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.

80 usw. wachsen, sobald nötig. Eine Kapazität von 0 ist erlaubt, wobei die nächste Stufe der Vergrößerung dann mindestens 1 ist.

Implementiert die folgenden Methoden:

Array(int): Der Konstruktor bekommt die Anfangskapazität übergeben und legt daraufhin einen Puffer für so viele Elemente vom Typ E an. Die *Größe* des Arrays ist anfangs 0. Eine negative Kapazität führt zu einer *IllegalArgumentException*.

int size(): Liefert die Größe des Arrays.

int capacity(): Liefert die aktuelle Kapazität des Puffers, in dem die Daten gespeichert sind. Ist immer mindestens so groß wie *size()*. Diese Methode gibt es eigentlich nur, um das Verhalten des Wachsens des Puffers in Tests überprüfen zu können.

void set(int, E): Schreibt einen Wert (zweiter Parameter) an eine durch einen Index (erster Parameter) definierte Stelle in das Array. Liegt der Index außerhalb der bisherigen Größe des Arrays, wird diese so erhöht, dass er gerade noch hineinpasst.[5] Ist der Index negativ oder ein unmöglicher positiver Wert (welcher könnte das sein?), wird eine *ArrayIndexOutOfBoundsException* erzeugt.

E get(int): Liefert das durch einen Index bezeichnete Element des Arrays zurück. Liegt der Index außerhalb der Grenzen des Arrays ($0 \ldots size() - 1$), wird eine *ArrayIndexOutOfBoundsException* erzeugt.

Für die Tests ist die Klasse *ArrayTest* im Projekt vorgesehen.

Aufgabe 2 Generisch iterativ (25 %)

Lasst die Klasse *Array<E>* die Schnittstelle *Iterable<E>* implementieren. Der dafür erzeugte Iterator muss lediglich die Methoden *hasNext* und *next* implementieren, so wie in der zugehörigen Dokumentation beschrieben.

[5] Beachtet das oben beschriebene Verhalten für die Vergrößerung der Kapazität.