# Funktionale und objektorientierte Programmierkonzepte

Florian Kadner und Lukas Röhrig
(Gesamtleitung: Prof. Karsten Weihe)

Wintersemester 18/19
v2.3

# Übungsblatt 8

| | |
|---|---|
| Themen: | Generics und Collections |
| Relevante Folien: | Generics und Collections |
| Abgabe der Hausübung: | 21.12.2018 bis 23:55 Uhr |

## V Vorbereitende Übungen

### V1 Theoriefragen

☆ ☆ ☆

1. Welche Vorteile ergeben sich durch die Nutzung von Generics?
2. Diskutieren Sie Vor- und Nachteile von Collections gegenüber herkömmlichen Arrays unter den folgenden Aspekten:
   (a) Finden von Elementen
   (b) Einfügen neuer Elemente
   (c) Löschen von Elementen
3. Auch in Racket haben Sie Listen kennengelernt. Ist das Java-Interface `java.util.list` vergleichbar mit den Listen aus Racket?

### V2 Collections und Exceptions

★ ☆ ☆

Gegeben sei folgender Codeausschnitt:

```
1 double foo(double[] numbers, double n) {
2   LinkedList<Double> list = new LinkedList<Double>();
3   for (double x : numbers) {
4     if (x > 0 && x <= n && x % 2 != 0) {
5       list.add(x);  }
6   }
7   Collections.sort(list);
8   return list.getLast();  }
```

(1) Beschreiben Sie kurz und bündig, aber präzise und unmissverständlich was der oben gegebene Code macht.

(2.1) An welcher Stelle kann im Code eine Exception geworfen werden? Durch welche Eingaben wird sie ausgelöst?

(2.2) Modifizieren Sie den Code mithilfe eines try/catch-Blockes so, dass in diesen Fällen die Nachricht der Exception auf der Konsole ausgegeben wird.

## V3 A-well-a bird bird bird, bird is the word Part I ★ ☆ ☆

In dieser Aufgabe betrachten wir eine stark reduzierte Typhierarchie zur Modellierung von Vögeln. Dabei stellen die Pfeile die Erbbeziehungen zwischen Klassen dar. Dazu ist das folgende Typdiagramm in Abbildung 1 gegeben. Hier ist `Bird` also die Oberklasse und die drei anderen Klassen sind Erben dieser.

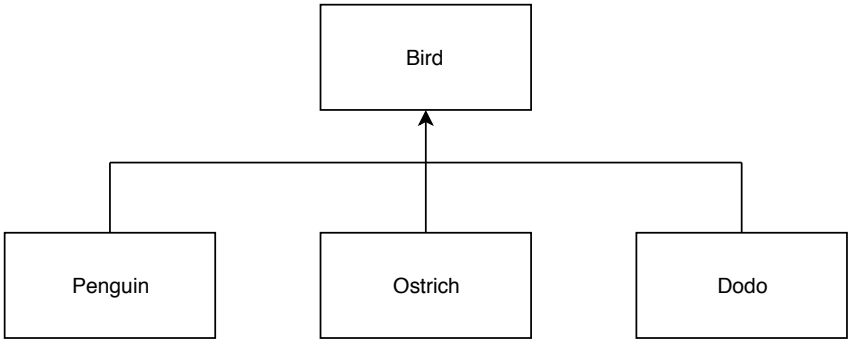



Abbildung 1: Typhierarchie mit drei Vogelarten

Wir nutzen die generische Datenstruktur `Vector<E>`. Dabei beschränken wir uns auf die Methode `void add(E entry)`, die ein Element vom Typ `E` in den Vector einfügt.

(1): Deklarieren und initialisieren Sie eine Variable v mit dem **statischen** Basistyp `List` und dem **dynamischen** Typ `Vector`, so dass darin genau Objekte der Typen `Birds`, `Penguin`, `Ostrich` und `Dodo` gespeichert werden können. Nutzen Sie generische Typparameter!

(2): Geben Sie Java-Code an, um in den obigen Vector `v` jeweils ein neues Element vom Typ `Penguin` und `Ostrich` einzufügen. Sie dürfen zur Vereinfachung die Parameter der Konstruktoren der Klassen `Penguin` und `Ostrich` durch ..... abkürzen.

(3): Die Methode `addAll(Birds)` existiert im Interface `List` und fügt eine gesamte Collection in eine gegebene Collection ein. Die Klasse `ArrayList` implementiert das Interface `List`. Ist die folgende Anweisung – **nach** dem Code der vorherigen Aufgaben – dann zulässig?

```
new ArrayList<Dodo>().addAll(v);
```

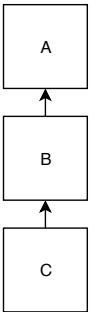
## V4 Elemente tauschen ★ ☆ ☆

Schreiben Sie eine Methode

```
void switchElements(T[] a, int i, int j)throws IllegalArgumentException
```

Die Methode vertauscht die Elemente im übergebenen Array `a` an den zwei angegebenen Indizes `i` und `j`. Falls für `a` eine `null`-Referenz übergeben wird oder einer der Indizes nicht in dem Array liegt, soll eine `IllegalArgumentException` geworfen werden.

## V5 Typhierarchie ★ ★ ☆




Wir betrachten eine Typhierarchie (dargestellt in der Abbildung rechts) mit einer Klasse A und einem Erben B. Von B ist wiederum C abgeleitet.
Markieren Sie im folgenden Java-Code jeweils hinter `//`, ob der Compiler die Zeile akzeptiert („Okay“) oder ablehnt („Fehler“). Lösen Sie die Aufgabe zunächst durch eigene Überlegungen und überprüfen Sie erst später mittels Eclipse.

```
1  class A {}
2  class B extends A {}
3  class C extends B {}
4
5  public class G {
6  public void m(List<B> a, List<? extends B> b,
7                  List<? super B> c) {
8
9      a.add(new C()); //
10     b.add(new B()); //
11     c.add(new C()); //
12     a.add(new B()); //
13     b.add(new A()); //
14     c.add(new B()); //
15     a.add(new A()); //
16     b.add(null);    //
17     c.add(new A()); //
18     b.add(new C()); //
19
20
21   }
22
23   public static void main(String args[]) {
24     m(new Vector<B>(), new Vector<C>(), new Vector<A>());
25   }
26 }
```

## V6 A-well-a bird bird bird, bird is the word Part II ★ ★ ☆

Wir erweitern unsere Typhierarchie für Vögel aus Aufgabe V3 und betrachten neben den nicht-fliegenden Vögeln nun auch ihre flatternden Artgenossen.

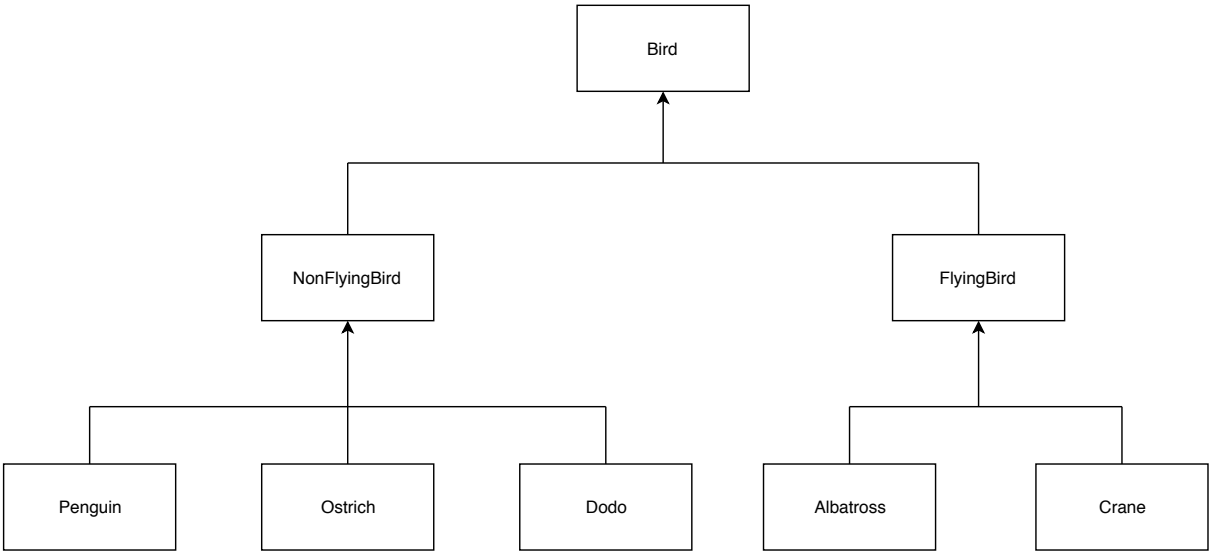



Abbildung 2: Erweiterte Typhierarchie

Vervollständigen Sie die untenstehenden Deklarationen der Methoden. Dabei sind nur die mit `.....` markierten Stellen zu bearbeiten! Geben Sie zu jeder Typangabe eine kurze Erklärung, warum genau diese Typangabe die am besten passende oder korrekte ist. *Ihre Lösungen sollen möglichst weitgehend die Typsicherheit garantieren, aber gleichzeitig flexibel für möglichst viele konkrete Parametertypen sein.*

Es sind immer generische **Sub**typen zu nutzen.

**Hinweis**: Zur Vereinfachung wird in den Beispielen nicht auf `null` oder leere Liste getestet.

(1): Die Methode `getFirst (List<.....> aListOfBirds)` liefert den ersten Vogel einer (nicht-leeren) Liste. Das Ergebnis muss kompatibel zum Typ `Bird` sein.

```
Bird getFirst(List<.....> aListOfBirds){
  return aListOfBirds.get(0);
}
```

(2): Die Methode `void add(Birds b, List<.....> aListOfBirds)` fügt einen neuen Vogel in die Liste ein. Es sollen dabei Vögel jedes bekannten Typs eingefügt werden können.

```
void add(Bird b, List<.....> aListOfBirds)
  aListOfBirds.add(b);
}
```

## V7 Array Utility-Klasse ★★☆

In dieser Aufgabe wollen wir eine bereits vorhandene Utility-Klasse, für Arrays vom Datentyp `int`, so umschreiben, dass diese für jeden beliebigen Datentyp verwendet werden kann. Die Klasse `ArrayUtils` implementiert folgende Methoden:

`void printArray(int[] array)` bekommt ein `int`-Array übergeben und gibt dessen Elemente auf der Konsole aus.

```
1 public static void printArray(int[] array) {
2   for (int i = 0; i < array.length; i++) {
3         System.out.print(array[i]);
4         if (i < array.length - 1) {
5               System.out.print(" ; ");
6         }
7   }
8   System.out.println(); }
```

`int getArrayIndex(int[] array, int value)` bekommt ein `int`-Array und einen Wert übergeben und durchsucht das Array nach dem übergebenen Wert. Wird der Wert gefunden, wird dessen Index im Array zurückgegeben, andernfalls -1.

```
1 public static int getArrayIndex(int[] array, int value) {
2   for (int i = 0; i < array.length; i++) {
3         if (array[i] == value) {
4               return i;
5         }
6   }
7   return -1;  }
```

`void simpleSort(int[] array)` sortiert das übergebene `int`-Array in aufsteigender Reihenfolge.

```
1 public static void simpleSort(int[] array) {
2   for (int i = 0; i < array.length; i++) {
3         for (int j = i + 1; j < array.length; j++) {
4               if (array[i] > array[j]) {
5                     int backup = array[i];
6                     array[i] = array[j];
7                     array[j] = backup;
8               }
9 } } }
```

Schreiben Sie nun eine Klasse `GenericArrayUtils`, die alle drei oben genannten Methoden mit einem beliebigen Datentyp `T`, bzw. `T[]` für Arrays, implementiert.

**Hinweis:** Überlegen Sie sich, wie Sie Elemente vom Typ T miteinander vergleichen können, bzw. welche Voraussetzung dieser Typ `T` mit sich bringen muss und wie sich dies im Klassenkopf der zu implementierenden Klasse widerspiegelt.

## V8 XYZ ★ ★ ★

Gegeben seien die folgenden Klassen:

```
1  public class Pair<X, Y> {
2    public X x;
3    public Y y;
4
5    public Pair(X x , Y y) {
6      this.x = x;
7      this.y = y;  }
8  }
9
10 public class Triple<X, Y, Z> {
11   public X x;
12   public Y y;
13   public Z z;
14
15   public Triple(X x , Y y, Z z) {
16     this.x = x;
17     this.y = y;
18     this.z = z;  }
19 }
20
21 public class Utils {  }
```

Erweitern Sie nun die Klasse `Utils` um eine `public`-Klassenmethode `intoMap`, ohne dabei den Klassenkopf zu modifizieren. Die Methode bekommt eine `java.util.List` von Triplen übergeben und gibt eine `java.util.Map` zurück. Jedes Triple in der Liste wird in die `Map` überführt, indem Sie die `x`-Variable des Triples als Schlüssel verwenden und ein neues Paar aus der `y`- und `z`-Variable des Triples erstellen, um dies als Wert des zugehörigen Schlüssels zu verwenden.

## V9 Matrizenmultiplikation Reloaded Reloaded ★ ★ ★

Ja Sie haben richtig gelesen, die gute, alte Matrizenmultiplikation kommt wieder einmal zurück (diesmal ist auch wirklich das letzte Wiedersehen). Neben Ihrer Implementierung in Racket und Java wollen wir in dieser Aufgabe eine Matrix Klasse implementieren, die es uns erlaubt jeden beliebigen vergleichbaren Datentyp unter Anwendung von Java Generics mit ihr zu verwenden.

Um Ihnen die Aufgabe zu erleichtern, wird ihnen ein Interface bereitgestellt, welches benutzt werden soll um arithmetische Operationen mit den Matrizen durchzuführen. Bevor man einen Datentyp mit unserer Matrix Klasse benutzen kann, muss man zuvor das arithmetische Interface für diesen konkreten Datentyp implementieren. Machen Sie sich mit den Beispielen auf der folgenden Seite vertraut.

**Generisches Interface:**

```java
public interface Arithmetic<T> {

  /**
   * returns the representation of zero
   */
  T zero();

  /**
   * Returns the result of the addition of a and b
   */
  T add(T a, T b);

  /**
   * Returns the result of the multiplication of a and b
   */
  T mul(T a, T b);

}
```

**Konkrete Implementierung für Gleitkommazahlen mit dem Datentyp Float:**

```java
public class FloatArithmetic implements Arithmetic<Float> {

  @Override
  public Float zero() {
        return 0f;
  }

  @Override
  public Float add(Float a, Float b) {
        return a + b;
  }

  @Override
  public Float mul(Float a, Float b) {
        return a * b;
  }

}
```

Bearbeiten Sie ausgehend davon die Aufgaben auf der nächsten Seite.

### V9.1 Erstellen der Matrix-Klasse

Erstellen Sie zunächst die Klasse `public class Matrix<T extends Comparable<T>>`, die die folgenden aufgezählten Variablen besitzt:

- `private Arithmetic<T> arithmetic` ist zuständig für das Durchführen von arithmetischen Operationen.
- `private LinkedList<LinkedList<T>> data` enthält die Daten der Matrix. Hierbei repräsentiert die äußere LinkedList die Reihen und die innere LinkedList die Spalten der Matrix.
- `private int rows` wird zum speichern der aktuellen Anzahl der Reihen der Matrix verwendet.
- `private int columns` wird zum speichern der aktuellen Anzahl der Spalten der Matrix verwendet.

Implementieren Sie nun folgende Methoden:

- `public Matrix(int rows, int columns, Arithmetic<T> arithmetic)` erhält die gewünschte Größe der Matrix in Form von Reihen und Spalten und ein entsprechendes Objekt dessen Klasse das arithmetische Interface implementiert. Alle übergebenen Variablen dieser Methode sollen in den entsprechenden Klassenvariablen gespeichert werden. Zusätzlich soll jeder Zellenwert mit `arithmetic.zero()` initialisiert werden.
- `public int getRows()` gibt die aktuelle Anzahl der Reihen der Matrix zurück.
- `public int getColumns()` gibt die aktuelle Anzahl der Spalten der Matrix zurück.
- `public T getCell(int row, int column)` erhält den Index einer Reihe sowie den Index einer Spalte und gibt den Wert der Zelle zurück.
- `public void setCell(int row, int column, T value)` erhält den Index einer Reihe sowie den Index einer Spalte und einen gewünschten Wert und setzt diesen an der entsprechenden Stelle in der Matrix ein.

### V9.2 Addition und Multiplikation

Implementieren...

... Sie nun die Methode `public Matrix<T> add(Matrix<T> other)`. Diese erhält eine andere Matrix, addiert die übergebene Matrix und die Matrix, auf der die Methode aufgerufen wurde, miteinander und gibt das Ergebnis der Addition zurück. Sollten die Reihen- und/oder Spaltenanzahl der beiden Matrizen nicht übereinstimmen, soll `null` zurückgegeben werden.

... Sie nun die Methode `public Matrix<T> mul(Matrix<T> other)`. Diese erhält eine andere Matrix, multipliziert die Matrix, auf der die Methode aufgerufen wurde, und die übergebene Matrix miteinander und gibt das Ergebnis der Multiplikation zurück. Sollte die Spaltenanzahl der aktuellen Matrix sich von der Reihenanzahl der übergebenen Matrix unterscheiden, soll `null` zurückgegeben werden.

# H Achte Hausübung — Gesamt 8 Punkte
## *US-amerikanische Politik*

In dieser achten Hausübung machen wir einen Ausflug in die Politik der Vereinigten Staaten von Amerika. In den beiden Aufgaben werden Sie mittels Generics und Collections zwei völlig unterschiedliche Problemstellungen bearbeiten.

## H1 Stringbuilder — 0 Punkte

In den Aufgaben H2 und H3 wird Ihnen der sogenannte `StringBuilder` begegnen, dessen Funktionalität schnell erklärt ist. Mithilfe dieser Klasse können Sie schnell und einfach Strings konkatenieren, ohne dabei den `+`-Operator verwenden zu müssen. Dies ist zum einen wesentlich schneller sollten viele Konkatenationen stattfinden und ist zum anderen meist übersichtlicher.

Wir legen zu Beginn immer ein neues Objekt der Klasse `StringBuilder` an. Mittels der `append(String)`-Methode können einfach Strings zum `StringBuilder` hinzugefügt werden. Mit der `toString()`-Methode kann der Builder schließlich in einen String umgewandelt werden.

An einem Beispiel wird dies schnell deutlich:

**Variante 1 ohne Stringbuilder**

```
String s = "";
s = s + "Das " + "ist " + "ein String!";
```

**Variante 2 mit Stringbuilder**

```
StringBuilder sb = new StringBuilder();
sb.append("Das ").append("ist ").append("ein String!");
String s = sb.toString();
```

Für weitere Informationen schauen Sie hier:

https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

**Hinweis:**

Der String `"\n"` kann für einen Zeilenumbruch verwendet werden!

So wird der Aufruf von

```
System.out.print("Achtung \n Umbruch");
```

auf der Konsole folgendermaßen ausgegeben

```
Achtung
Umbruch
```

## H2 Huntington-Hill-Methode 3 Punkte

Die Vereinigten Staaten von Amerika haben ein Zweikammer-Parlaments-System zu dem neben dem Senat auch das Repräsentantenhaus gehört. Letzteres beinhaltet $H$ Sitze, welche alle 10 Jahre neu unter den Mitgliedsstaaten verteilt werden. Eine der wichtigsten Aufgaben des Repräsentantenhauses ist die maßgebliche Beteiligung an der Gesetzgebung. Es besitzt das alleinige Initiativrecht bei Steuer- und Haushaltsgesetzen und kann außerdem als einzige Institution Amtsenthebungsverfahren gegen Amtsträger einleiten.

Die $H$ Sitze im US-amerikanischen Repräsentantenhaus werden auf die $N$ Staaten mittels der Huntington-Hill-Methode aufgeteilt. Dieses geht wie folgt vor:

1. Zu Beginn erhält jeder Staat initial einen Sitz.
2. Danach bilden wir für jeden Staat eine Priorität, die sich mittels der Population $p$ und der aktuell zugeteilten Anzahl an Sitzen $n$ des Staates wie folgt berechnet:

$$\frac{p}{\sqrt{n(n+1)}}$$

3. Nun erhält der Staat mit der höchsten Priorität einen Sitz mehr.
4. Solange es noch Sitze zu verteilen gibt, gehe wieder zu 2. zurück.

In dieser Aufgabe sollen Sie die Huntington-Hill-Methode umsetzen und dabei Verwendung von Collections machen.

**Verbindliche Anforderung**: Sie dürfen keine neue Klassen hinzufügen und dürfen den bereits existierenden Klassen der Vorlage keine Klassenvariablen hinzufügen.

**Tests:**

In dieser Aufgabe sind Ihnen bereits Tests in der Klasse `HuntingtonHillStudentTests` gegeben. Sie können diese beliebig mit Tests Ihrer Wahl ergänzen, sind dazu in dieser Aufgabe allerdings nicht verpflichtet. Zum Testen steht eine Datei `USPopulation.txt` in der Vorlage bereit, welche reale Daten aus dem Jahr 2010 enthält.

### H2.1 Staaten- und Fehlerklasse 0 Punkte

Ergänzen Sie die Klasse `State` um die Methode `public double priority()`, die die Priorität mittels der aktuellen Anzahl an zugewiesenen Sitzen des jeweiligen Staates und der gegebenen Population berechnet.

Ergänzen Sie außerdem die Klasse `MoreStatesThanSeatsException`, welche von `Exception` erbt. Ihr Konstruktor soll den Konstruktor der Oberklasse mit der Nachricht `"There are more States than Seats available!"` aufrufen.

### H2.2 Daten einlesen 1 Punkt

Ergänzen Sie nun den Konstruktor der Klasse `HuntingtonHill`. Dieser bekommt einen String übergeben, welcher den Dateinamen einer einzulesenden Text-Datei darstellt, sowie die insgesamt zu verteilende Sitzanzahl. In der Text-Datei steht in jeder Zeile der Name eines Staates und seine Population durch ein Semikolon getrennt. Also beispielsweise:

```
Maryland;5789929
Massachusetts;6559644
Michigan;9911626
Minnesota;5314879
```

Es wurde bereits ein Attribut `private HashMap<String, State> states` in der Klasse angelegt. Der Konstruktor soll nun jede Zeile der einzulesenden Datei in ein `State`-Objekt umwandeln. Fügen Sie jedes `State`-Objekt als Wert in die `HashMap` ein, indem Sie den Name des jeweiligen Staates als Schlüssel benutzen.

### H2.3 Staaten verteilen 1 Punkt

Ergänzen Sie nun die Methode `distributeSeats()` der Klasse `HuntingtonHill`, welche eine Exception des Typs `MoreStatesThanSeatsException` werfen kann. Die Methode soll nun die Sitze an die Staaten, welche in der Hashmap gespeichert wurden, nach der Huntington-Hill-Methode verteilen. Sollten mehr Staaten als Sitze vorhanden sein, soll eine entsprechende Exception geworfen werden. Die Anzahl an zu verteilenden Sitzen wurde im Konstruktor übergeben.

### H2.4 Ergebnisse printen 1 Punkt

Ergänzen Sie außerdem die Methode `printDistribution()`, welche einen `String` zurückliefert. Die Verteilung soll immer in folgendem Format zurückgegeben werden, welches anhand des obigen Vier-Staaten Beispiels und 6 zu verteilenden Sitzen verdeutlicht wird:

```
Distributed 6 seats to 4 states.

Massachusetts: 2
Michigan: 2
Maryland: 1
Minnesota: 1
```

Zu Beginn wird also zunächst genannt, wie viele Sitze, an wie viele Staaten verteilt wurden. Nach zwei Absätzen werden dann die Staaten nacheinander aufgelistet mit Ihrer zugewiesenen Sitzanzahl. Die Staaten sind dabei zuerst nach Ihrer Anzahl an Sitzen absteigend und danach lexikographisch aufsteigend geordnet.

## H3 Das Sicherheitsprotokoll 5 Punkte

In dieser Aufgabe beschäftigen wir uns mit einer (stark vereinfachten) Modellierung einer Rettungsmission bei akut drohender Gefahr der US-amerikanischen Regierung. Die Grundidee ist dabei, dass die Mitglieder der Regierung gemäß Ihrer Priorität in eine Rangliste geordnet werden und dann in dieser Reihenfolge nacheinander in Sicherheit gebracht werden.

### H3.1 Bestandsaufnahme 0 Punkte

Alle Klassen sind bereits vorhanden und müssen nur gegebenenfalls von Ihnen ergänzt werden.

Werfen Sie jedoch einen Blick auf die drei Enum-Typen `SecurityLevel`, `AlertLevel` und `Sex`. In diesen drei Dateien ändern Sie nichts mehr ab!

`Sex` enthält die Geschlechter männlich und weiblich, `SecurityLevel` die Sicherheitsstufen hoch, mittel und niedrig und `AlertLevel` das Bedrohungslevel nach dem Homeland Security Advisory System.

### H3.2 Regierungsmitglieder 1 Punkt

Zunächst wollen wir die Regierungsmitglieder modellieren.

Die abstrakte Klasse `GovernmentEmployee` ist bereits angelegt und enthält:

- Die `protected`-Attribute `String name` und `Sex sex`
- Die abstrakten Methoden `SecurityLevel getSecurityLevel()` und `String getTitle()`
- Die bereits implementierte Methode `String toString()`

Ihre Aufgabe ist es nun zunächst die drei Unterklassen `President`, `Secretary` und `Other` zu ergänzen, die alle von `GovernmentEmployee` erben.

Die Konstruktoren der drei Klassen bekommen jeweils einen `String` und ein `Sex` gegeben und setzen diese auf die jeweiligen Attribute der Oberklasse.

Die Methode `getSecurityLevel()` gibt das Sicherheitslevel der Personengruppe zurück, also `SecurityLevel.HIGH` (`President`), `SecurityLevel.MEDIUM` (`Secretary`) und `SecurityLevel.LOW` (`Other`).

Die Methode `compareTo()` erhält jeweils ein `GovernmentEmployee`-Objekt. Hierbei soll gelten: wenn der übergebene Regierungsangestellte ein höheres Sicherheitslevel hat als der aktuelle, wird der Wert -1 zurückgegeben, bei gleicher Priorität 0, ansonsten 1.

Die Methode `getTitle()` liefert die korrekte Anrede zurück. Diese besteht entweder aus `Madame` oder `Mister` zu Beginn, gefolgt von einem eventuellen Titel (bei den Klassen `President` und `Secretary` ist dieser deckungsgleich mit dem Klassennamen, bei der Klasse `Other` entfällt der Titel) und dem Namen des Regierungsmitgliedes.

### H3.3 PriorityQueue 1 Punkt

Eine PriorityQueue ist stets nach den Prioritäten ihrer einzelnen Elemente geordnet. Fügen Sie ein neues Element ein, so rückt es gemäß seiner Priorität an die jeweilige Stelle in der Queue. Die Java Bibliothek bietet bereits eine Klasse PriorityQueue an, wir wollen in dieser Aufgabe jedoch unsere eigene implementieren. Sie dürfen also selbstverständlich andere bereits vorhandenen Implementierungen von Queues (AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue) **nicht** benutzen. Überlegen Sie sich allerdings, wie Sie die Elemente der Queue speichern wollen. Wir verwenden Generics in dieser Aufgabe, um die Implementierung unabhängig zu gestalten und auch in einem anderen Anwendungsfall als unserem zu verwenden.

Die Klasse soll dabei folgende Methoden bereitstellen:

- Der Konstruktor, welche eine leere Queue erstellt.
- `void enqueue(T e)` soll ein neues Element in die Queue einfügen. Dabei soll die Priorität der Elemente beachtet werden. Dabei steht das Element mit der höchsten Priorität ganz am Anfang der Queue. Bei gleicher Priorität wird das Element als letztes dieser Prioritätsklasse eingefügt, es gibt also kein Vordrängeln. Zum Vergleich dieser Elemente verwenden Sie die Methode `compareTo`.
- `public T dequeue()` entfernt das Element mit der größten Priorität aus der Queue und gibt dieses zurück.
- `int getSize()` gibt die Anzahl der Elemente in der Queue zurück.
- `public String toString()`: liefert einen String, der in absteigend sortierter Reihenfolge alle Elemente der Queue enthält. Dabei soll die Methode `toString()` der Elemente verwendet werden. Jeder Eintrag soll die Form `"#i: xyz\n"` haben, wobei i für die konkrete Position in der Queue (beginnend ab Position 0) und xyz für die Ausgabe der toString()-Methode des Elements steht.
- `void addToHeadOfPriorityClass(T e)` schaut ob das übergebene Element bereits in der Queue ist. Wenn ja wird es an der aktuellen Position gelöscht und in seiner jeweiligen Prioritätenklasse an die oberste Stelle gesetzt. Die Queue bleibt also gemäß der Prioritäten geordnet, nur innerhalb einer Prioritätenklasse rückt ein Element nach vorne.

### H3.4 EmergencyQueue 1 Punkt

Ergänzen Sie nun die Klasse `EmergencyQueue<T extends GovernmentEmployee>`, welche bereits ein Attribut `private PriorityQueue<GovernmentEmployee> queue` enthält:

- `public void rescue(int n)`: Die Methode soll die ersten `n` Mitglieder aus der Queue entfernen und damit in Sicherheit bringen. Geschieht dies soll eine Nachricht auf der Konsole ausgegeben werden, bei dem zuerst der Name über die `getTitle()`-Methode genannt wird und anschließend die Meldung `" was rescued"` folgt. Nach jedem geretteten Mitarbeiter folgt ein Zeilenumbruch.

- Die drei Methoden:
  - `public void chooseDesignatedSurvivor(Secretary ds)`: Bestimmt einen Minister als Designated Survivor und gibt ihm damit automatisch die höchste Priorität aller Minister und aktualisiert dementsprechend die Position in der Queue.
  - `public void enqueue(GovernmentEmployee newItem)`: Ordnet einen neuen Regierungsvertreter in die Queue ein.
  - `public String toString()`: Ruft die `toString()`-Methode des `queue`-Objektes auf.

  rufen die jeweiligen Methoden auf dem Attribut `queue` auf.

### H3.5 SecurityAgency 1 Punkt

Zum Schluss kümmern wir uns um die Klasse `SecucrityAgency`, welche zuständig für die Kontrolle des Sicherheitszustands der USA ist. Die Klasse besitzt bereits ein `private`-Attribut `al` vom Typ `AlertLevel`, welche die aktuelle Gefährdungslage ausdrückt. Der Konstruktor soll dieses Level bei der Initialisierung zunächst auf `AlertLevel.LOW` setzen. Außerdem ist bereits ein `private`-Attribut `emergency` vom Typ `EmergencyQueue<GovernmentEmployee>` gegeben, sowie ein `private`-Attribut `emps` vom Typ `ArrayList<GovernmentEmployee>`, in dem alle Mitarbeiter gespeichert werden sollen.

Ergänzen Sie die Methode `add(GovernmentEmployee e)`, welche einen neuen Mitarbeiter zu `emps` hinzufügt.

Implementieren Sie die Methode `public void changeAlertLevel(AlertLevel newAL)`, welche das Alarmlevel neu einstuft. Dabei gilt folgendes:

- Beträgt das Level mindestens `AlertLevel.ELEVATED`, so werden alle `Presidents` aus `emps` in die EmergencyQueue eingefügt.
- Beträgt das Level mindestens `AlertLevel.HIGH`, so werden alle `Presidents` und `Secretarys` aus `emps` in die EmergencyQueue eingefügt.
- Beträgt das Level `AlertLevel.SEVERE`, so werden alle aus `emps` in die EmergencyQueue eingefügt.

Sie brauchen hier nur die Hochstufung des Sicherheitslevels zu betrachten, also nur Fälle bei denen von einem niedrigen in ein höheres Sicherheitslevel eingestuft wird.

### H3.6 Testen 1 Punkt

Implementieren Sie abschließend eine Testklasse, die das korrekte Verhalten der EmergencyQueue und der PriorityQueue testet.

Schreiben Sie zu jeder Methode der beiden Klassen mindestens 2 Tests!

Ihnen stehen bereits einige Beispielregierungsmitglieder bereit.