Hinweise zur Klausur

Sehr geehrte Studentinnen und Studenten,

auf den folgenden Seiten finden Sie die Prog2-Klausur aus dem SS2022. Zu jeder Aufgabe wurde eine Beispiellösung hinzugefügt, sowie Hinweise zur Bewertung.

- Die Beispiellösungen sind oft nicht die einzige mögliche Lösung. Auch andere Lösungen wurden akzeptiert, wenn Sie die Aufgabenstellung erfüllten.
- Die Bewertungshinweise sind nicht vollständig. Eine Vielzahl von eingereichten Varianten der Lösungen wurde individuell bewertet.
- Grundsätzlich wurden typische Flüchtigkeitsfehler, die von einer IDE erkannt worden wären, nicht negativ bewertet. Dies gilt z.B. für fehlende Semikolon oder ähnliches.
- Fehlende `import`-Anweisungen wurde nicht als Fehler gewertet.

Die Klausur hatte zwei leicht unterschiedliche Versionen. Einige typische Fehler, die uns auffielen, habe ich hier zusammengestellt:

Die **Hinweise** zur Klausur wurden von vielen Studenten nicht beachtet. Dort stand insbesondere, dass die *"ausführliche und sinnvolle Kommentierung des Codes"* bewertet wird. Tatsächlich gab es bei den einzelnen Aufgaben hierfür insgesamt 12 Punkte. Das entspricht bei der Bewertung einer vollen Notenstufe, und hätte bei vielen "nb" zu einer bestandenen Klausur geführt. Auch in der Lehrveranstaltung wurde hierauf mehrfach hingeweisen, im Forum wurde es mehrfach angesprochen. Mir ist absolut unverständlich, warum diese Punkte liegengelassen wurden.

Die Hinweise enthielten auch die Information, dass am Ende der Klausur die benötigten Informationen zu den verwendeten Java-Klassen bereitgestellt wurden. Auch dies haben einige von Ihnen nicht beachtet. Schon bei der ersten Aufgabe hat sich das ggf. ausgewirkt.

Aufgabe 1 war einfach lösbar, wenn erkannt wurde, dass `Collection` die Schnittstelle `Iterable` implementiert (was aus der beigefügten Dokumentation ersichtlich war).

In **Aufgabe 3** war gefordert, eine Klasse `Kellerspeicher` oder `Schlange` aus der Java-Klasse `Vector` abzuleiten. Die Klasse `Vector` war mit allen notwendigen Methoden beschrieben. Statt die Eigenschaften von `Vector` einfach zu nutzen, wurde oft innerhalb der Klasse ein zusätzliches `Vector`-Objekt angelegt (was zu einem Punkteabzug führte). Einige Studenten haben sogar ein ganz anderes Objekt, oft ein Feld (Array), angelegt, um Daten zu speichern. In diesem Fall wurde die gesamte Aufgabe mit null Punkten bewertet. Offensichtlich haben einige von Ihnen die Grundlagen der objektorientierten Programmierung noch nicht verstanden.

In **Aufgabe 4** war der häufigste Fehler, daß das Werfen der `Exception` nicht korrekt mit einer Lambdafunktion geprüft wurde.

Bei **Aufgabe 5** war die fehlende Anwendung des LESS-Prinzip ein typischer Fehler.

Aufgabe 7 war von der Logik wesentlich anspruchsvoller als die anderen Aufgaben. Es war die einzige Aufgabe, bei der vor der Lösung wirklich nachgedacht werden musste. Offensichtlich

habe hier viele von Ihnen einfach losprogrammiert, ohne sich vorher einen Lösungsansatz zu überlegen. Das Durchlaufen eines zweidimensionalen Feldes ist eigentlich nicht schwierig.

Bei **Aufgabe 9** wurde in vielen Lösungen die geforderte Typsicherheit nicht beachtet. Stattdessen wurde einfach mit der Oberklasse `Person` gearbeitet, was zu einem falschen Ergebnis führte. In einigen Lösungen wurde nur ein Typparameter verwendet.

Bitte stellen Sie individuelle Rückfragen zu Ihrer Klausur im dafür vorgesehenen Termin! Einzelne EMails dieser Art werde ich nicht beantworten.

Nachfolgend beginnen die Hinweise, die Sie auch in der Klausur bekommen hatten.

- Bei Programmieraufgaben geht es **nicht nur** darum, eine Aufgabe korrekt zu lösen! Bewertet werden auch:
  - Sichtbarkeit von Variablen und Methoden; Kapselung
  - Qualität der Programmierung (z.B. kein überflüssiger Speicherverbrauch,...)
  - ausführliche und sinnvolle Kommentierung des Codes (besser zuviel als zuwenig)
- **Verwenden Sie nicht** die Java-Klassen `AbstractCollection`, `AbstractMap` oder `Collections`, und auch nicht deren Unterklassen wie z.B. `ArrayList, ArrayDequeue, LinkedList, HashMap, Set`, **es sei denn, dies ist ausdrücklich angegeben.**
- **Achten Sie auf Zugriffsrechte von Variablen und Methoden**. Klasseninterne Variablen sollten nach außen möglichst nicht sichtbar sein. D.h. Wenn Sie z.B. für einen Unittest den Zugriff auf eine interne Klassenvariable `var` benötigen, dann implementieren Sie in der Klasse eine lesende getter-Methode `getVar()`, die Ihnen den aktuellen Wert der Variable liefert.
- **Nur lesbare und eindeutige Lösungen werden bewertet!**
- Am Ende der Klausur finden Sie Ausschnitte aus der Java Dokumentation!

Falls eine Aufgabenstellung unklar sein sollte, ergänzen Sie die Aufgabenstellung sinnvoll. Notieren Sie Ihre Ergänzung bei Ihrer Lösung.

Ergebnis (bitte nichts eintragen!):

Frage:	1	2	3	4	5	6	7	8	9	10	Summe:
Punkte:	10	12	10	8	6	6	15	6	6	11	90
Erreicht:											

Falls die angegebene Punktzahl auf diesem Deckblatt von der Punktzahl bei der Aufgabenstellung abweichen sollte, gilt die Angabe bei der Aufgabe. Die maximal erreichbare Gesamtpunktzahl wird dann entsprechend angepasst.

1. (10 Punkte) Generische Methode

   Schreiben Sie in einer nicht-generischen Klasse `Util` eine statische generische Methode `int numElementesInCollection(Collection<T>)`, welche die Anzahl der Elemente in einer als Parameter übergebenen Kollektion (Java `Collection`) als Ergebnis zurückliefert.

   Verwenden Sie dabei nicht die Methode `size` der Kollektion!

   Beispiel: Die Aufrufe

```
public static void main(String[] args) {
  int[] ints = {8,3,9,2,5,1,8};
  Collection<Integer> col = new ArrayList<>();
  for(int i:ints)
    col.add(i);
  System.out.println("Aufgabe␣1.1:␣Col␣enthält␣"+Util.
     countElementsInCollection(col)+"␣Elemente");
}
```

   erzeugen eine Ausgabe wie

```
   Col enthält 7 Elemente
```

   Lösung:

```
import java.util.Collection;
public class Util {
  /*
   * Anzahl der Elemente in einer Kollektion
   * @return: int
   */
  public static <T> int countElementsInCollection(Collection<
     T> col) {
    int counter = 0;    // Zähler für die Anzahl der Elemente
    for(@SuppressWarnings("unused") T e:col)
      counter++;
    return counter;
  }
}
```

   Bewertungshinweise:

   - korrekter Klassenkopf: 2 P
   - korrekter Methodenkopf: 2 P
   - korrekte Verwendung von Generics (insbes. im Parameter): 3 P
   - korrektes Ergebnis: 2 P
   - ausreichende Kommentierung der Methode: 1 P
   - Punktabzug bei umständlicher Programmierung oder unnötiger Sichtbarkeit von Variablen und Methoden.
   - kein Punktabzug für fehlende `import`-Anweisungen.

2. (12 Punkte) Unittest

Schreiben Sie in einer Klasse `UtilTest` einen Unittest, welcher die Methode `int numElementesInCollection(Collection<T>)` in einer Testmethode `countElementsInCollectionTest` für leere und gefüllte Kollektionen prüft. Dabei dürfen Sie die Java-Klasse `ArrayList` verwenden.

Lösung:

```
import static org.junit.jupiter.api.Assertions.*;
import java.util.ArrayList;
import java.util.Collection;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class UtilTest {
  int[] ints10 = {8,3,9,2,5,1,8,5,9,2};   // int-Feld für
     Tests
  Collection<Integer> col0, col10;      // Kollektionen für
     Tests

  @BeforeEach
  void init() {
    col0 = new ArrayList<>();   // leere Kollektion
    col10 = new ArrayList<>();    // befüllte Kollektion
    for(int i:ints10)
      col10.add(i);
  }

  @Test
  /*
   * Korrekte Bestimmung der Anzahl von Elementen bei leerer
      und gefüllter Kollektion
   */
  void countElementsInCollectionTest() {
    assertEquals(0, Util.countElementsInCollection(col0));
    assertEquals(ints10.length, Util.
       countElementsInCollection(col10));
  }
}
```

Bewertungshinweise:

- korrekter Unittest, inkl. `@Test`, für leere und gefüllte Liste, jeweils: 3 P (max 6 P)
- korrekte Verwendung von `assert` jeweils: 2 P (max 4 P)
- ausreichende Kommentierung der Methode: 2 P
- Punktabzug bei umständlicher Programmierung oder unnötiger Sichtbarkeit von Variablen und Methoden.
- kein Punktabzug für fehlende `import`-Anweisungen.

3. (10 Punkte) Kellerspeicher

Erstellen Sie eine Klasse `Kellerspeicher` (LIFO), die Sie direkt von der Javaklasse `Vector` ableiten. Folgende Methoden sollen von der Klasse (mindestens) bereitgestellt werden:

- `boolean isEmpty()` : Gibt `TRUE` zurück, wenn der Kellerspeicher leer ist, sonst `FALSE`.
- `E peek()` : Gibt das oberste Element auf dem Speicher zurück, ohne es vom Speicher zu löschen. Wirft eine `ArrayIndexOutOfBoundsException`, falls der Speicher leer ist.
- `E pop()` : Gibt das oberste Element auf dem Speicher zurück und löscht es aus dem Speicher. Wirft eine `ArrayIndexOutOfBoundsException`, falls der Speicher leer ist.
- `E push(E item)` : Schiebt ein neues Element auf den Speicher.

Beispiel: Die Aufrufe

```
Kellerspeicher<Integer> ks = new Kellerspeicher<>();
System.out.println("Kellerspeicher.isEmpty()␣ist␣"+ks.
   isEmpty());
  ks.push(5);
  System.out.print("pushed␣"+ks.peek());
  System.out.println(",␣removed␣"+ks.pop());
}
```

erzeugen eine Ausgabe wie

```
Kellerspeicher.isEmpty() ist true
pushed 5, removed 5
```

Lösung:

```
public class Kellerspeicher<T> extends Vector<T>{
  public void push(T elem) {
    add(0,elem);  // Hinzufügen an Position 0
  }
  // gib das oberste Element aus:
  public T peek() {
    return get(0);
  }
  // gib das oberste Element aus
  // und entferne es aus dem Speicher:
  public T pop() {
    return remove(0);
  }
}
```

Bewertungshinweise:

- korrekter Methodenkopf: ohne internes Datenobjekt: 2 P; mit Vector: 0 P; mit Feld ö.ä.: gesamte Aufgabe 0 Punkte!
- push, peek, pop korrekt: jeweils 2 P (max. 6)
- ausreichende Kommentierung der Methoden: 2 P
- Punktabzug bei umständlicher Programmierung oder unnötiger Sichtbarkeit von Variablen und Methoden.
- kein Punktabzug für fehlende `import`-Anweisungen.

4. (8 Punkte) Unittest zum Kellerspeicher

   Schreiben Sie für die Klasse `Kellerspeicher` einen Unittest `KellerspeicherTest`, welcher die beiden folgenden Tests durchführt:

   - `void pushPeekTest()`: Befindet sich ein gerade eingefügtes Element an oberster Position im Speicher?
   - `popEmptyExceptionTest()`: Wird beim Entfernen eines Elements aus einem leeren Speicher die richtige Exception geworfen?

Lösung:

```
class KellerspeicherTest {
  Kellerspeicher<Integer> ks;
  int[] ints10 = {4,7,9,2,5,6,3,1,8,5};
  @BeforeEach
  void init() {
    ks = new Kellerspeicher<>();
  }
  @Test
  /* befindet sich ein gerade hinzugefügtes Element
   * an der obersten Position des Speichers?
   */
  void pushPeekTest() {
    for(int i:ints10) {
      ks.push(i);
      assertEquals(i, ks.peek());
    }
  }
  @Test
  /* wird beim Entfernen eines Elements aus einem leeren
     Speicher
   * die richtige Exception geworfen?
   */
  void popEmptyExceptionTest() {
    assertThrows(ArrayIndexOutOfBoundsException.class, () ->
       ks.pop(), "oops");
  }
}
```

Bewertungshinweise:

- korrekter Unittest: 2 P
- Korrekte Prüfung des Hinzufügens mit `assert`: 2 P
- Korrekte Prüfung der Exception: 3 P
- ausreichende Kommentierung des Unittests: 1 P
- Punktabzug bei umständlicher Programmierung oder unnötiger Sichtbarkeit von Variablen und Methoden.
- kein Punktabzug für fehlende `import`-Anweisungen.

5. (6 Punkte) Wildcards

   Schreiben Sie in einer nicht-generischen Klasse `Util` eine statische generische Methode `ksCopy`, welche als Parameter zwei `Kellerspeicher` erhält und den Inhalt des ersten in den zweiten kopiert. Der erste Kellerspeicher soll dabei nicht verändert werden, die Reihenfolge der kopierten Elemente im zweiten Kellerspeicher ist egal.

Lösung:

```
/* Kopiere Kellerspeicher 'from' nach 'to' */
public static <T> void ksCopy(Kellerspeicher<? extends T>
   from, Kellerspeicher <? super T> to) {
  for(T e:from)
    to.push(e);
}
```

Bewertungshinweise:

- Korrekter Methodenkopf: 5 P
- Korrekter Methodenrumpf: 1 P
- ausreichende Kommentierung: 1 P

6. (6 Punkte) Schreiben Sie zu dieser Methode eine Testmethode, welche prüft, ob der Zielspeicher alle kopierten Elemente enthält.

Lösung:

```
@Test
/* Kopieren von Elementen von/in einen Kellerspeicher */
void copyKellerspeicherTest() {
  // Quelle, Ziel, Elemente:
  Kellerspeicher<Integer> ksFrom = new Kellerspeicher<>();
  Kellerspeicher<Number> ksTo = new Kellerspeicher<>();
  int[] ints5 = {1,3,5,7,9};

  for(int i:ints5)          // Quelle füllen
    ksFrom.push(i);
  Util.ksCopy(ksFrom, ksTo);      // Elemente kopieren

  for(int i:ints5)
    assertEquals(i, ksTo.pop());  // Elemente im Ziel prü
       fen
}
```

Bewertungshinweise:

- Korrekter Test: 5 P
- ausreichende Kommentierung: 1 P

7. (15 Punkte) Iterator

Die Klasse `A2D` implementiert ein generisches zweidimensionales Feld:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class A2D<T> implements Iterable<T> {
  int zeilen, spalten;  // Anzahl der Zeilen und Spalten
  T[][] feld;       // das Feld
  // Konstruktor
  @SuppressWarnings("unchecked")
  public A2D(int z, int s) {
    zeilen = z;
    spalten = s;
    feld = (T[][]) new Object[zeilen][spalten];
  }
  // element hinzufügen an der Position z,s
  public void insert(T elem, int z, int s) {
    feld[z][s] = elem;
  }
  public T get(int z, int s) {
    return feld[z][s];
  }
  @Override
  public Iterator<T> iterator() {
    // TODO Auto-generated method stub
    return null;
  }
}
```

Implementieren Sie die fehlende Methode `iterator()`. Fügen Sie den fehlenden Iterator hinzu, der nach folgender Regel über das Feld iterieren soll:

- Zunächst werden alle Elemente der *geraden* Zeilen der Reihe nach ausgegeben;
- anschließend die Elemente der *ungeraden* Zeilen.

Beispiel: Die Aufrufe

```
    A2D<Integer> a2D = new A2D<>(3,4);
    int[][] ints2d = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    for(int z=0; z<ints2d.length; z++)
      for(int s=0; s<ints2d[0].length; s++)
        a2D.insert(ints2d[z][s], z, s);
    System.out.println("a2D␣=␣"+a2D);
    System.out.print("Iterator␣liefert:␣");
    for(int i:a2D)
      System.out.print(i+"␣");
    System.out.println();
```

erzeugen eine Ausgabe wie

```
  a2D = [[1 2 3 4 ][5 6 7 8 ][9 10 11 12 ]]
  Iterator liefert: 1 2 3 4 9 10 11 12 5 6 7 8
```

Lösung:

```
@Override
public Iterator<T> iterator() {
  return new a2DiteratorGeradeUngeradeZeilen();
}
private class a2DiteratorGeradeUngeradeZeilen implements
   Iterator<T>{
  int z=0, s=0; // nächstes Element
  @Override
  public boolean hasNext() {
    return z<zeilen && s<spalten;
  }
  @Override
  public T next() {
    if(!hasNext())  // gibt es noch ein Element?
      throw new NoSuchElementException();
    T result = feld[z][s];
    if(s<spalten-1) // Zeile noch nicht zuende?
      s++;
    else {       // Zeile zuende
      s = 0;     // fange wieder vorne an
      z = z+2;  // in der übernächsten Zeile
      if(z>=zeilen && z\%2 == 0)   // Ende der geraden
         Zeilen
        z = 1;
    }
    return result;
  }
}
```

Bewertungshinweise:

- korrekte `iterator()`-Methode: 1 P
- private Iteratorklasse korrekt angelegt: 3 P inkl. Variablen
- `hasNext()` korrekt: 2 P
- `next()` korrekt: 6 P (Rückgabe: 1; Spaltenindex: 1; Spalten- / Zeilenende: je 2)
- `next()` wirft korrekte Exception: 1 P
- ausreichende Kommentierung der Methoden: 2 P
- Punktabzug bei umständlicher Programmierung oder unnötiger Sichtbarkeit von Variablen und Methoden.
- kein Punktabzug für fehlende `import`-Anweisungen.

8. (6 Punkte) Vergleichbarkeit

Objekte einer Klasse

```
public class Flurstueck {
  private int nummer, groesse;
  public Flurstueck(int nummer, String groesse) {
    this.groesse = groesse;
    this.nummer = nummer;
  }
  public int getGroesse() {return groesse;}
  public int getNummer() {return nummer;}
}
```

sollen in der Methode
`int compare(Flurstueck f1, Flurstueck f2, Comparator<Flurstueck> comp)` über die Größe verglichen werden.
Implementieren Sie dazu einen geeigneten Komparator `ComparatorFlurstueckGroesse`.

Lösung:

```
import java.util.Comparator;
// Comparator zum Vergleich von Fluerstuecken über ihre
   Groesse
public class ComparatorFlurstueckGroesse implements
   Comparator<Flurstueck>{
  @Override
  public int compare(Flurstueck f1, Flurstueck f2) {
    return f1.getGroesse().compareTo(f2.getGroesse());
  }
}
```

Bewertungshinweise:

- korrekte Signatur: 3 P
- korrekte `compare`-Methode: 2 P
- ausreichende Kommentierung der variablen und Methoden: 1 P

9. (6 Punkte) Einschränkung

Die Klassen `Student` und `Boxer` sind beide von `Person` abgeleitet:

```
public class Student extends Person{
  int matrikelnummer;   // die Matrikelnummer d. Studenten
  public Student(String vorname, String nachname, int
     matrikelnummer) {
    super(vorname,nachname);
    this.matrikelnummer = matrikelnummer;
  }
}

public class Boxer extends Person{
  int gewicht;    // Gewicht in kg
  public Boxer(String vorname, String nachname, int gewicht) {
    super(vorname,nachname);
    this.gewicht = gewicht;
  }
}
```

Schreiben Sie eine Klasse `Paar`, bei der Sie im Konstruktor zwei Objekte dieser drei Klassen übergeben und diese in den Methoden `getErstes()` (für das erste Objekt) bzw. `getZweites()` **typsicher** abfragen können.

Ein Aufruf wie

```
    Paar<Student, Boxer> p = new Paar<>(new Student("Stefan", "
       Schlimm", 123456), new Boxer("Bertram", "Bullie", 99));
    Student s = p.getErstes();
    Boxer b = p.getZweites();
    System.out.println("Student␣=␣"+s+",␣Boxer␣=␣"+b);
```

resultiert in einer Ausgabe wie

```
  Student = Stefan Schlimm, Boxer = Bertram Bullie
```

Lösung:

```
public class Paar<T extends Person,U extends Person> {
  T erstes; // erstes Element
  U zweites;  // zweites Element
  // Konstruktor
  public Paar(T erstes, U zweites) {
    this.erstes = erstes;
    this.zweites = zweites;
  }
  public T getErstes() {return erstes;}
  public U getZweites() {return zweites;}
}
```

Bewertungshinweise:

- korrekte Signatur mit zwei Klassenvariablen: 2 P
- korrekte Methoden: jeweils 1 P, insges. max. 3 P
- ausreichende Kommentierung der variablen und Methoden: 1 P

10. (11 Punkte) Typableitung

Die folgenden Klassen und Methoden sind wie folgt definiert:

```
public class A {}
public class B extends A{}
public class C extends A{}

static private<T> T ret1(T e, T z) {return e;}
static private<T,U> T ret2(T e, U z) {return e;}
```

Bestimmen Sie den Typ von `T` und `U` jeweils

- einzeln für jeden Parameter (optional, hierfür gibt es keine Punkte!)
- gemeinsam anhand aller Parameter (`T` (auch optional!)
- als Rückgabetyp der Methode (ggf. *(*vor) dem Cast!) (dies wird bewertet)

Stellen Sie außerdem fest, ob die Zuweisung übersetzt (compiliert) und ob der Aufruf einen Laufzeitfehler erzeugt. Aufrufe:

```
A a = ret1(new B(), new C());      // (1)
a = (A) ret1(new B(), new C());    // (2)
B b = ret1(new B(), new C());      // (3)
B b = (B) ret1(new B(), new C()); // (4)
C c = ret1(new B(), new C());      // (5)
C c = (C) ret1(new B(), new C()); // (6)

a = ret2(new B(), new C());        // (7)
a = (A) ret2(new B(), new C());    // (8)
b = ret2(new B(), new C());        // (9)
b = (B) ret2(new B(), new C());    // (10)
C c = ret2(new B(), new C());      // (11)
c = (C) ret2(new B(), new C());    // (12)
```

Füllen Sie die Tabelle entsprechend aus:

Aufruf	T erstes	T zweites	T Aufruf	T Rückgabe	compiliert?	LZF?
Beispiel	A	B	A	C	ja	nein
(1)						
(2)						
(3)						
(4)						
(5)						
(6)						
(7)						
(8)						
(9)						
(10)						
(11)						
(12)						

Lösung:

Aufruf	T erstes	T zweites	T Aufruf	T Rückgabe	compiliert?	LZF?
Beispiel	A	B	A	C	ja	nein
(1)	B	C	A	A	ja	nein
(2)	B	C	A	A	ja	nein
(3)	B	C	A	A	nein	nein
(4)	B	C	A	A (B)	ja	nein
(5)	B	C	A	A	nein	nein
(6)	B	C	A	A (C)	ja	ja
(7)	B	C	B	B	ja	nein
(8)	B	C	B	B (A)	ja	nein
(9)	B	C	B	B	ja	nein
(10)	B	C	B	B	ja	nein
(11)	B	C	B	B	nein	nein
(12)	B	C	B	B (C)	nein	nein

Bewertungshinweise:

- jeder korrekte Rückgabetyp: 0.5 P; maximal 6 P
- geklammerte (gecastete) Typen werden *nicht* als korrekt bewertet
- jeder korrekt erkannte Fehler beim Compilieren oder Laufzeitfehler: 1 P, maximal 5 P

Auszüge aus der Java Dokumentation

AbstractCollection

```
public abstract class AbstractCollection<E>
extends Object
implements Collection<E>
```

This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.

AbstractList

```
public abstract class AbstractList<E>
extends AbstractCollection<E>
implements List<E>
```

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access"data store (such as an array). For sequential access data (such as a linked list), AbstractSequentialList should be used in preference to this class.

ArrayList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

Methoden

`public boolean add(E e)`

Appends the specified element to the end of this list.

- Parameters: e - element to be appended to this list
- Returns: true (as specified by Collection.add(E))

`public void add(int index, E element)`

Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

- Parameters:
  - index - index at which the specified element is to be inserted
  - element - element to be inserted
- Throws: IndexOutOfBoundsException - if the index is out of range (index $<$ 0 || index $>$ size())

`public E get(int index)`

Returns the element at the specified position in this list.

- Parameters: index - index of the element to return
- Returns: the element at the specified position in this list
- Throws: IndexOutOfBoundsException - if the index is out of range (index $<$ 0 || index $>=$ size())

`public boolean isEmpty()`

Tests if this list has no components. Returns true if and only if this list has no components, that is, its size is zero; false otherwise.

`public E remove(int index)`

Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices).

- Parameters: index - the index of the element to be removed
- Returns: the element that was removed from the list
- Throws: IndexOutOfBoundsException - if the index is out of range (index $<$ 0 || index $>=$ size())

Collection

```
public interface Collection<E>
extends Iterable<E>
```

The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Iterable

```
public interface Iterable<T>
```

Methoden

`Iterator<T> iterator()`

Returns an iterator over elements of type T.

List

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Vector

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

Methoden

`public boolean add(E e)`

Appends the specified element to the end of this vector.

- Parameters: e - element to be appended to this list
- Returns: true (as specified by Collection.add(E))

`public void add(int index, E element)`

Inserts the specified element at the specified position in this vector. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

- Parameters:
  - index - index at which the specified element is to be inserted
  - element - element to be inserted
- Throws: ArrayIndexOutOfBoundsException - if the index is out of range (index $<$ 0 || index $>$ size())

`public E get(int index)`

Returns the element at the specified position in this vector.

- Parameters: index - index of the element to return
- Returns: the element at the specified position in this vector
- Throws: ArrayIndexOutOfBoundsException - if the index is out of range (index $<$ 0 || index $>=$ size())

`public boolean isEmpty()`

Tests if this vector has no components. Returns true if and only if this vector has no components, that is, its size is zero; false otherwise.

`public E remove(int index)`

Removes the element at the specified position in this vector. Shifts any subsequent elements to the left (subtracts one from their indices).

- Parameters: index - the index of the element to be removed
- Returns: the element that was removed from the list
- Throws: ArrayIndexOutOfBoundsException - if the index is out of range (index $<$ 0 || index $>=$ size())