Inhalt

Motivation

Ziel: Klasse Kiste, die nur ein Attribut (von beliebigem Typ) mit jeweils einer get- und einer set-Methode bereitstellt

- Kiste für Integer
- Kiste für String
- Kiste für Person
- Kiste für Punkt2D
- ...

Zwei „naive“ Lösungsansätze

1. Für jeden „Basistyp“ eine eigene Klasse
   Vorteil: absolute Typsicherheit
   Nachteil: sehr viel sehr ähnlicher Code
2. nur eine Klasse: Kiste für Object
   Vorteil: keine Code-Duplizierung
   Nachteil: keine Typsicherheit, explizites Casting nötig

Beispiel (nicht gut): ObjectKiste

```
// Definition
public class ObjektKiste {
    private Object inhalt;
    public void set(Object o){this.inhalt = o;}
    public Object get(){return inhalt;}
}

// Nutzung der Klasse:
public static void testObjektKiste() {
    ObjektKiste k = new ObjektKiste();
    k.set("hallo");
    String s = (String) k.get(); // cast notwendig
    int i = (Integer) k.get();  // fuehrt zu Laufzeitfehler
}
```

Besser: Generische Klasse

- Verwende bei der Definition der Klasse einen Typ-Parameter (Typvariable)
  $\rightsquigarrow$ Klasse braucht nur einmal definiert zu werden: keine Code-Duplizierung
- Definiere bei der Nutzung der Klasse, welcher konkrete Typ als Argument für den Typparameter verwendet werden soll
  $\rightsquigarrow$ Compiler kann Typsicherheit prüfen
- Bemerkung:
  Als Argumente kommen nur Referenztypen in Frage, keine simple types
  daher: „Wrapper“-Klassen `Integer`, `Double` ...
  statt **int**, **double** ...

Beispiel: (Generische) Kiste

```
// Definition
public class Kiste<T> {
    private T inhalt;
    public void set(T o){this.inhalt = o;}
    public T get(){return inhalt;}
}

// Nutzung der Klasse:
public static void testKiste() {
    Kiste<String> k = new Kiste<>();
    k.set("hallo");
    String s = k.get(); // kein cast notwendig
    //int i = k.get(); // wird bereits vom Compiler bemerkt
}
```

Zwei Kisten

```
public static void testZweiKisten() {
    Kiste<String> k1 = new Kiste<>();
    Kiste<Integer> k2 = new Kiste<>();
    k1.set("hallo");
    k2.set(123);
    String s = k1.get();
    int i = k2.get();
}
```

Typlöschung

Was passiert?

1. der Compiler prüft bereits zur Übersetzungszeit(!), ob Zuweisungen und Zugriffe typkonform sind
2. ersetzt an allen Stellen den Typparameter T durch Object
   $\rightsquigarrow$ „Typlöschung“
3. und ergänzt bei Zuweisungen die expliziten casts
4. es wird nur eine `.class`-Datei erzeugt (Raw-Type)

Einschränkungen

- T kann nicht als Parameter von statischen Methoden genutzt werden.

```
public static void meth(T v) { /*geht nicht */}
public static T meth() {/* geht auch nicht */}
```

- Es können keine Objekte vom Typ T erzeugt werden.

```
T elem = new T(); /*geht nicht */
```

- Es kann nicht der instanceof -Operator verwendet werden

```
if (x instanceof Kiste<T>) /*geht nicht */
if (x instanceof Kiste<Integer>) /*geht auch nicht */
```

- Methoden können nicht mit anderen Typen überladen werden

```
public void set(String s) {/*geht nicht */}
```

Arrays von T

- Es können zwar in einer generischen Klasse Array-Variablen mit Basistyp T definiert werden:

```
T[] feld; // das geht
```

- ... aber es können keine Arrays von diesem Typ erzeugt werden:

```
feld = new T[3]; // geht nicht
```

Ausweg: explizites Casten

```
T[] feld;
feld = (T[]) new Object[3];
```

Das erzeugt eine Compiler-Warnung „Type safety: Unchecked cast ...“ die aber mit @SuppressWarnings(''unchecked'') unterdrückt werden kann.

Mehrere Typparameter

Auch möglich:
Klasse mit mehreren (verschiedenen) Typparametern.
Zum Beispiel Paar:
Nicht nur

- Paare von Integern
- Paare von Personen
- Paare von ...

sondern auch

- Paare von String und Integer (Begriff, Seitennummer)
- Paare von Stud und String (Person, Studiengang)
- Paare von Integer, Double (Matr.Nr, Note)
- Paare von Personen und Personenpaar (Person, Eltern)
- ...

Inhalt

Beispiel: Generische Interfaces

Auch Interfaces können generisch sein, zum Beispiel:

```
public interface Setzbar<T> {
    void set(T o);
    T get();
    void reset();
}
```

Zur Implementierung gibt es prinzipiell zwei Möglichkeiten:

- Die implementierende Klasse ist ebenfalls generisch (mit derselben Typvariablen T).
- Die implementierende Klasse ist nicht generisch, sondern implementiert das Interface für einen konkreten Typ.

Beispiel: Generische implementierende Klasse

Die generische Klasse Kiste<T> implementiert das Interface Setzbar<T>:

```
public class Kiste<T> implements Setzbar<T>{
    private T inhalt;

    // weiter wie oben, zusaetzlich:

    @Override
    public void reset() {
        this.inhalt = null;
    }
}
```

Beispiel: Nicht-Generische implementierende Klasse

Die Klasse `Spieler` implementiert das Interface Setzbar<String>:

```
public class Spieler implements Setzbar<String>{
    private String nickname;

    @Override
    public void set(String n){
        this.nickname = n;
    }

    @Override
    public String get() {
        return this.nickname;
    }

    @Override
    public void reset() {
        this.nickname = "dummy";
    }
}
```

Inhalt

Motivation

- Schon gesehen: Einschränkungen bei generischen Klassen
  „T kann nicht als Parameter von statischen Methoden genutzt werden.“
- Trotzdem zum Beispiel gewünscht:
  statische Methoden zum Umgang mit Arrays, wie etwa
  - print-Ausgabe des Inhalts eines Arrays
  - Initialisierung eines Arrays mit Default-Werten
  - Ermittlung des „mittleren“ Elementes eines Arrays (Element an mittlerer Position)
  - zufällige Auswahl eines Elementes eines Arrays

Beispiel: Zufällige Auswahl

```
static Random r = new Random();

public static <T> T zufall(T[] feld) {
    int pos = r.nextInt(feld.length);
    return feld[pos];
}
```

Aufruf zum Beispiel durch

```
String[] feld = {"anna", "ben", "chris"};
String n = zufall(feld);
System.out.println(n);
```