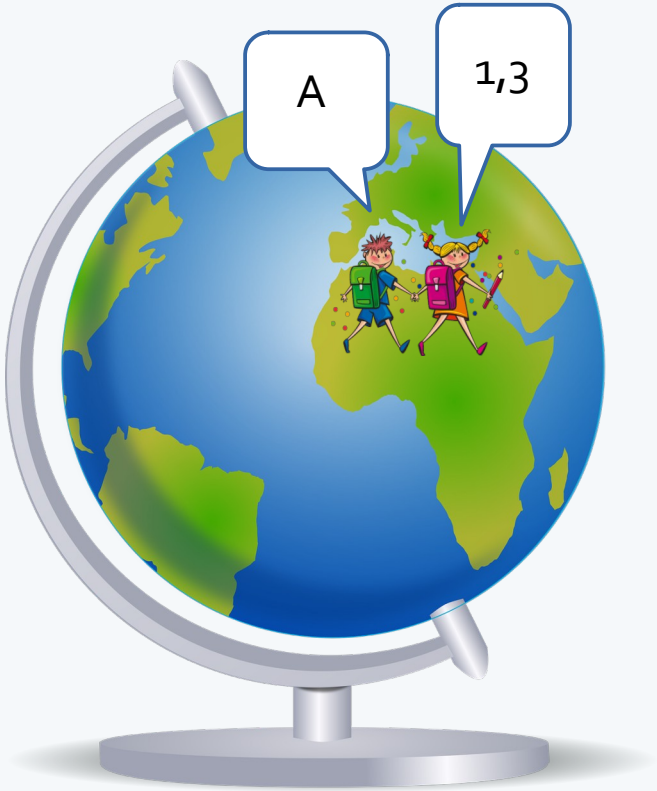
Fehler, die Ihnen automatisch schnell auf die Füße fallen:

- Compilerfehler
- Laufzeitfehler

Welcher Fehler ist Ihnen lieber?
Ich bevorzuge Compilerfehler, weil die garantiert noch bei mir auftreten und nicht erst beim Anwender.

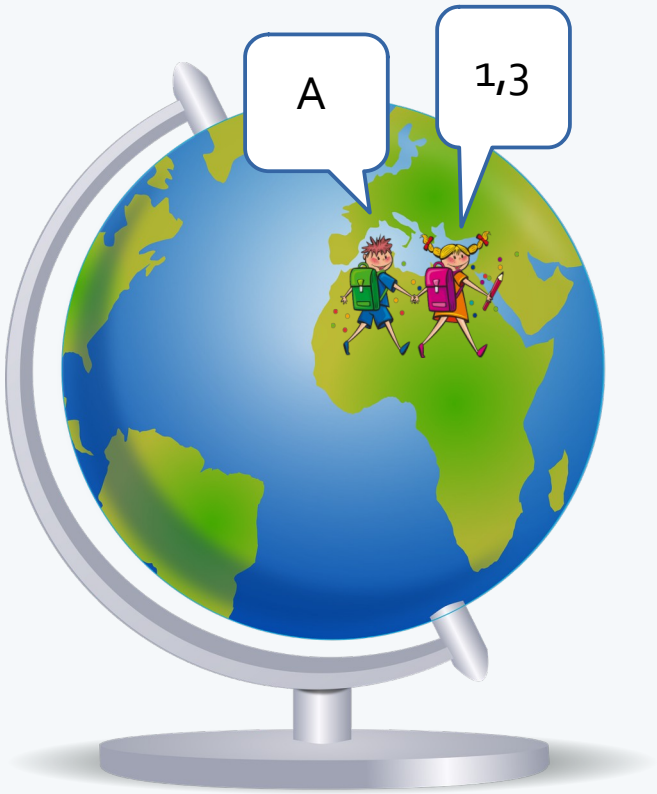
Fehler ... passieren

damit ich gar nicht erst so viele Fehler mache, besuche ich eine Schule und lerne ...
das ist aber auf der Erde kompliziert, überall wird anders bewertet ...
Wie kann ich diese Notensysteme in meiner Software „unter einen Hut bringen"?



Fehler ... passieren

```
package de.baleipzig.generics;
public class Grade {
  private Object value;
  public Object getValue() {
    return value;
  }
  public void setValue(Object aValue) {
    value = aValue;
  }
  public boolean isNotGraded() {
    return value == null;
  }
}
```

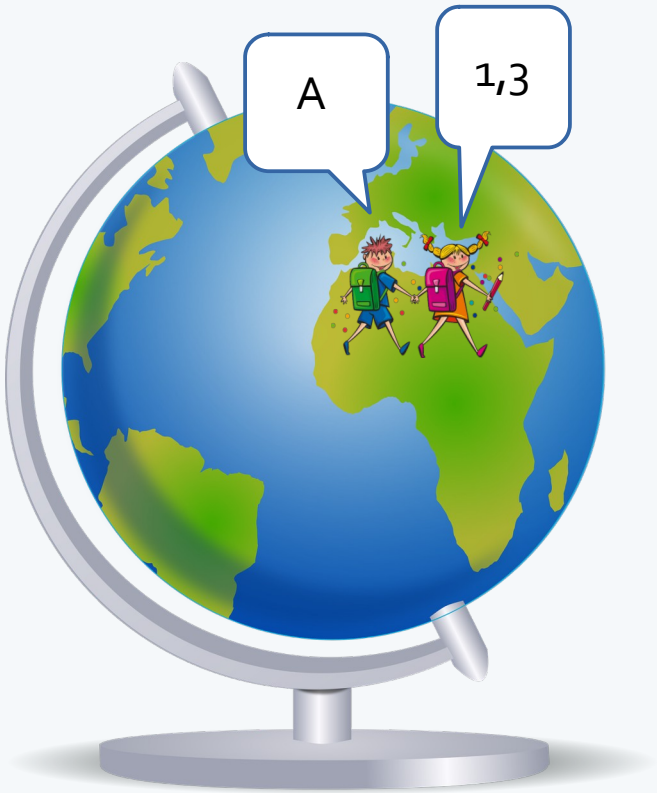


Grundidee: Mit Datentyp „Object“ können wir Schulnoten aller Notensysteme erfassen

Fehler ... passieren

```
package de.baleipzig.generics;
public class Grade {
  private Object value;
  public Object getValue() {
    return value;
  }
  public void setValue(Object aValue) {
    value = aValue;
  }
  public boolean isNotGraded() {
    return value == null;
  }
}
```

```
Grade germanGrade = new Grade();
germanGrade.setValue(1.3);

Grade swedishGrade = new Grade();
swedishGrade.setValue('A');

germanGrade.setValue(„sehr gut“);
```



kann vorkommen; macht sich bei einer Durchschnitts-berechnung aber nicht gut ...

Fehler ... passieren

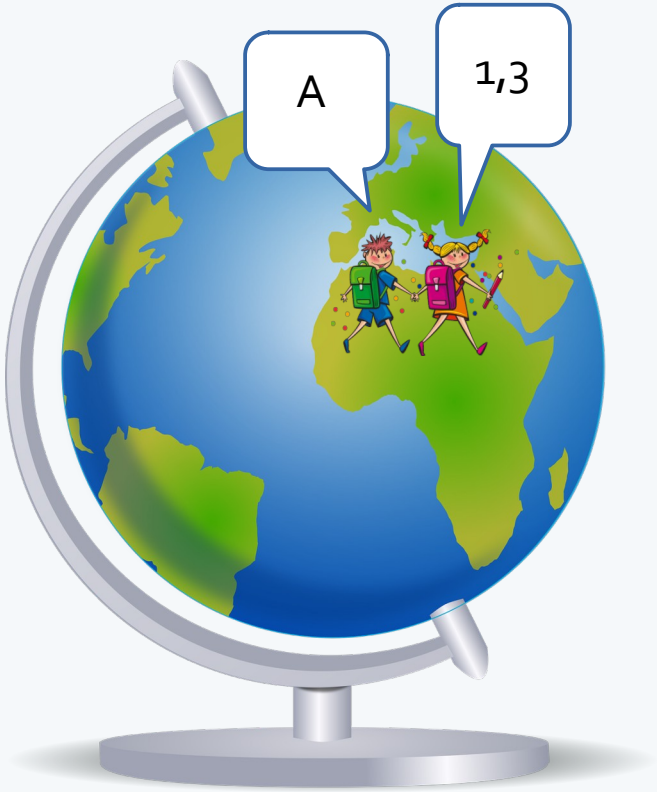
Und nun?

```
package de.baleipzig.generics;
public class afghanGrade { … }
```

```
package de.baleipzig.generics;
public class egyptianGrade { … }
```

.
.
.

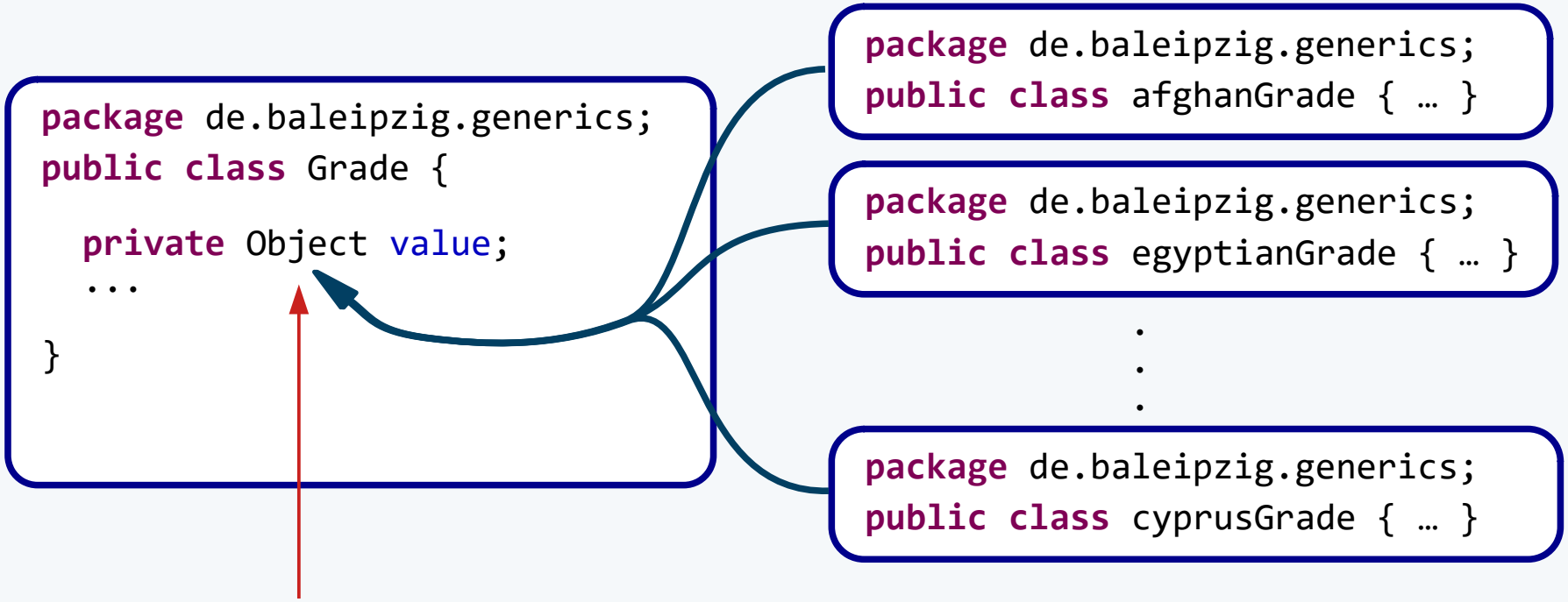
```
package de.baleipzig.generics;
public class cyprusGrade { … }
```



Bei weit über 100 Staaten auf der Erde darf das nicht die Lösung sein.

Lösungsidee

benötigt: sichere Flexibilität



Wenn wir statt einem festen Datentyp wie „Object" hier auch einen Platzhalter für unseren Datentypen benutzen könnten, den wir jeweils erst bei Benutzung der Klasse festlegen müssten, wäre das Problem gelöst.

„Generics“

„Generics“

(„Generische Programmierung“ bzw. „parametrisierte Typen“)

- Platzhalter („Typ-Parameter“) für Datentypen
- Verwendung in Klassen, Schnittstellen und Methoden

➔ Programmierung wird flexibel, ohne die Flexibilität mit stark erhöhter Fehleranfälligkeit (Typunsicherheit) zu bezahlen

„Generics“

„Generics“ - Namenskonvention

- Platzhalter („Typ-Parameter“) werden durch einen einzelnen Großbuchstaben gekennzeichnet

  (technisch ist mehr möglich, aber so kann man die „Typ-Parameter“ gut von den sonst üblichen Namen unterscheiden und die Verwechslungsgefahr ist gering)

- übliche Namen für „Typ-Parameter“:

Name	Hintergedanke (engl. Bezeichnung)
E	Element
K	Key
N	Number
T	Type
V	Value
S,U,V …	bei Verwendung mehrerer „Typ-Parameter“

„Generics“

„Generics“ - Verwendungskennzeichnung

- in Klassen

```
public class Grade<T> { … }

public class Coordinate<X,Y> { … }
```

- in Schnittstellen

```
public interface Point<T> { … }

public interface Pair<A,B> { … }
```

- in Methoden

```
public <T> void specialMethod (T t) { … }

public <A,B> boolean otherSpecialMethod (A a, B b) { … }
```

Angabe eines oder mehrerer „Typ-Parameter“ in spitzen Klammern direkt nach dem Klassen- bzw. Schnittstellennamen

Angabe der „Typ-Parameter“ vor der Methodendefinition

„Typ-Parameter“ können zusätzlich noch an dieser Stelle beschränkt werden (siehe später)

„Generics" – in Klassen

Beispiel – Generische Version unserer Klasse „Grade"

```
package de.baleipzig.generics;

public class Grade<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T aValue) {
        value = aValue;
    }

    public boolean isNotGraded() {
        return value == null;
    }

}
```

Kennzeichnung einer generischen Klasse durch Angabe eines oder mehrerer „Typ-Parameter" in spitzen Klammern direkt nach dem Klassennamen.

„Typ-Parameter" kann dann innerhalb der Klasse als Platzhalter für den bei der Instanzierung festgelegten Datentyp benutzt werden.

„Generics" – in Klassen

Beispiel – Generische Version unserer Klasse „Grade"

```
package de.baleipzig.generics;

public class Grade<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T aValue) {
        value = aValue;
    }

    public boolean isNotGraded() {
        return value == null;
    }

}
```

Kennzeichnung einer generischen Klasse durch Angabe eines oder mehrerer „Typ-Parameter" in spitzen Klammern direkt nach dem Klassennamen.

„Typ-Parameter" kann dann innerhalb der Klasse als Platzhalter für den bei der Instanzierung festgelegten Datentyp benutzt werden.

Aufgabe 1: Lassen Sie sich mit Hilfe Ihrer Entwicklungsumgebung für die Klasse eine „toString"-Methode generieren.

„Generics" – in Klassen

Beispiel – Verwendung der generischen Klasse „Grade"

```
package de.baleipzig.generics;
public class Test {
  public static void main(String[] args) {
    Grade<Double> germanGrade = new Grade<>();
    germanGrade.setValue(1.3);
    Grade<Character> swedishGrade = new Grade<>();
    swedishGrade.setValue('A');
  }
}
```

Bei Benutzung der generischen Klasse wird bei der Deklarierung wieder in spitzen Klammern ein passendes „Typ-Argument" (bzw. mehrere passende „Typ-Argumente") angegeben **[WOBEI PRIMITIVE DATENTYPEN DABEI NICHT ERLAUBT SIND]** und entsprechend werden nach dem Konstruktoraufruf mittels „new" auch spitze Klammern angegeben, die leer bleiben können, falls die dort zu verwendenden „Typ-Argumente" offensichtlich sind.

„Generics“ - Rohtypen

Beispiel – Verwendung der generischen Klasse „Grade“

```
package de.baleipzig.generics;
public class Test {
  public static void main(String[] args) {
    Grade<Double> germanGrade = new Grade<>();
    germanGrade.setValue(1.3);
    Grade<Character> swedishGrade = new Grade<>();
    swedishGrade.setValue('A');
  }
}
```

Man kann aus Kompatibilitätsgründen bei der Benutzung die spitzen Klammern samt Inhalt auch weglassen und sogenannte „Rohtypen“ benutzen:

```
Grade oldGrade = new Grade();
```

Man sollte es aber aus Gründen der Typsicherheit unterlassen, da diese Möglichkeit wirklich nur aus Gründen der Rückwärtskompatibilität besteht!

„Generics“ - Übung

Beispiel – Verwendung der generischen Klasse „Grade“

```
package de.baleipzig.generics;
public class Test {
  public static void main(String[] args) {
    Grade<Double> germanGrade = new Grade<>();
    germanGrade.setValue(1.3);
    Grade<Character> swedishGrade = new Grade<>();
    swedishGrade.setValue('A');
  }
}
```

Aufgabe 2
Erstellen Sie für weitere Tests bitte eine derartige Testklasse.

Man kann aus Kompatibilitätsgründen bei der Benutzung die spitzen Klammern samt Inhalt auch weglassen und sogenannte „Rohtypen“ benutzen:

```
Grade oldGrade = new Grade();
```

Man sollte es aber aus Gründen der Typsicherheit unterlassen, da diese Möglichkeit wirklich nur aus Gründen der Rückwärtskompatibilität besteht!

Übung – „Generics“ und „enum“

Nutzung eines „enum“ als „Typ-Argument“

```
Grade<Double> germanGrade = new Grade<>();
germanGrade.setValue(1.3);
```

Aufgabe 3

Der Datentyp „Double“ ist uns noch nicht sicher genug. Erstellen Sie beispielhaft ein „enum“ , welches zumindest:

- die Schulnoten EINS, EINS_MINUS, ZWEI_PLUS, ZWEI und FUENF samt zugehörigem Double-Wert (Instanzvariable namens „grade“),
- einen passenden Konstruktor,
- eine sinnvolle Methode „getGrade()“ und
- eine Methode „hasPassed()“, die einen passenden Wahrheitswert zurückliefert,

zur Verfügung stellt.

Nutzen Sie dieses „enum“ testweise als „Typ-Argument“ für unsere Variable „germanGrade“.

Übung – Lösung 3

Nutzung eines „enum“ als „Typ-Argument“

```
package de.baleipzig.generics;
public enum TestPrimarySchoolGrade {
  EINS(1.0), EINS_MINUS(1.3), ZWEI_PLUS(1.7), ZWEI(2.0), FUENF(5.0);
  private double grade;
  private TestPrimarySchoolGrade(double aGrade) {
    grade = aGrade;
  }
  public double getGrade() {
    return grade;
  }
  public boolean hasPassed() {
    return grade != 5.0;
  }
}
```

```
Grade<TestPrimarySchoolGrade> germanGrade = new Grade<>();
germanGrade.setValue(TestPrimarySchoolGrade.EINS);
System.out.println(germanGrade.getValue().getGrade());
System.out.println(germanGrade.getValue().hasPassed());
```

„Generics“ – in Methoden

Beispiel – Generische Version einer Methode

```
package de.baleipzig.generics;

import java.util.*;

public class Test {
  public static <E> void testGenericMethod(E element) {
    System.out.println(element.getClass().getName() + " = " + element);
  }

  public static void main(String[] args) {
    //...

  }

}
```

Aufgabe 4: Rufen Sie die statische Methode testweise auf.

„Generics“ – in Methoden (Lösung 4)

Beispiel – Generische Version einer Methode

```
package de.baleipzig.generics;

import java.util.*;

public class Test {
  public static <E> void testGenericMethod(E element) {
    System.out.println(element.getClass().getName() + " = " + element);
  }

  public static void main(String[] args) {
    //...
   testGenericMethod(11);
  }

}
```

„Generics“ - Was macht der Compiler damit?

„Generics“ – Grundsätzliche Umsetzungsvarianten

heterogen

- spezialisierter Quellcode für jeden benötigten Typ
- dementsprechend evtl. viele Klassendateien

homogen

- generalisierter Quellcode zur Abbildung aller Typen
- also nur eine Klassendatei
- entsprechende Typumwandlungen („Cast“) statt parametrisierter Anweisungen

„Generics“ - Was macht der Compiler damit?

„Generics“ – Grundsätzliche Umsetzungsvarianten

- Java nutzt die „homogene“ Umsetzung mittels sogenannter „Typlöschung“ („Type Erasure“)
- Ziel war die Kompatibilität bei Bytecode und Quellcode mit älteren Versionen (da insbesondere die „Collections“ auch ohne „Generics“ schon viel benutzt wurden)

homogen

- generalisierter Quellcode zur Abbildung aller Typen
- also nur eine Klassendatei
- entsprechende Typumwandlungen („Cast“) statt parametrisierter Anweisungen

- „Generics“ gehören nach wie vor nicht zum Typsystem der Java-Laufzeitumgebung; Typinformationen werden daher als Signatur-Attribute gespeichert, um beim Disassembeln trotzdem verfügbar zu sein, ohne bei älteren Java-Laufzeitumgebungen Probleme zu verursachen

„Generics“ - Was macht der Compiler damit?

Typlöschung („Type Erasure“)

- der „Typ-Parameter“ (ggf. samt Grenzen) wird überall durch den dort höchstmöglichen Typ ersetzt, also ohne Grenzangaben durch „Object“

```java
public class Grade<T> {
  private T value;
  public T getValue() {
    return value;
  }
  public void setValue(T aValue) {
    value = aValue;
  }
  public boolean isNotGraded() {
    return value == null;
  }
}
```

```java
public class Grade {
  private Object value;
  public Object getValue() {
    return value;
  }
  public void setValue(Object aValue) {
    value = aValue;
  }
  public boolean isNotGraded() {
    return value == null;
  }
}
```

„Generics" - Was macht der Compiler damit?

Typlöschung („Type Erasure")

- der „Typ-Parameter" (ggf. samt Grenzen) wird überall durch den dort höchstmöglichen Typ ersetzt, also ohne Grenzangaben durch „Object"
- bei der Verwendung werden entsprechend automatisch passende Typumwandlungen („Casts") eingefügt

```
Grade<Character> swedishGrade = new Grade<>();
swedishGrade.setValue('A');
Character gradeChar = swedishGrade.getValue();
```

```
Grade swedishGrade = new Grade();
swedishGrade.setValue('A');
Character gradeChar = (Character)swedishGrade.getValue();
```

„Generics“ - Was macht der Compiler damit?

„Generics“ & „Type Erasure“

```
Grade<Character> swedishGrade = new Grade<>();
swedishGrade.setValue('A');
Character gradeChar = swedishGrade.getValue();
```

```
Grade swedishGrade = new Grade();
swedishGrade.setValue('A');
Character gradeChar = (Character)swedishGrade.getValue();
```

vs. „Object“ von Anfang an

```
Grade swedishGrade = new Grade();
swedishGrade.setValue('A');
Character gradeChar = (Character) swedishGrade.getValue();
```

„Generics“ - Was macht der Compiler damit?

„Generics“ & „Type Erasure“

```
Grade<Character> swedishGrade = new Grade<>();
swedishGrade.setValue('A');
Character gradeChar = swedishGrade.getValue();
```

```
Grade swedishGrade = new Grade();
swedishGrade.setValue('A');
Character gradeChar = (Character)swedishGrade.getValue();
```

vs. „Object“ von Anfang an

```
Grade swedishGrade = new Grade();
swedishGrade.setValue('A');
Character gradeChar = (Character) swedishGrade.getValue();
```

1. Arbeit der Typumwandlung liegt ohne „Generics“ beim Programmierer

„Generics“ - Was macht der Compiler damit?

„Generics“ & „Type Erasure“

```
Grade<Character> swedishGrade = new Grade<>();
swedishGrade.setValue('A');swedishGrade.setValue(1.0);
Character gradeChar = swedishGrade.getValue();
```

Hinweis in Entwicklungsumgebung bzw. Compilerfehler, der besagt, dass die Methode für Parameter mit Typ „Double“ nicht anwendbar ist

vs. „Object“ von Anfang an

```
Grade swedishGrade = new Grade();
swedishGrade.setValue('A');swedishGrade.setValue(1.0);
Character gradeChar = (Character) swedishGrade.getValue();
```

Laufzeitfehler: Exception in thread "main" java.lang.ClassCastException: class java.lang.Double cannot be cast to class java.lang.Character...

1. Arbeit der Typumwandlung liegt ohne „Generics“ beim Programmierer
2. Laufzeitfehler statt Compilerfehler (mehr Typsicherheit)

„Generics“ - Warum sind Sie also wichtig?

„Generics“ unterstützen

- Wiederverwendbarkeit
- Typsicherheit
- Quellcode-Optimierung (weniger nötige Typumwandlungen)

„Generics“- Probleme?

Aufgrund „Type Erasure“ ist nicht alles denkbare möglich

- keine new-Instanzierung mit „Typ-Parameter“ möglich: ~~`new T();`~~
  (da würde ggf. ein Objekt statt einem String oder einem Alien erzeugt)
- Typumwandlung ggf. nicht möglich:
  ~~`Grade<String> test = (Grade<String>) new Grade<Double>();`~~
  (weil Grade test = (Grade) new Grade(); sinnlos ist)
- Ausnahmen mit „Generics“ sind nicht erlaubt:
  ~~`class MyException<T> extends Exception { ... }`~~
  (dies könnte zu catch-Blöcken mit gleichen Parametern führen, was nicht erlaubt ist)
- … (es gibt noch ein paar weitere Dinge, an denen einen der Compiler automatisch hindert)

„Generics“ - Einfache Beschränkungen

„Typ-Parameter“ können eingeschränkt werden

- auf Typen, die bestimmte Klassen erweitern oder
- auf Typen, die bestimmte Schnittstellen implementieren

- dies geschieht **für beide Fälle** durch das Schlüsselwort „extends“, wobei mehrere Beschränkungen (verknüpft durch „&“) gleichzeitig gefordert werden können

- **Beispiel:** Wir wollen vergleichbare Zensuren haben und schränken dafür zunächst unsere generische Klasse „Grade“ ein

```
public class Grade<T extends Comparable<T>> { ... }
```

könnte theoretisch noch komplizierter werden:

```
public class Grade<T extends
  eineKlasse & eineSchnittstelle & nochEineSchnittstelle> { ... }
```

„Generics“ - Einfache Beschränkungen

„Typ-Parameter“ können eingeschränkt werden

Aufgabe 5
Nutzen Sie ihr zur Notenrepräsentation erstelltes „enum“ (diese implementieren ein entsprechendes „Comparable“), um in Ihrer Testklasse eine weitere Variable zu deklarieren und initialisieren Sie diese so, dass Ihr dazugehöriges Objekt eine „ZWEI“ repräsentiert.
Vergleichen Sie dann die „value“-Eigenschaften beider Zensuren, indem Sie den „value“ der einen Zensur (mittels Getter auslesbar) als Parameter für die „compareTo“-Methode (aus der „Comparable“-Schnittstelle) angewendet auf den „value“ der anderen Zensur benutzen.

```
public class Grade<T extends Comparable<T>> { ... }
```

„Generics“ - Einfache Beschränkungen

„Typ-Parameter“ können eingeschränkt werden

```
Grade<TestPrimarySchoolGrade> germanGrade = new Grade<>();
germanGrade.setValue(TestPrimarySchoolGrade.EINS);

Grade<TestPrimarySchoolGrade> germanGrade2 = new Grade<>();
germanGrade2.setValue(TestPrimarySchoolGrade.ZWEI);

System.out.println(germanGrade.getValue()
        .compareTo(germanGrade2.getValue()));
```

Konsolenausgabe:
-3

„EINS“ ist besser als „ZWEI“ (weil wir glücklicher Weise die Werte im „enum“ so geordnet haben ;-)); eine informativere Ausgabe wäre schöner

```
public class Grade<T extends Comparable<T>> { ... }
```

„Generics“ - Einfache Beschränkungen

„Typ-Parameter“ können eingeschränkt werden

```
Grade<TestPrimarySchoolGrad
germanGrade.setValue(TestPr

Grade<TestPrimarySchoolGrad
germanGrade2.setValue(TestP

System.out.println(germanGr
        .compareTo(germanGr
```

Konsolenausgabe:
-3

„E
Wei
ein

Aufgabe 6a
Bereiten Sie in Ihrer Klasse „Grade<T>“ eine weitere generische Methode mit Rückgabewert vom Typ „String“ vor, die einen Parameter mit generischem Datentyp V entgegennimmt, wobei V „Comparable<T>“ implementieren muss.

```
public class Grade<T extends Comparable<T>> { ... }
```

„Generics“- Einfache Beschränkungen

„Typ-Parameter“ können eingeschränkt werden

Lösung 6a

```
public class Grade<T extends Comparable<T>> {
  //...
  public <V extends Comparable<T>> String compareGrades(V aValue) {

  }
}
```

Aufgabe 6b

Implementieren Sie die Methode „compareGrades“, so dass beim Aufruf durch eine Zensur mit einem gleichartigen Notenwert als Parameter ein String mit einer sinnvollen Vergleichsaussage zurückgegeben wird. Nutzen Sie dafür aus, dass für „aValue“ nun die Methode „compareTo()“ verfügbar ist und die aufrufende Zensur selbst mit „value“ eine zum Vergleich geeignete Eigenschaft besitzt. Testen Sie die Methode.

„Generics“- Einfache Beschränkungen

„Typ-Parameter“ können eingeschränkt werden

Lösung 6b

```
public <V extends Comparable<T>> String compareGrades(V aValue) {
  if (aValue.compareTo(value) > 0) {
    return aValue + " is worse than " + value;
  } else if (aValue.compareTo(value) == 0) {
    return aValue + "equals" + value;
  } else {
    return aValue + " is better than " + value;
  }
}
```

Testaufruf z.B.

```
System.out.println(germanGrade.compareGrades(germanGrade2.getValue()));
```

oder z.B.

```
System.out.println(germanGrade.compareGrades(TestPrimarySchoolGrade.FUENF));
```

„Generics“ - Übung (Komplettaufgabe)

Aufgabe 7

Schreiben Sie eine generische Klasse „Coordinate“, die mit zwei generischen Typen X und Y arbeitet.

Jede „Coordinate“ soll zwei private Eigenschaften haben:

- eine x-Koordinate („xCoordinate“) von Typ X und
- eine y-Koordinate („yCoordinate“) von Typ Y.

Lassen Sie sich alle nötigen Getter und Setter sowie einen Konstruktor von Ihrer Entwicklungsumgebung generieren.

Testen Sie Ihre Klasse, indem Sie in Ihrer Testklasse folgende Befehle ausführen:

```
Coordinate<Double, Double> firstCoordinate = new Coordinate<>(1.0, 2.0);
Coordinate<Integer, Integer> secondCoordinate = new Coordinate<>(1, 2);
Coordinate<String, String> thirdCoordinate = new Coordinate<>("oben", "rechts");
```

Zusatzaufgabe

Beschränken Sie Ihre Typen X und Y auf nummerische Werte, so dass der letzte Befehl zur Erstellung von „thirdCoordinate“ nicht mehr möglich ist.

„Generics“- Übung (Komplettaufgabe)

Lösung 7

```
package de.baleipzig.generics;
public class Coordinate<X, Y> {
    private X xCoordinate;
    private Y yCoordinate;
    public X getxCoordinate() {
        return xCoordinate;
    }
    public void setxCoordinate(X xCoordinate) {
        this.xCoordinate = xCoordinate;
    }
    public Y getyCoordinate() {
        return yCoordinate;
    }
    public void setyCoordinate(Y yCoordinate) {
        this.yCoordinate = yCoordinate;
    }
    public Coordinate(X xCoordinate, Y yCoordinate) {
        this.xCoordinate = xCoordinate;
        this.yCoordinate = yCoordinate;
    }
}
```

bei der automatischen Erstellung durch die Entwicklungsumgebung erhalten Sie leider die Variante mit identischen Variablennamen, so dass die „this“ zur Kennzeichnung der Instanzzugehörigkeit jeweils nötig sind

„Generics“ - Sonstiges

Mal angenommen …

wir schränken unsere neue Klasse etwas ein

```
public class Coordinate<X extends Number, Y extends Number> { ... }
```

und könnten nun nutzen

```
Coordinate<Double, Double> firstCoordinate = new Coordinate<>(1.0, 2.0);
Coordinate<Integer, Integer> secondCoordinate = new Coordinate<>(1, 2);
Coordinate<String, String> thirdCoordinate = new Coordinate<>("oben", "rechts");
Coordinate<Number, Number> fourthCoordinate = new Coordinate<>(1, 2);
```

Was ist, wenn wir unsere Koordinaten in einem Array sammeln möchten?

```
Coordinate<Number, Number>[] myCoordinates = new Coordinate[3];
myCoordinates[0] = fourthCoordinate;
myCoordinates[1] = secondCoordinate;   nicht typsicher!
```

das geht so nicht, „Integer“ ist zwar eine „Number“ (erbt davon),
aber „Coordinate<Integer, Integer>“ **erbt nicht** (oder haben Sie irgendwo ein „extends“ gesehen?) von „Coordinate<Number,Number>“,
ist also auch keine „Coordinate<Number,Number>“

„Generics“ - Sonstiges

Wir denken mal an ein anderes Beispiel ...

```
List<Animal> wildAnimals;
```

wild Animals:
Bären
Greifvögel
Fische
Füchse
Wolf
...

```
addAnimals
   (wildAnimals)
```

```
public void addAnimals(List<Animal> someAnimals) {
  someAnimals.add(new Wolf());
}
```

„Generics“ - Sonstiges

Wir denken mal an ein anderes Beispiel … ABER:

```
List<Sheep> professorsSheeps;
```

Professor's Sheeps:
Schaf „Anton“
Schaf „Berta“
Schaf „Carla“
Schaf „David“
Schaf „Egon“
Wolf

„Generics“ - Sonstiges

Wir denken mal an ein anderes Beispiel … ABER:

```
List<Sheep> professorsSheeps;
```

Professor's Sheeps:
Schaf „Anton“
Schaf „Berta“
Schaf „Carla“
Schaf „David“
Schaf „Egon“
Wolf

würde ich auch nicht zulassen

```
addAnimals
  (professorsSheeps)
```

```
public void addAnimals(List<Animal> someAnimals) {
  someAnimals.add(new Wolf());
}
```

„Generics" - Sonstiges

- **Typsicherheit** ist einer der Vorteile bei der Verwendung von „Generics", der nicht einfach verschenkt werden sollte

- mit normalen Arrays geht:

```
Number[] myNumbers = new Number[3];
Integer myInt = 1;
Double myDouble = 2.0;
myNumbers[0] = myInt;
myNumbers[1] = myDouble;
```

  Arrays sind in Java „kovariant",
  d.h. weil „Integer" ein Subtyp von „Number" ist,
  ist „Integer[]" auch ein Subtyp von „Number[]"

- „Generics" sind „invariant" (und nicht „kovariant"),
  eine List<Integer> ist keine List<Number> usw.

„Wildcards“

Falls ich etwas für Wölfe ODER Schafe brauche …

Um Typfamilien einsetzen zu können, gibt es „Wildcards“.

Unser altes Beispiel

```
Coordinate<Double, Double> firstCoordinate = new Coordinate<>(1.0, 2.0);
Coordinate<Integer, Integer> secondCoordinate = new Coordinate<>(1, 2);
Coordinate<Number, Number> fourthCoordinate = new Coordinate<>(1, 2);
```

Was ist, wenn wir unsere Koordinaten in einem Array sammeln möchten?

```
Coordinate<Number, Number>[] myCoordinates = new Coordinate[3];
myCoordinates[0] = fourthCoordinate;
myCoordinates[1] = secondCoordinate;   // geht nicht
```

```
Coordinate<?, ?>[] myCoordinates = new Coordinate[3];
myCoordinates[0] = fourthCoordinate;
myCoordinates[1] = secondCoordinate;   // geht
```

„Wildcards“

Falls ich etwas für Wölfe ODER Schafe brauche …

Um Typfamilien einsetzen zu können, gibt es „Wildcards“.

Beispiel mit „Collections“

```
List<Number> myNumberList;
List<Integer> myIntegerList = new ArrayList<>();
myNumberList = myIntegerList; // geht nicht
```

```
List<?> myNumberList;
List<Integer> myIntegerList = new ArrayList<>();
myNumberList = myIntegerList; // geht
```

„Wildcards“

Falls ich etwas für Wölfe ODER Schafe brauche …

Um Typfamilien einsetzen zu können, gibt es „Wildcards“.

- „Wildcards“ können zur Instanzierung parametrisierter Typen benutzt werden
- „Wildcards“ stehen für „(teilweise bestimmte) unbekannte Typen“, wobei der Typ zur Laufzeit bekannt sein muss
  - ➔ dementsprechend sind dann auch nur die für ALLE diese Typen zugelassenen Dinge erlaubt (es könnte schließlich jeder sein)

```
List<?> myNumberList;
myNumberList.add(42);
```

„?“ könnte für „Schaf“ stehen,
und 42 ist definitiv kein „Schaf“

  - ➔ „Wildcards“ können zur Variablendeklaration genutzt werden (da geht es nur um die Referenz), aber nicht zur Objekterzeugung (welches konkrete Objekt sollte gebaut werden?)
- häufig Einsatz als Argument von Methoden, selten bei Variablendeklarationen, insbesondere im Zusammenhang mit dem „Java Collections Framework“

„Wildcards“

Falls ich etwas für Wölfe ODER Schafe brauche …

Unterschied zwischen „?“ und „Object“ als Typ aller Typen am Beispiel

```
List<Object> myObjectList;
List<Integer> myIntegerList = new ArrayList<>();
myObjectList = myIntegerList; // geht nicht
myObjectList.add(42);         // geht
```

eine List<Integer> ist keine List<Object>, aber ein „Integer“ wie 42 ist ein „Object“

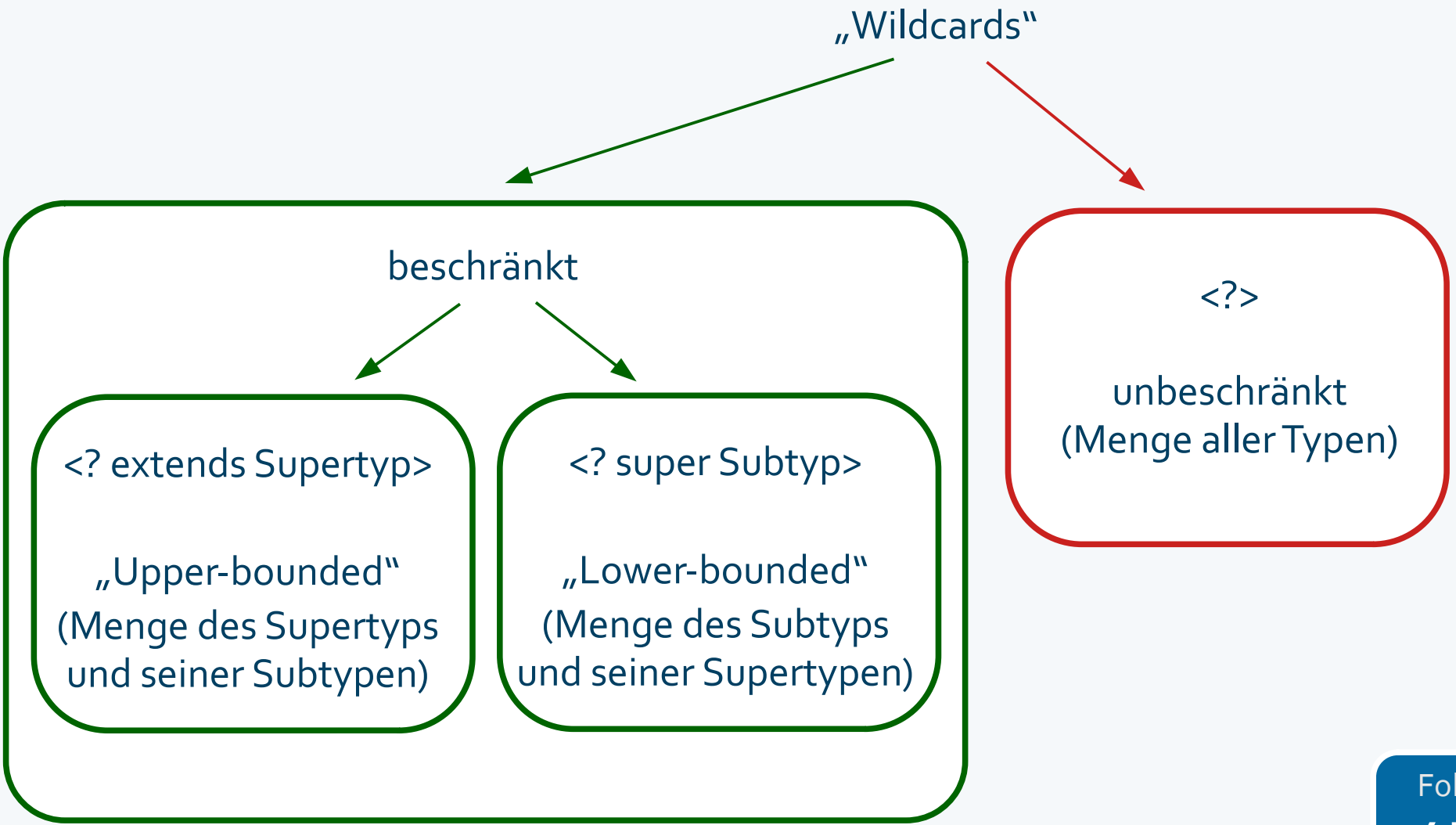
List<Object> … heterogene Liste ggf. verschiedener Objekte

```
List<?> myUnknownTypList;
List<Integer> myIntegerList = new ArrayList<>();
myUnknownTypList.add(42);          // geht nicht
myUnknownTypList = myIntegerList; // geht
```

eine List<Integer> ist eine List<?>, denn „Integer“ ist ein beliebiger Typ, aber ein „Integer“ wie 42 ist kein „?“ (kann nicht jeder Typ sein)

List<?> … homogene Liste von typgleichen Objekten unbekannten Typs

„Wildcards“ - Arten



„Wildcards“ – Arten

Vorbereitende Übungsaufgabe - Aufgabe 8

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ erweitern.

Vervollständigen Sie dafür zunächst Ihre „Coordinate“-Klasse durch den Standardkonstruktor, der ohne Argumente aufgerufen werden kann, und setzen Sie bei dessen Aufruf die beiden Koordinatenwerte jeweils auf „null“.

„Wildcards“ – Arten

Vorbereitende Übungsaufgabe - Aufgabe 8

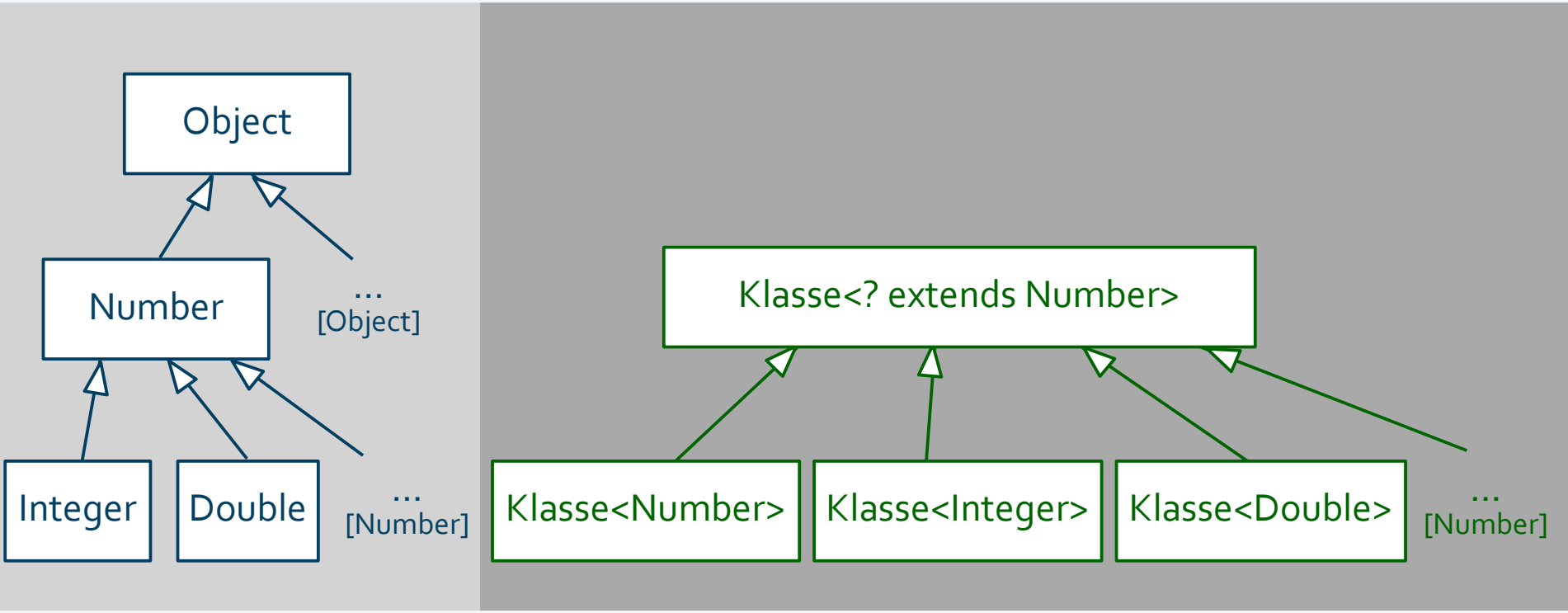
Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ erweitern.

Vervollständigen Sie dafür zunächst Ihre „Coordinate“-Klasse durch den Standardkonstruktor, der ohne Argumente aufgerufen werden kann, und setzen Sie bei dessen Aufruf die beiden Koordinatenwerte jeweils auf „null“.

```
package de.baleipzig.generics;

public class Coordinate<X extends Number, Y extends Number> {
  //...

  public Coordinate() {
    this.xCoordinate = null;
    this.yCoordinate = null;
  }

}
```

„Upper-bounded-Wildcards“

- <? extends Supertyp> - Menge des Supertyps und seiner Subtypen



- derartige Beschränkungen werden häufig für Eingangsvariable benutzt, da aufgrund der Generalisierung keine Methoden mit durch die „Wildcard“-Nutzung spezialisierten Argumenten benutzt werden können

  (die Spezial-Methode müsste bei allen „? extends Number“-Argumenten funktionieren)

„Upper-bounded-Wildcards“

Aufgabe 9a
Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine Methode „copyFrom(Coordinate<…> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der aufrufenden Koordinate durch die der übergebenen „coordinate“ ersetzt werden.

Schreiben Sie in Ihrer „Coordinate“-Klasse einen entsprechenden Methodenkopf, bei der das an die Methode übergebene Argument so eingeschränkt wird, dass ein Kopiervorgang ohne Laufzeitfehler möglich ist.

„Upper-bounded-Wildcards“

Aufgabe 9a

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine Methode „copyFrom(Coordinate<…> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der aufrufenden Koordinate durch die der übergebenen „coordinate“ ersetzt werden.

Schreiben Sie in Ihrer „Coordinate“-Klasse einen entsprechenden Methodenkopf, bei der das an die Methode übergebene Argument so eingeschränkt wird, dass ein Kopiervorgang ohne Laufzeitfehler möglich ist.

```
package de.baleipzig.generics;
public class Coordinate<X extends Number, Y extends Number> {
  //...
  public void copyFrom(Coordinate<? extends X, ? extends Y> coordinate) {

  }
}
```

„Upper-bounded-Wildcards“

Aufgabe 9b

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine Methode „copyFrom(Coordinate<…> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der aufrufenden Koordinate durch die der übergebenen „coordinate“ ersetzt werden.

Implementieren Sie den Rest der Methode.

„Upper-bounded-Wildcards“

Aufgabe 9b

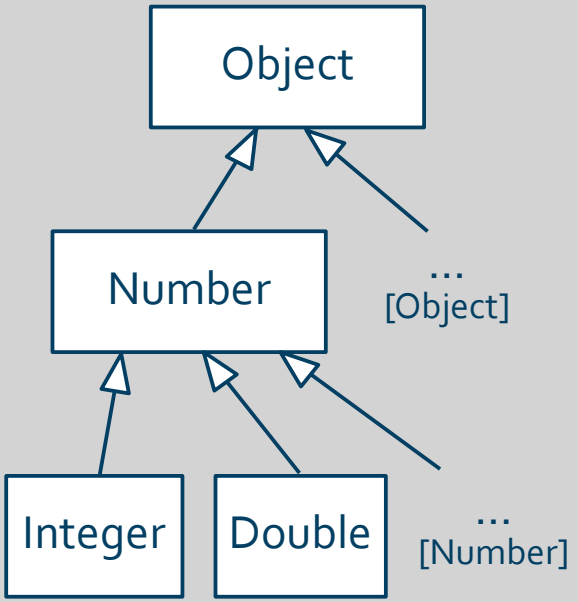
Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine Methode „copyFrom(Coordinate<…> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der aufrufenden Koordinate durch die der übergebenen „coordinate“ ersetzt werden.

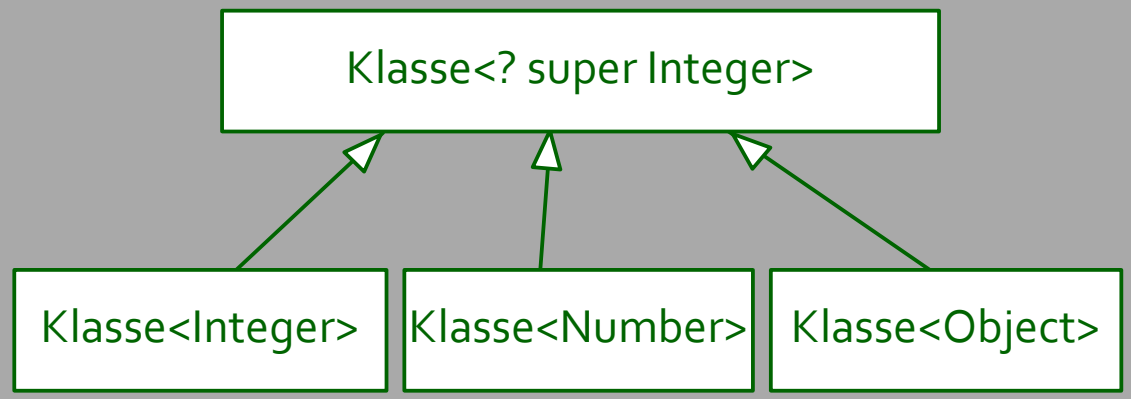
Implementieren Sie den Rest der Methode.

```
package de.baleipzig.generics;
public class Coordinate<X extends Number, Y extends Number> {
  //...
  public void copyFrom(Coordinate<? extends X, ? extends Y> coordinate) {
    xCoordinate = coordinate.getxValue();
    yCoordinate = coordinate.getyValue();
  }
}
```

„Lower-bounded-Wildcards“

- <? super Subtyp> - Menge des Subtyps und seiner Supertypen





- derartige Beschränkungen werden häufig für Ausgangsvariable benutzt, da aufgrund der Spezialisierung keine Methoden mit durch die „Wildcard“-Nutzung spezialisierten Rückgabewerten möglich sind

  (die Spezial-Methode müsste für alle „? super Integer“-Rückgabewerte funktionieren)

„Upper-bounded-Wildcards“

Aufgabe 10a

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine Methode „copyTo(Coordinate<…> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der übergebenen „coordinate“ durch die der aufrufenden Koordinate ersetzt werden.

Schreiben Sie in Ihrer „Coordinate“-Klasse einen entsprechenden Methodenkopf, bei der das an die Methode übergebene Argument so eingeschränkt wird, dass ein Kopiervorgang ohne Laufzeitfehler möglich ist.

„Upper-bounded-Wildcards“

Aufgabe 10a

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine Methode „copyTo(Coordinate<…> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der übergebenen „coordinate“ durch die der aufrufenden Koordinate ersetzt werden.

Schreiben Sie in Ihrer „Coordinate“-Klasse einen entsprechenden Methodenkopf, bei der das an die Methode übergebene Argument so eingeschränkt wird, dass ein Kopiervorgang ohne Laufzeitfehler möglich ist.

```
package de.baleipzig.generics;
public class Coordinate<X extends Number, Y extends Number> {
  //...
  public void copyTo(Coordinate<? super X, ? super Y> coordinate) {

  }
}
```

„Upper-bounded-Wildcards“

Aufgabe 10b

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine Methode „copyTo(Coordinate<…> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der übergebenen „coordinate“ durch die der aufrufenden Koordinate ersetzt werden.

Implementieren Sie den Rest der Methode.

„Upper-bounded-Wildcards“

Aufgabe 10b

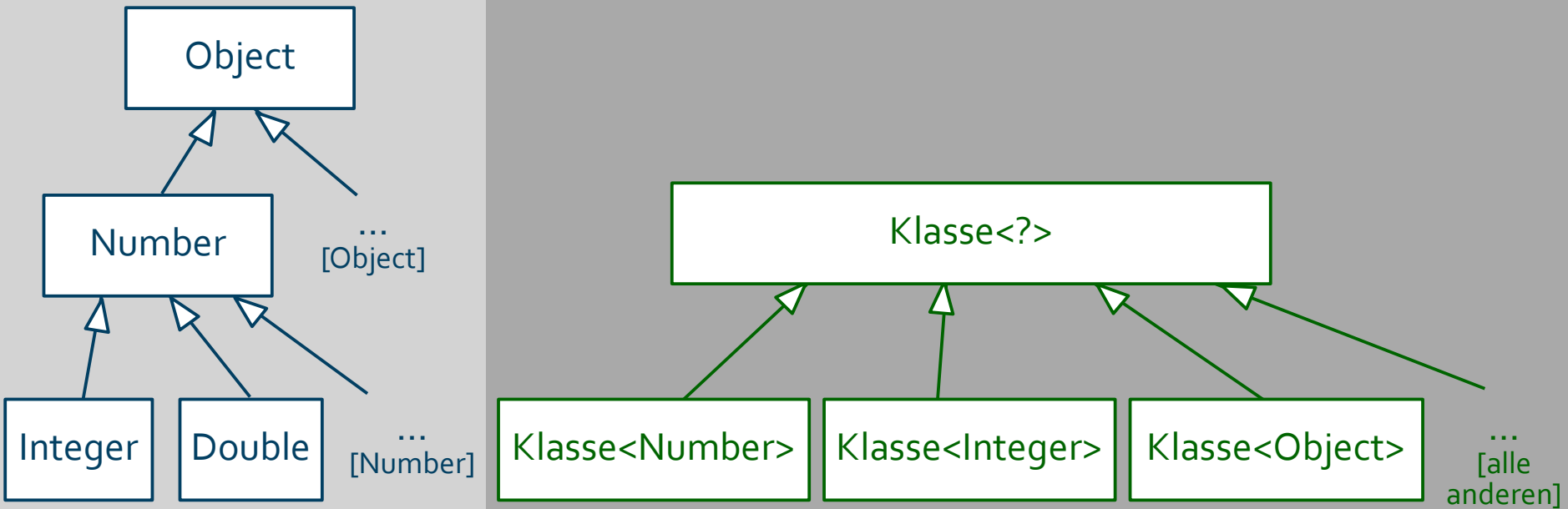
Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine Methode „copyTo(Coordinate<...> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der übergebenen „coordinate“ durch die der aufrufenden Koordinate ersetzt werden.

Implementieren Sie den Rest der Methode.

```
package de.baleipzig.generics;
public class Coordinate<X extends Number, Y extends Number> {
  //...
  public void copyTo(Coordinate<? super X, ? super Y> coordinate) {
    coordinate.setxValue(xCoordinate);
    coordinate.setyValue(yCoordinate);
  }
}
```

unbeschränkte „Wildcards“

- <?> - Menge aller Typen



- derartige Beschränkungen werden häufig für Eingangsvariable benutzt, wenn nur Methoden von „Object“ benötigt werden, da weder Methoden mit durch die „Wildcard“-Nutzung spezialisierten Argumenten noch welche mit derartig spezialisierten Rückgabewerten benutzt werden können

(bei Nutzung als Ein- und Ausgangsvariable benutze keine „Wildcard“)

„Wildcards“

Aufgabe 11a

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine statische Methode „copy(Coordinate<…> destinationCoordinate, Coordinate<…> sourceCoordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der übergebenen „destinationCoordinate“ durch die der übergebenen „sourceCoordinate“ ersetzt werden.

Schreiben Sie in Ihrer „Coordinate“-Klasse einen entsprechenden Methodenkopf, bei der die an die Methode übergebenen Argument so eingeschränkt werden, dass ein Kopiervorgang ohne Laufzeitfehler möglich ist.

„Wildcards“

Lösung 11a – mehrere Varianten

```
package de.baleipzig.generics;
public class Coordinate<U extends Number, V extends Number> {
  //...
  public static <U extends Number,V extends Number> void copy(
    Coordinate<? super U, ? super V> destinationCoordinate,
    Coordinate<? extends U, ? extends V> sourceCoordinate) {
  }
}
```

```
  //...
  public static <U extends Number,V extends Number> void copy(
    Coordinate<U, V> destinationCoordinate,
    Coordinate<? extends U, ? extends V> sourceCoordinate) {
  }
```

```
  //...
  public static <U extends Number,V extends Number> void copy(
    Coordinate<? super U, ? super V> destinationCoordinate,
    Coordinate<U, V> sourceCoordinate) {
  }
```

„Wildcards“

Aufgabe 11b

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine statische Methode „copy(Coordinate<…> destinationCoordinate, Coordinate<…> sourceCoordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der übergebenen „destinationCoordinate“ durch die der übergebenen „sourceCoordinate“ ersetzt werden.

Implementieren Sie den Rest der Methode.

„Wildcards“

Aufgabe 11b

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine statische Methode „copy(Coordinate<…> destinationCoordinate, Coordinate<…> sourceCoordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der übergebenen „destinationCoordinate“ durch die der übergebenen „sourceCoordinate“ ersetzt werden.

Implementieren Sie den Rest der Methode.

```
package de.baleipzig.generics;
public class Coordinate<X extends Number, Y extends Number> {
  //...
  public static <U extends Number, V extends Number> void copy(
    Coordinate<U, V> destinationCoordinate,
    Coordinate<? extends U, ? extends V> sourceCoordinate) {
        destinationCoordinate.setxValue(sourceCoordinate.getxValue());
        destinationCoordinate.setyValue(sourceCoordinate.getyValue());
  }
}
```

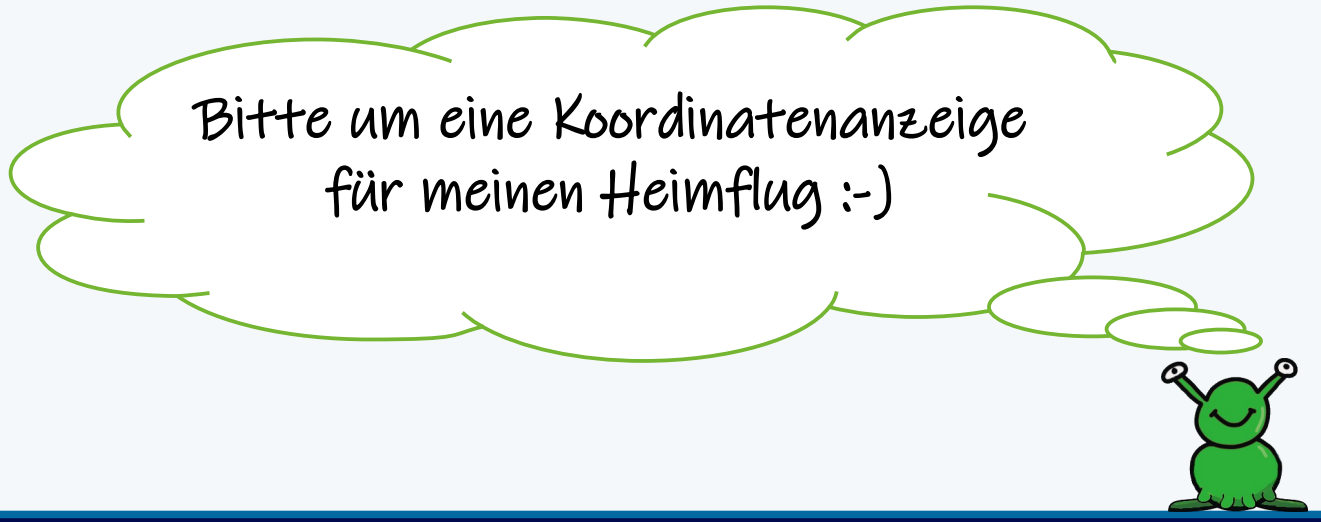
„Wildcards“ - Abschlussaufgabe

Aufgabe 12

Wir wollen unsere Klasse „Coordinate<X extends Number, Y extends Number>“ um eine statische Methode „printCoordinate(Coordinate<…> coordinate)“ erweitern, mit der die x- und y-Koordinaten-Werte der übergebenen „coordinate“ mit Vorwort „Coordinate: “ auf der Konsole ausgegeben werden.

a) Schreiben Sie in Ihrer „Coordinate“-Klasse die entsprechende Methoden.

b) Testen Sie diese Methode und mindestens eine zuvor geschriebene Methode in Ihrer Testklasse.



„Wildcards“ - Abschlussaufgabe

Lösung 12a

```
package de.baleipzig.generics;
public class Coordinate<X extends Number, Y extends Number> {
  //...
  public static void printCoordinate(Coordinate<?,?> coordinate) {
    System.out.println("Coordinate: " + coordinate.getxValue() +
      ", " + coordinate.getyValue());
  }
}
```

„Wildcards“ - Abschlussaufgabe

Lösung 12b

```
package de.baleipzig.generics;
import java.util.*;
public class Test {
  public static void main(String[] args) {
    //...
    Coordinate<Double, Double> firstCoordinate = new Coordinate<>(1.0, 2.0);
    Coordinate<Integer, Integer> secondCoordinate = new Coordinate<>(1, 2);
    Coordinate<Number, Number> copyOfSomeCoordinate = new Coordinate<>();
    Coordinate.printCoordinate(copyOfSomeCoordinate);
    copyOfSomeCoordinate.copyFrom(secondCoordinate);
    Coordinate.printCoordinate(copyOfSomeCoordinate);
    firstCoordinate.copyTo(copyOfSomeCoordinate);
    Coordinate.printCoordinate(copyOfSomeCoordinate);
    Coordinate.copy(copyOfSomeCoordinate, secondCoordinate);
    Coordinate.printCoordinate(copyOfSomeCoordinate);
  }

}
```

Vielen Dank für Ihre Aufmerksamkeit.