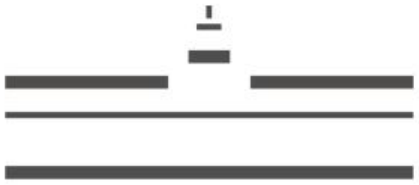

Westfälische
Wilhelms-Universität
Münster

Programmieren in Java

Generische Datentypen

wissen.leben
WWU Münster

Name: Patrick Förster, Thorsten Hovenga

Zentrum für
Informations
Verarbeitung

Generischer Typ

- In der Typ-Semantik gehören generische Datentypen in die Gruppe der „zusammengesetzten Datentypen“ (Vorlesung 2)
- Zusammengesetzten Datentypen beinhaltet Daten eines bestimmten definierten Datentyps

```
public class Person{…}
Person[] persons = …
```

- Hier wird bspw. ein Wert des Datentyps **`Array`** auf Basis des Datentyps **`Person`** definiert
- Häufig stellt man fest, dass für eine Menge an Datentypen die gleichen Operationen definiert werden, diese jedoch unabhängig vom Basistyp sind
- Angenommen man hat einen Datentyp **`Garbage`**, der eine Menge an Werten aufnimmt
- Eine *Instanz* dieses Datentyps wäre in etwa ein Mülleimer

Der Datentyp Garbage

- Die Spezifikation des Datentyps könnte in etwa wie folgt aussehen:

```
public class Garbage {
    public void add(Object object);
    public void remove(Object object);
    public void dump();
}
```

```
Garbage mülleimer = new Garbage()
mülleimer.add("TEST");
mülleimer.add(10);
mülleimer.remove("TEST");
mülleimer.dump();
```

- Ein solcher „Mülleimer“ nimmt Objekte jeden Typs auf
- Was wenn ein Mülleimer her soll, der nur spezielle Daten aufnimmt?
- Mit der aktuellen Spezifikation nicht möglich
- Die Spezifikation müsste auf den Datentyp angepasst werden
- Gleiches gilt für jeden Wunsch einen Mülleimer zu definieren der bestimmte Daten aufnehmen soll

String-Mülleimer

- Der Basistyp garantiert die Typsicherheit des Mülleimers
- Ein **`String`**-Mülleimer:

```
public class StringGarbage {
    public void add(String object);
    public void remove(String object);
    public void dump();
}
```

- Dieser Mülleimer kann nur **`Strings`** aufnehmen
- Nebenbei: Warum erbt dieser Mülleimer nicht von **`Garbage`**?

- Diese Typsicherheit ist nur nach „Außen“ wichtig, wie die Daten innerhalb der Klasse verwaltet werden ist unwichtig
- Die Daten könnten bspw. in einem **`Object`**-Array abgelegt werden
- Diese Verwaltung wäre gänzlich unabhängig vom definierten Basistyp

Beispiel

```
public class StringGarbage {
    private Object[] content = new Object[0];

    public void add(String string) {
        content = Arrays.copyOf(content, content.length + 1);
        content[content.length - 1] = string;
    }
…}
```

```
public class PersonGarbage {
    private Object[] content = new Object[0];

    public void add(Person person) {
        content = Arrays.copyOf(content, content.length + 1);
        content[content.length - 1] = person;
    }
…}
```

Beispiel (2)

- Instanzen beider Klassen können nur Daten des eigenen Basistyps aufnehmen (**`String`**, **`Person`**)
- Innerhalb der Klasse wird vom Basistyp kein Gebrauch gemacht!
- Das ist durchaus auch eine legitime Designentscheidung :
  - Ein „Nutzer“ der Klasse „sieht“ nur die für ihn öffentlichen Methoden
  - Diese lassen kein „Missbrauch“ der Mülleimers zu
  - Intern ist der „Urheber“ der Klasse verantwortlich

- Das Beispiel zeigt:
  - Könnte man vom Basistyp abstrahieren, müsste man nicht für jeden gewünschten Typ eine eigene Klasse definieren
  - Vererbung einer **`Object`**-basierten Elternklasse hilft hier nicht weiter

Typvariable

- Wie abstrahiert man vom Basistyp?
- Eine Methode abstrahiert bspw. von den von ihr genutzten Daten durch Variablen
- Erst wenn die Methode wirklich aufgerufen wird, „kennt“ diese die richtigen Daten
- Generische Datentypen werden ähnlich definiert:
  - Der Klasse, die den Datentyp beschreibt, wird ein sogenannte Typparameter hinzugefügt

```
public class GenericGarbage<T> {…}
```

  - Erst beim Erzeugen einer Instanz wird über diesen Parameter angegeben, was der Basistyp diese Instanz sein soll

```
GenericGarbage<String> mülleimer = new GenericGarbage<String>();
```

Generischer Garbage

- Es genügt nicht, lediglich die Klasse um eine Variable zu erweitern
- Im Mülleimer-Beispiel sind **remove** und **add** abhängig vom Basistyp
- Auch diese Methoden müssen der Typvariablen angepasst werden

```
public class GenericGarbage<T> {
    public void add(T object);
    public void remove(T object);
    public void dump();
}
```

- Da die Implementierung der **Garbage**-Klasse bereits ohne Bezug auf den Basistyp auskam, ist der generische Datentyp **GenericGarbage** nun fertig

```
GenericGarbage<String> mülleimer = new GenericGarbage<String>();

mülleimer.add("TEST");
mülleimer.add(10);                    // ERROR!
```

Ein Mülleimer für Mülleimer

- Die Klasse **GenericGarbage** symbolisiert einen Datentyp
- Auch generische Datentypen können als Typparameter eingesetzt werden
- Es ist also möglich einen Mülleimer zu instanziieren, der nur Mülleimer enthält, die ihrerseits nur **Strings** enthalten

```
GenericGarbage<GenericGarbage<String>> mülleimer;
mülleimer = new GenericGarbage<GenericGarbage<String>>();

GenericGarbage<String> stringeimer = new GenericGarbage<String>();

mülleimer.add(stringeimer);
```

- Dieser „Übermülleimer“ ist so allerdings sehr restriktiv
- Warum sollte er nur Mülleimer akzeptieren, die Strings beinhalten?
- Hier könnte man entweder den nicht-generischen Mülleimer als Typ nutzen oder es müsste eine Möglichkeit geben zu sagen, dass man keine Information über den Typ des geschachtelten Mülleimers benötigt

Wildcards

- Wenn einem der konkrete Typ einer generischen Variable nicht interessiert, kann diese mit „**?**“ als sogenannte Wildcard definiert werden

```
GenericGarbage<GenericGarbage<?>> mülleimer;
mülleimer = GenericGarbage<GenericGarbage<?>>()

mülleimer.add(new GenericGarbage<String>());
mülleimer.add(new GenericGarbage<Person>());
```

- Dieser Mülleimer akzeptiert alle anderen Mülleimer unabhängig vom Typ
- Wildcards können nur auf der linken Seite einer Zuweisung definiert werden!
- Rechts, bei der Instanziierung muss der konkrete Typ angeben sein
- *Hmm.. Oben wird „**?**“ auf der rechten Seite benutzt... warum geht das?*
- Die Wildcard gehört zum geschachtelten Typ
- Instanziiert wird der „Übermülleimer“ und dessen Typ ist angeben und auch konkret genug

Nicht alles ist Müll

• Nicht jeder Wert ist Müll, daher sei ein Interface eingeführt, das Werte eines bestimmten Datentyps als potentiellen Müll identifiziert

```
public interface Trashable {}
```

```
public class Bottle implements Trashable {…}
```

• Es wäre nun wünschenswert beim Gebrauch des Datentyps **Garbage** garantieren zu können, dass Daten mindestens „**Trashable**“ sind

```
GenericGarbage<Bottle> eimer1 = new GenericGarbage<Bottle>();
// ➔ OK
GenericGarbage<String> eimer2 = new GenericGarbage<String>();
// ➔ ERROR
```

• Die zweite Zuweisung soll nicht mehr möglich sein, da der Datentyp **String** nicht das Interface **Trashable** implementiert und Werte dieses Datentyps damit kein Müll sind

Generische Typen einschränken

- Die gewünschte Einschränkung muss bei der Definition des generischen Datentyps angegeben werden

```
public class GenericGarbage<T extends Trashable> {…}
```

- Die **extends** Angabe garantiert, dass Instanzen von **GenericGarbage** nur mit Typen erzeugt werden können, die mindestens **Trashable** sind
- Die Syntax ist etwas verwirrend:
  - Das Schlüsselwert **extends** wird hier sowohl für Interfaces wie auch für Klassen genutzt

- Die **Garbage**-Implementierung machte bisher keinerlei Gebrauch vom eigentlichen Basistyp
- Einschränkungen können dazu genutzt werden, eine bestimmte Grundfunktionalität zu garantieren

Basisfunktionalität durch Einschränkung

- Das **Trashable**-Interface bekommt eine neue Methode **dispose**, die ein Mülleimer aufruft, wenn dieser geleert wird (Methode: **dump**)

```
public interface Trashable {
    void dispose();
}
```

- Die Implementierung der **GenericGarbage**-Klasse wird zudem auf einen **Trashable**-Array umgestellt
- Die Einschränkung des Typparameters garantiert nun, dass jedes Objekt, welches dem Mülleimer hinzugefügt wird die Methode **dispose** bereitstellt

```
public void dump() {
    for (Trashable trashable : content) trashable.dispose();
    content = new Trashable[0];
}
```

Kovarianz

- Folgende Klasse ist eine Erweiterung von **GenericGarbage**

```
class Container<T extends Trashable> extends GenericGarbage<T> {…}
```

- Man beachte, das der Typparamter weitergereicht wird
- Generische Datentypen sind „kovariant“:
  - Bei gleichem Basistyp ist die Zuweisung einer Instanz einer erweiternden Klasse immer möglich:

```
GenericGarbage<Trashable> garbage = new Container<Trashable>();
```

  - Der Basistyp ist **Trashable** und auf beiden Seiten gleich
  - **Container** ist einer Erweiterung des erwarteten Typs links
  - Unterschiedliche Basistypen sind nicht erlaubt, selbst wenn diese in Vererbungsrelation stehen:

```
GenericGarbage<Trashable> garbge = new GenericGarbage<Bottle>(); // ERROR
```

Kurzer Ausflug: Enum

- Enums sind Teilmengen aus dem Wertebereich **int**, wobei jedes Element der Teilmenge einem bestimmten Namen zugeordnet ist
- In Java werden Enums über das Schlüsselwort **enum** definiert und sind spezielle Klasse:

```
[public|private|protected] [static] enum Color {
    YELLOW,
    BROWN,
    BLUE,
    BLACK
}
```

- Die Werte werden konsekutiv angefangen bei **0** einem **int**-Wert zugeordnet
- Ein Zugriff erfolgt also über die Punktnotation: **Color.*YELLOW***
- Werte von Color sind Instanzen der generischen Klasse **Enum**:

```
public abstract class Enum<E extends Enum<E>>
```

Enums (2)

- Über die Methoden **ordinal**() und **name**() können die zugordnete Ordnungszahl sowie der Name eines Enums abgefragt werden

```
Color anEnum = …
…
System.out.println("Das Enum " + anEnum.name() + " hat die
Ordnungszahl " + anEnum.ordinal());
```

- Enums können in **switch**-Anweisungen genutzt werden:

```
switch (anEnum) {
    case YELLOW: …
    case BLUE: …
    case BROWN: …
    case BLACK: …
}
```

- *Achtung*: Im Gegensatz zum „normalen“ Zugriff darf in der **case**-Anweisung nicht der Name der Enum-Klasse angegeben werden

Ein Enum für den Müll

- Das **`Trashable`**-Interface wird nun erweitert
- Das neue Interface **`Sortable`** soll sortierbaren Müll repräsentieren
- Dazu wird ein Enum genutzt, um jeder Instanz einer **`Sortable`**-Klasse entsprechend der deutschen Mülltrennung eine bestimmte Farbe zuzuweisen

```
public interface Sortable extends Trashable {
    public static enum Color {
        YELLOW,
        BROWN,
        BLUE,
        BLACK
    }

    Color getColor();
}
```

Wildcards mit Einschränkung

- Eine weitere Klasse **`Separator`** soll nun beliebige Mülleimerinhalte entsprechend ihrer Farbe mit Hilfe einer Methode sortieren
- Bei der Implementierung einer solchen Methode kommt einem die Kovarianz in die Quere:

```
public void separate(GenericGarbage<Sortable> garbage);
```

- So könnte die Methode nur mit Mülleimern aufgerufen werden, die auf Basis des Datentyps **`Sortable`** erstellt wurden
- Die Vorteile des generischen Datentyps wäre so verloren
- Ähnlich wie bei den Typvariablen können auch Wildcards mit Einschränkung definiert werden

```
public void separate(GenericGarbage<? extends Sortable> garbage);
```

- Jetzt akzeptiert die Methode jeden Mülleimer, der seinerseits Inhalte vom Typ **`Sortable`** unterstützt

Typlöschung

- Generische Datentypen sind ein Konstrukt, das erst mit Java 5 Einzug hielt
- Das Laufzeitsystem kennt schon allein auch Gründen der Interoperabilität kein generischen Typsystem
- Der Compiler löscht beim Kompilieren eines Programmes alle Typinformationen
- Typvariablen werden durch **`Object`** bzw. dem **`extends`**-Typ ersetzt und an den entsprechenden Stellen durch Casts in den gewünschten Typ gewandelt

```
GenericGarbage<Bottle> mülleimer = new GenericGarbage<Bottle>();
müll.add(new Bottle());
Bottle bottle = müll.find();
```

beispielhaft

```
Garbage mülleimer = new Garbage();
müll.add(new Bottle());
Bottle bottle = (Bottle) müll.find();
```

Typlöschung und ihre Konsequenzen

- Die Typlöschung für zu einer Reihe von Einschränkungen:
  - Es können keine Objekte nur mit Basis der Typvariablen erstellt werden: : ~~`new T()`~~
  - Bei einfachen uneingeschränkten Typvariablen können Methoden nicht mit Object überladen werden:

```
public class SimpleGarbage<T> {
        public void add(T object);
        public void add(Object object);
}
```

  - Typvariablen gehören zu Instanzen und sind daher im statischen Kontext innerhalb einer Klasse nicht valide
  - Es können keine generischen Arrays angelegt werden, da der Compiler schon zur Compilezeit den genauen Typ des Arrays wissen möchte

Aufgabe

- Teil 1:
  - Implementiert die Klasse **`Separator`** wie auf Folie 18 beschrieben
  - Die Klasse soll zudem vier interne Attribute vom Typ **`GenericGarbage<Sortable>`** haben, auf die der Müll je nach Farbe verteilt wird
  - Zudem soll die Klasse eine Methode **`getGargabe(Sortable.Color color)`** haben, die den zur Farbe passenden Mülleimer aus dem zweiten Punkte liefert
- Teil 2:
  - Implementiert eine Klasse **`RecylceFactory`**, die eine Methode **`RecycledItem[] recylce(Separator separator)`** hat
  - Diese Methode soll anhand von Blaupausen (Klasse **`Blueprint`**) den Inhalten der Mülleimer des gelieferten Separator recyceln
  - Jede Blaupause erwartet dabei eine gewissen Anzahl an gelbem, blauem, braunem und schwarzem Müll

Aufgabe

- Solange der Separator genügend Müll hat, um eine Blaupause zu erfüllen, soll recycelt werden
- Ist genügend Müll für eine bestimmte Blaupause vorhanden soll eine Instanz der Klasse **`RecycledItem`** erstellt werden und dem Rückgabe-Array der Methode hinzugefügt werden

- Die Klassen **`Blueprint`**, **`RecycledItem`**, **`GenericGarbage`** werden gestellt
- Zudem eine Klasse **`RandomTrash`**, deren Instanzen randomisiert einer bestimmten Farbe entsprechen