Universität Bremen

Praktische Informatik 2
Generische Programmierung

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für
Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

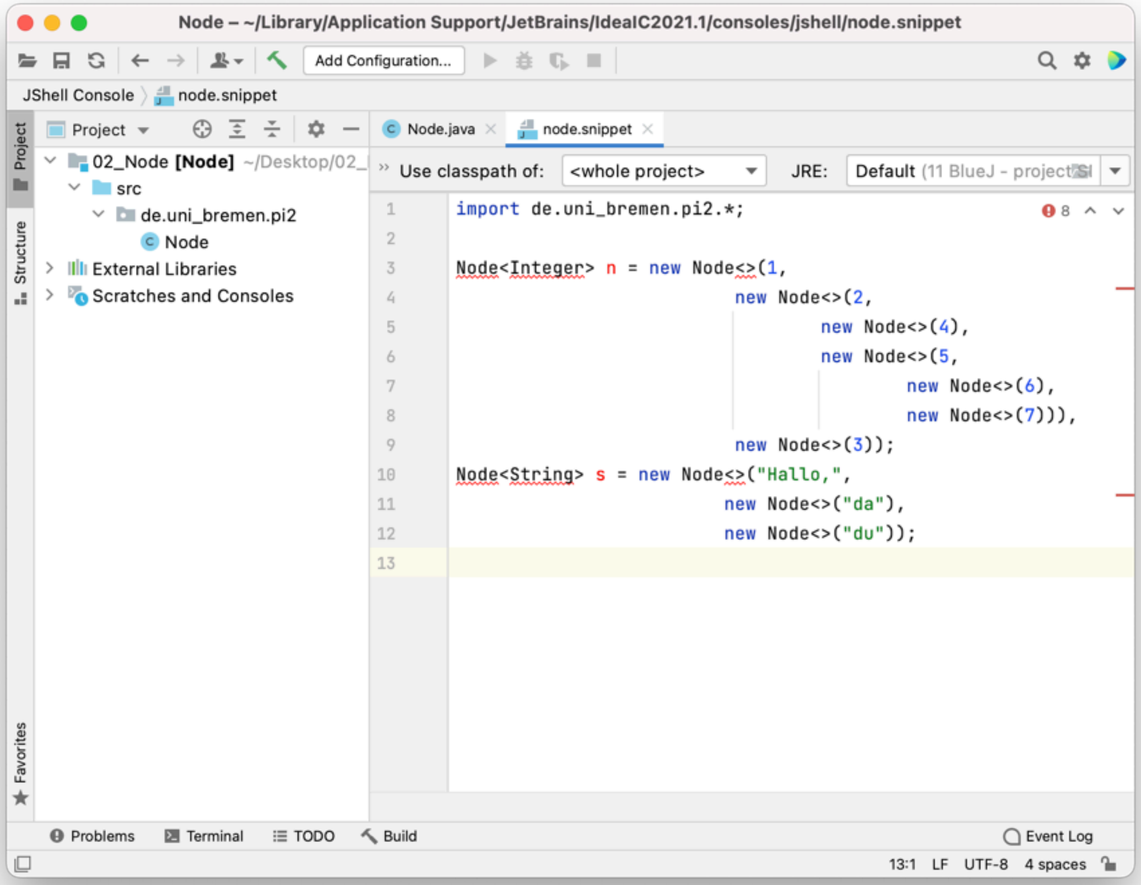
Universität Bremen

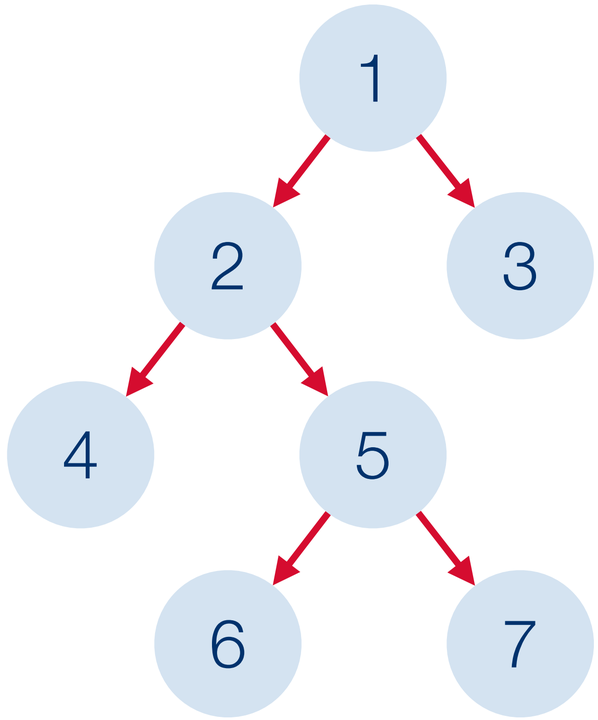
Motivation

```
final List<String> a = new ArrayList<>();
a.add("PI 2");
final String s = a.get(0);
```

- Sammlungen sind Klassen, die Instanzen anderer Klassen speichern können
- Sie sind recht unabhängig von den in ihnen gespeicherten Daten
- Oft sollen aber nur Instanzen eines bestimmten Typs in einer Sammlung gespeichert werden → **Typparameter**
- Sammlungen können bestimmte Anforderungen an die in ihnen gespeicherten Daten stellen → **Typebounds**
- Typen mit Typparametern heißen **generische Typen** und sind nicht auf Sammlungen beschränkt

Generische Klassen: Demo





Universität Bremen

Generische Klassen

- Generische Klassen haben **Typvariablen**, die im Klassenrumpf verwendet werden können
- Typvariablen können (fast) wie ein normaler Typ benutzt werden
- Mehrere Typvariablen sind möglich

```
class Pair<F, S>
{
    private final F f;
    private final S s;
    Pair(final F ff, final S ss) {f = ff; s = ss;}
    F getFirst() {return f;}
    S getSecond() {return s;}
}
```

```
class Node<E>
{
    private E info;
    private final Node<E> left, right;
    Node(final E i) {this(i, null, null);}
    Node(final E i, final Node<E> l, final Node<E> r)
    {
        info = i; left = l; right = r;
    }
    E getInfo() {return info;}
    Node<E> getLeft() {return left;}
    Node<E> getRight() {return right;}
    void setInfo(final E i) {info = i;}
}
```

Universität Bremen

Generische Schnittstellen

- Funktionieren analog zu generischen Klassen
- Können bei der Verwendung mit konkreten Typparametern versehen werden oder mit generischen
  - Genauso wie beim Erben von einer generischen Klasse

```
abstract class Medium implements Iterable<String>
{ //...
    @Override
    public Iterator<String> iterator()
    {
        final String[ ] zeilen = toString().split("\n");
        return Arrays.asList(zeilen).iterator();
    }
```

```
public class LinkedList<E> extends AbstractSequentialList<E> implements Queue<E> // ...
```

Universität Bremen

Generische Typen

- **Generische Klasse** oder **generische Schnittstelle** ist eine Klassen- bzw. Schnittstellendefinition, in der unbekannte Typen durch Typvariablen vertreten sind

```
public class Node<E>
{
    private E info …
```

- Ein **generischer Typ** ist eine Typangabe, in der eine generische Klasse mit einem konkreten Typargument versehen wird

```
Node<Integer> n = new Node<Integer>(23);
Pair<Integer, String> p = new Pair<Integer, String>(28359, "Bremen");
```

- **Diamond-Operator** (Java 7) oder **var** (nur lokale Variablen, Java 10)

```
Pair<Integer, String> p = new Pair<>(28359, "Bremen");
```

```
var p = new Pair<Integer, String>(28359, "Bremen");
```

Universität Bremen

Typebounds

- Manchmal sollen nicht alle Typen für Belegung der Typvariablen zulässig sein, z.B. wenn es Anforderungen an Typargumente gibt

```
class Node<E extends Number> { ...
```

- Ein **Typebound** bedeutet „ist kompatibel zu“
- Typebounds können Klassen und Interfaces sein (Auch bei Interfaces wird hier **extends** verwendet)
- Mit **&** können mehrere Typebounds pro Typvariable aufgezählt werden (erster Klasse oder Interface, weitere nur Interfaces)

```
class Node<E extends A & B & C> { …
```

- Typebounds können wiederum die Typvariable enthalten

```
class Node<E extends Comparable<E>> { …
```

Universität Bremen

Typebounds: Beispiel

```
class Node<E extends Number>
{
                        :
    double getSum()
    {
        return (left == null ? 0 : left.getSum()) +
                info.doubleValue() +
                (right == null ? 0 : right.getSum());
    }
}
```

```
Node<Integer> i = new Node<Integer>(23); // ok
Node<String> s = new Node<String>("PI 2"); // Fehler!
```

Universität Bremen

Invarianz generischer Typen

- Jede generische Klasse erzeugt viele generische Typen, z.B erzeugt **Node<E>** **Node<Number>**, **Node<Integer>**, **Node<Double>**…

- Wie stehen die von einer generischen Klasse erzeugten Typen zueinander?

  - Eine Ableitungsbeziehung zwischen Typargumenten überträgt sich **nicht** auf die generischen Typen

  - Dies wird als **Invarianz** bezeichnet

```
class Integer extends Number { …
Number n = new Integer(23); // ok
Node<Number> nn = new Node<Integer>(23); // Fehler!
```

Universität Bremen

Bivarianz generischer Typen

- **Wildcard** bei Typangaben
- Generische Typen können unbestimmte Typargumente nennen
- Kein Zugriff auf Methoden oder Attribute, die Typargument verwenden
  - Ausnahmen: **Object** lesen und **null** schreiben (weil das immer geht)

```
static int count(final Node<?> n)
{
    if (n == null) {
        return 0;
    }
    else {
        return 1 + count(n.getLeft())
                 + count(n.getRight());
    }
}
```

```
Node<?> nx;
nx = new Node<String>("foo");
nx = new Node<Integer>(1);
nx = new Node<Double>(3.14);
```

```
double d = nx.getInfo(); // Fehler
nx.setInfo(2.72); // Fehler
```

Universität Bremen

Covarianz generischer Typen

- Zu einem Wildcard-Typ mit Typargument **?** sind alle generischen Typen der betreffenden generischen Klasse kompatibel

- Manchmal soll dies durch einen sog. **Upper-Typebound** eingeschränkt werden (der **allgemeinste** erlaubte Typparameter)

```
Node<? extends Number> nb;
nb = new Node<Integer>(23);
nb = new Node<Object>(new Object()); // Fehler!
```

```
static double sum(final Node<? extends Number> n)
{
    return n == null ? 0 : n.getInfo().doubleValue()
            + sum(n.getLeft()) + sum(n.getRight());
}
```

Universität Bremen

Covarianz generischer Typen

- **Upper-Typebounds** eröffnen **Covarianz** für generische Typen
- Allgemein gilt: **C<A>** ist kompatibel zu **C<? extends B>**, wenn
  A ist kompatibel zu **B**
- **Problem aus Java-Anfangszeiten**: Arrays kennen Covarianz, aber statische Typprüfung versagt

```
Integer[ ] i = new Integer[23];
Number[ ] n = i; // Erlaubt
n[0] = 3.14; // ArrayStoreException
```

- **Lösung bei generischen Typen**: (Fast) nur Lesen erlaubt

```
Node<? extends Number> nb = new Node<Integer>(23);
Number n = nb.getInfo();
nb.setInfo(3.14); // Fehler!
```

Universität Bremen

Contravarianz generischer Typen

- Manchmal soll ein Typparameter durch einen sog. **Lower-Typebound** eingeschränkt werden (der **speziellste** erlaubte Typparameter)

```
static int subNodes(final Node<? super Integer> n)
{
    if (n == null) {
        return 0;
    }
    else {
        final int s = subNodes(n.getLeft()) + subNodes(n.getRight());
        n.setInfo(s);
        return s + 1;
    }
}
```

```
Node<? super Number> nb;
nb = new Node<Object>(new Object());
nb = new Node<Integer>(23); // Fehler!
```

```
static <T> void sort(T[ ] a, Comparator<? super T> c)
```

Universität Bremen

Contravarianz generischer Typen

- **Lower-Typebounds** eröffnen **Contavarianz** für generische Typen
- Allgemein gilt: **C<A>** ist kompatibel zu **C<? super B>**, wenn
  B ist kompatibel zu **A**
- (Fast) nur Schreibzugriff
  - **nb.info** ist vom Typ **Number** oder Basisklasse davon
  - An **nb.info** können also nur Objekte vom Typ **Number** oder Ableitungen davon zugewiesen werden (**nb.info** könnte vom Typ **Number** sein)
  - **nb.info** kann aber nur an Referenzen vom Typ **Object** zugewiesen werden (**nb.info** könnte vom Typ **Object** sein)

```
Node<? super Number> nb =
        new Node<Object>(new Object());
nb.setInfo(1);
Object o = nb.getInfo();
Number n = nb.getInfo(); // Fehler!
```

Universität Bremen

Zusammenfassung Varianzen

- **Nicht lesen**: Lesen von **Object** ist dennoch möglich
- **Nicht schreiben**: Schreiben von **null** ist dennoch möglich

	Typ	Lesen	Schreiben	Kompatible Typargumente
Invarianz	**C<T>**	ja	ja	**T**
Bivarianz	**C<?>**	nein	nein	Alle
Covarianz	**C<? extends B>**	ja	nein	**B** und abgeleitete Typen
Contravarianz	**C<? super B>**	nein	ja	**B** und Basistypen

Universität Bremen

Übersetzung generischer Klassen

- **C++**: Getrennte Übersetzung jeder Instanziierung generischer Klassen
  - Jeweils Ersetzen der Typparameter durch aktuelle Belegung und Übersetzen des Ergebnisses
  - Langsameres Übersetzen und größere Kompilate, aber bessere Optimierungsmöglichkeiten und weniger Einschränkungen
- **Java**: Umwandlung jeder generischen Klasse in eine nicht-generische
  - Nutzung der Typparameter nur zur Typüberprüfung
  - Schneller kompilierbar, weniger flexibel

Universität Bremen

Übersetzung generischer Klassen in Java

- Generische Datentypen werden in Java ausschließlich vom Compiler verarbeitet
- Die JVM weiß nichts von generischen Datentypen
- Mit **Type-Erasure** wird „generischer Code“ mit Typvariablen und Typargumenten auf normalen, nicht-generischen Java-Quelltext reduziert
- Der nicht-generische Java-Quelltext wird weiterverarbeitet wie bisher
- Aus jeder generischen Klasse wird **eine** nicht-generische Klasse generiert und in eine **.class**-Datei übersetzt

Universität Bremen

Type-Erasure für generische Klassen

- Typ-Variablen in spitzen Klammern werden gelöscht
- Alle Vorkommen von Typvariablen mit einem oder mehreren **Typebounds** werden durch den einzigen bzw. ersten **Typebound** ersetzt
- Alle Vorkommen von Typvariablen ohne **Typebounds** werden durch **Object** ersetzt

Universität Bremen

Type-Erasure: Beispiel

Generische Klasse

```
class Node<E> // eigentlich <E extends Object>
{
    private E info;
    private final Node<E> left, right;

    Node(final E i) {this(i, null, null);}
    Node(final E i, final Node<E> l, final Node<E> r)
    {
        info = i; left = l; right = r;
    }
    E getInfo() {return info;}
    Node<E> getLeft() {return left;}
    Node<E> getRight() {return right;}
    void setInfo(E i) {info = i;}
}
```

Nach Type-Erasure (Rawtype)

```
class Node
{
    private Object info;
    private final Node left, right;

    Node(final Object i) {this(i, null, null);}
    Node(final Object i, final Node l, final Node r)
    {
        info = i; left = l; right = r;
    }
    Object getInfo() {return info;}
    Node getLeft() {return left;}
    Node getRight() {return right;}
    void setInfo(Object i) {info = i;}
}
```

Universität Bremen

Type-Erasure für generische Typen

- Die Typ-Korrektheit wird statisch geprüft (d.h. zum Übersetzungszeitpunkt)
  - Typargumente müssen allen **Typebounds** genügen
  - Generische Typen müssen auch untereinander korrekt verwendet werden, insbesondere bei **Wildcard**-Typen
- Typargumente in spitzen Klammern werden gelöscht (Ergebnis: **Rawtype**)
- **Typecasts** werden eingeschoben, wo der Wert eines Typarguments benutzt wird
- **Rawtypes** lassen sich auch direkt benutzen, aber die Typsicherheit geht verloren (optionale Compiler-Warnung)

Universität Bremen

Type-Erasure: Beispiel

Generische Typen

```
Node<String> n = new Node<String>("foo");
String s = n.getInfo();
```

Nach Type-Erasure

```
Node n = new Node("foo");
String s = (String) n.getInfo();
```

Grenzen generischer Typen

```
class Container<E>
{
   E[ ] a = new E[100]; // Fehler
}
```

- Arrays von Elementen der Typvariablen
  - Ausweg:
  - Wenn aktiviert, Warnung „uses unchecked or unsafe operations“
  - Es können dennoch typsichere Klassen erzeugt werden
- Arrays von Elementen mit generischem Typ

```
class Container<E>
{
    E[ ] a = (E[ ]) new Object[100];  // optionale Warnung
}
```

```
class Container<E>
{
    @SuppressWarnings("unchecked")
    private E[ ] a = (E[ ]) new Object[100];
    void set(final int i, final E t) { a[i] = t; }
    E get(final int i) { return a[i]; }
}
```

```
@SuppressWarnings("unchecked") Node<E>[ ] children = new Node[2];
```

Universität Bremen

Grenzen generischer Typen

- Primitive Typargumente

```
Node<int> ni = new Node<int>(2); // Fehler!
```

```
Node<Integer> ni = new Node<Integer>(2); // Autoboxing
```

- Statische Elemente
  - Alle Klassen **Broken<T>** teilen sich Klassenattribut **data**. Welchen Typ soll es haben?

```
class Broken<T>
{
    static T data; // Fehler!
}
```

- Dynamische Typprüfung
  - Type-Erasure: **o instanceof E** → **o instanceof Object**

```
class Node<E>
{
    boolean isCompatible(final Object o)
    {
        return o instanceof E;
    }
} // Fehler!
```

Universität Bremen

Grenzen generischer Typen

- Konstruktoraufrufe
  - Wie soll Java zur Laufzeit wissen, welcher Konstruktor aufzurufen ist?
  - Ausweg: Konstruierte Objekte an generische Klasse übergeben
- Typparameter als Superklasse
  - Generische Klasse muss Konstruktor der Superklasse aufrufen können, welcher unbekannt ist

```
class Node<E>
{
    E info;
    Node() { info = new E(); } // Fehler
}
```

```
Node(final E i) { info = i; } ...
Node<Integer> ni = new Node<Integer>(2);
```

```
import java.util.Date;
class Timestamped<T> extends T // Fehler
{
    Date timestamp = new Date();
}
```

Grenzen generischer Typen

- Typecasts
  - Type-Erasure: **(E) o → (Object) o**
  - Compiler erzeugt: „warning: unchecked cast of type E“

```
void setInfo(Object o) { info = (E) o; } // sinnlos
```

- Exceptions
  - Alle generischen Exceptions sind gleich nach Type-Erasure:
    catch (UniversalException<String> e) → catch (UniversalException e)
  - Compiler erzeugt: „a generic class may not extend java.lang.Throwable“

```
class UniversalException<T> extends Exception
{
    T reason;
} // Fehler
```

Generische (polymorphe) Methoden

```
<T> T last(final T[ ] a)
{
    return a[a.length - 1];
}
```

```
String[ ] s = {"Hallo", "Welt"};
System.out.println(last(s));
```

- Polymorphe Methoden sind unabhängig von generischen Klassen
  - Sie können auch in nicht-generischen Klassen definiert werden
  - Die Belegung der Typparameter wird normalerweise automatisch ermittelt
- Klassen- und Objektmethoden sowie Konstruktoren können polymorph sein
- Beim Aufruf mit expliziter Angabe des Typparameters muss immer ein Punkt vor dem Typparameter stehen (**Klassenname.<Typ>methode(…)**, **referenz.<Typ>methode(…)**)

```
System.out.println(this.<String>last(s));
```

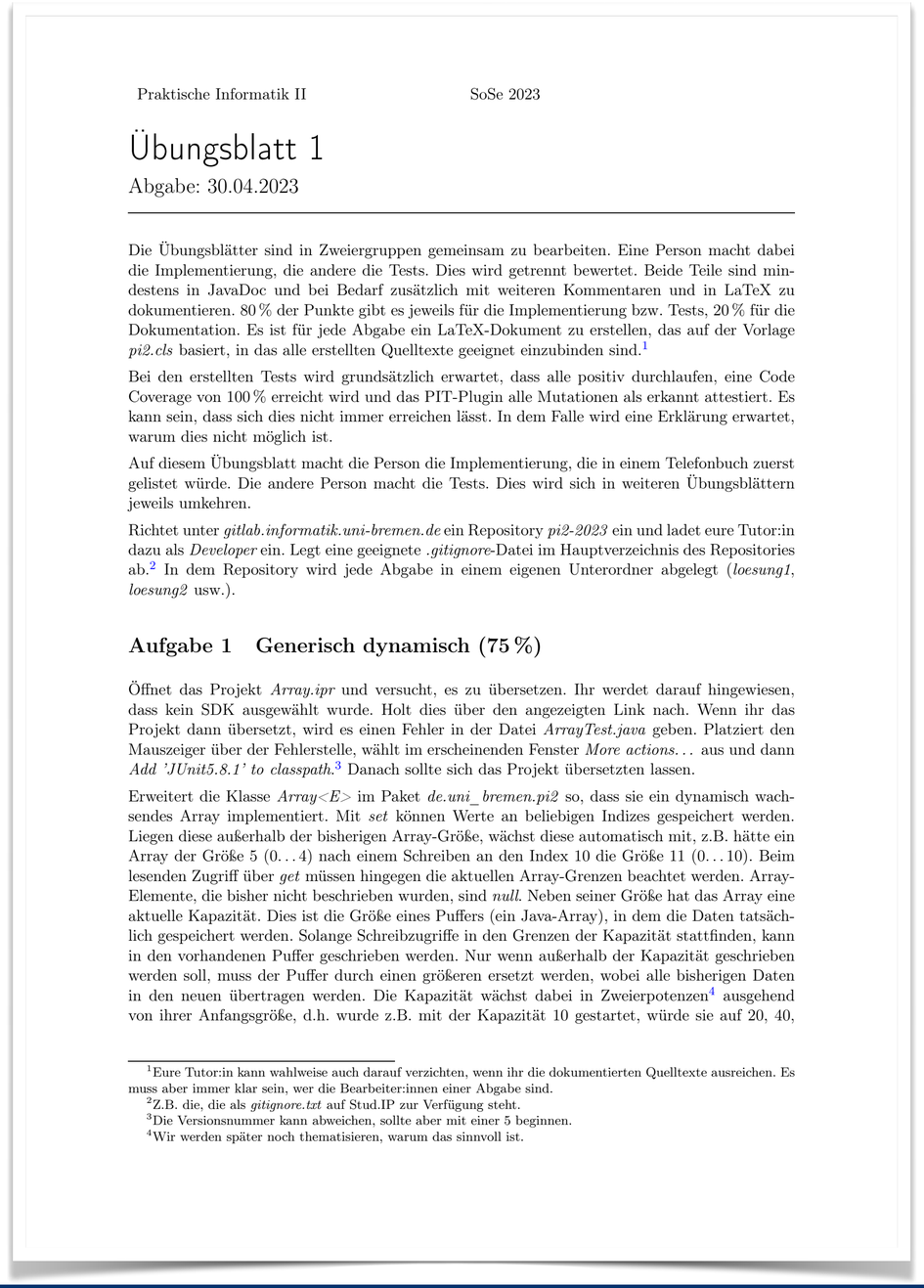
Zusammenfassung der Konzepte

- **Generische Klasse** und **Generischer Typ**
- **Typebound**
- **Invarianz**, **Bivarianz**, **Covarianz** und **Contavarianz**
- **Type-Erasure** und **Rawtype**
- **Generische Methoden**

Universität Bremen

Übungsblatt 1

- Aufgabe 1
  - Implementierung eines dynamisch wachsenden Arrays
  - Array hat aktuelle Größe und einen unterliegenden Puffer, der in Zweierpotenzen bei Bedarf wächst
- Aufgabe 2
  - Implementierung der Schnittstelle **Iterable**
- Eine Person implementiert, die andere testet



Praktische Informatik II SoSe 2023

Übungsblatt 1

Abgabe: 30.04.2023

Die Übungsblätter sind in Zweiergruppen gemeinsam zu bearbeiten. Eine Person macht dabei die Implementierung, die andere die Tests. Dies wird getrennt bewertet. Beide Teile sind mindestens in JavaDoc und bei Bedarf zusätzlich mit weiteren Kommentaren und in LaTeX zu dokumentieren. 80 % der Punkte gibt es jeweils für die Implementierung bzw. Tests, 20 % für die Dokumentation. Es ist für jede Abgabe ein LaTeX-Dokument zu erstellen, das auf der Vorlage *pi2.cls* basiert, in das alle erstellten Quelltexte geeignet einzubinden sind.[1]

Bei den erstellten Tests wird grundsätzlich erwartet, dass alle positiv durchlaufen, eine Code Coverage von 100 % erreicht wird und das PIT-Plugin alle Mutationen als erkannt attestiert. Es kann sein, dass sich dies nicht immer erreichen lässt. In dem Falle wird eine Erklärung erwartet, warum dies nicht möglich ist.

Auf diesem Übungsblatt macht die Person die Implementierung, die in einem Telefonbuch zuerst gelistet würde. Die andere Person macht die Tests. Dies wird sich in weiteren Übungsblättern jeweils umkehren.

Richtet unter *gitlab.informatik.uni-bremen.de* ein Repository *pi2-2023* ein und ladet eure Tutor:in dazu als *Developer* ein. Legt eine geeignete *.gitignore*-Datei im Hauptverzeichnis des Repositories ab.[2] In dem Repository wird jede Abgabe in einem eigenen Unterordner abgelegt (*loesung1*, *loesung2* usw.).

Aufgabe 1 Generisch dynamisch (75 %)

Öffnet das Projekt *Array.ipr* und versucht, es zu übersetzen. Ihr werdet darauf hingewiesen, dass kein SDK ausgewählt wurde. Holt dies über den angezeigten Link nach. Wenn ihr das Projekt dann übersetzt, wird es einen Fehler in der Datei *ArrayTest.java* geben. Platziert den Mauszeiger über der Fehlerstelle, wählt im erscheinenden Fenster *More actions...* aus und dann *Add 'JUnit5.8.1' to classpath*.[3] Danach sollte sich das Projekt übersetzten lassen.

Erweitert die Klasse *Array<E>* im Paket *de.uni_bremen.pi2* so, dass sie ein dynamisch wachsendes Array implementiert. Mit *set* können Werte an beliebigen Indizes gespeichert werden. Liegen diese außerhalb der bisherigen Array-Größe, wächst diese automatisch mit, z.B. hätte ein Array der Größe 5 (0...4) nach einem Schreiben an den Index 10 die Größe 11 (0...10). Beim lesenden Zugriff über *get* müssen hingegen die aktuellen Array-Grenzen beachtet werden. Array-Elemente, die bisher nicht beschrieben wurden, sind *null*. Neben seiner Größe hat das Array eine aktuelle Kapazität. Dies ist die Größe eines Puffers (ein Java-Array), in dem die Daten tatsächlich gespeichert werden. Solange Schreibzugriffe in den Grenzen der Kapazität stattfinden, kann in den vorhandenen Puffer geschrieben werden. Nur wenn außerhalb der Kapazität geschrieben werden soll, muss der Puffer durch einen größeren ersetzt werden, wobei alle bisherigen Daten in den neuen übertragen werden. Die Kapazität wächst dabei in Zweierpotenzen[4] ausgehend von ihrer Anfangsgröße, d.h. wurde z.B. mit der Kapazität 10 gestartet, würde sie auf 20, 40,

[1] Eure Tutor:in kann wahlweise auch darauf verzichten, wenn ihr die dokumentierten Quelltexte ausreichen. Es muss aber immer klar sein, wer die Bearbeiter:innen einer Abgabe sind.
[2] Z.B. die, die als *gitignore.txt* auf Stud.IP zur Verfügung steht.
[3] Die Versionsnummer kann abweichen, sollte aber mit einer 5 beginnen.
[4] Wir werden später noch thematisieren, warum das sinnvoll ist.