Prof. Dr. J. Giesl
D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Tutoraufgabe 1 (Überblickswissen):

a) Sammeln Sie verschiedene "populäre" Java-Exceptions, z.B. in der Vorlesung eingeführte Exceptions oder anderswo her bereits bekannte Exceptions, und diskutieren Sie, ggf. an Beispielen, wann und warum die jeweiligen Exceptions auftreten.

b) In der Vorlesung haben Sie zwei Arten von allgemeinen Listen kennengelernt. Einmal können generische Typen benutzt werden, um eine solche allgemeine Liste zu realisieren. Die zweite Implementation hingegen benutzt Werte vom Typ `Object`. Was sind die Nachteile dieser Umsetzung?

c) Wofür benötigt ein*e Java-Entwickler*in Module und Pakete?

d) Woraus besteht das Collection-Framework[1]? Warum ist es in den meisten Fällen sinnvoll, ein solches Framework zu verwenden und nicht eigene Implementierungen?

Lösung:

a) Aus der Vorlesung:

- `IOException`: Fehler in Ein- oder Ausgabe
- `ArithmeticException`: z.B. `x/0` für `int x`
- `ArrayIndexOutOfBoundsException`: Überschreiten des Indexbereichs eines Arrays
- `ClassCastException`: Fehlschlag bei expliziter Konversion von Ober- zu Unterklasse
- `NumberFormatException`: Versuch, einen String, der keine gültige Zahl enthält, in Zahl umzuwandeln
- `NullPointerException`: Versuch, auf Objektvariable über `null`-Verweis zuzugreifen

Weitere Exceptions:

- `NegativeArraySizeException`: Versuch, ein Array mit negativer Größe zu erzeugen.
- `OutOfMemoryError`: VM kann keinen Speicher mehr allozieren, da kein weiterer Speicher mehr verfügbar ist und der Garbage Collector keinen Speicher freimachen kann.
- `StackOverflowError`: Wird geworfen, falls Stack Overflow auftritt, verursacht durch zu tiefe Rekursionen.

b)

- Es sind keine Anforderungen an Werte ausdrückbar. Abhilfe: Verwende abstrakte Klasse oder Interface anstelle von `Object`.
- Verschiedene Objekt-Typen in der gleichen Liste.
- Wenn man Werte z.B. vom Typ `T` aus der Liste ausliest, dann sind es zunächst Werte der Oberklasse (also z.B. `Object` oder `Vergleichbar`). Die Werte müssen, damit man mit ihnen weiterarbeiten kann, erst explizit in die Unterklasse `T` "gecasted" werden.

c) Pakete ermöglichen eine feinere Datenkapselung (`protected`) und erlauben eine sinnvolle Struktur, insbesondere von großen Projekten.

Große Anwendungen können in verschiedene Module unterteilt werden. Module ermöglichen eine erweiterte Datenkapselung. Seit Java 9 können Anwendungen als Module organisiert sein. In einer Datei `module-info.java` kann eine Moduldeklaration angelegt werden. In dieser werden mit dem Keyword `requires` benötigte Module aufgelistet. Mit einem weiteren Keyword `exports` können Sie angeben, dass auf das spezifizierte Pakete von anderen Modulen zugegriffen werden kann. Listen Sie ein Paket nicht mit `exports` auf, so kann nicht auf das Paket von anderen Modulen zugegriffen werden.

[1] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collection.html`

d) "The Java platform includes a collections framework. A collection is an object that represents a group of objects (such as the classic Vector class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details." (vgl. `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`)

Das Collection-Framework besteht also aus Klassen und Methoden, um eine Kollektion/Menge von Objekten zu gruppieren und zu verwalten. Das Framework stellt eine Sammlung von Algorithmen zur Verfügung, die auf einer solchen Kollektion operieren. Beispiele für solche Algorithmen wäre ein Sortieralgorithmus (`sort` [2]) oder ein Suchalgorithmus (`binarySearch` [3]). Die meisten häufig benötigten Sammlungs-Datenstrukturen sind bereits im Collection-Framework implementiert. Die Implementierung wird von vielen Benutzern verwendet und ist oft[4] fehlerfrei. Außerdem ist die Implementierung meist auch möglichst effizient. Daher ist es oft unnötig, als Entwickler*in, Zeit in eigene Implementationen zu investieren. Es ist natürlich dennoch, gerade für Anfänger*innen, sinnvoll, einmal selbst die wichtigsten Datenstrukturen zu implementieren.

Tutoraufgabe 2 (Collections, Exceptions und Generics):

In dieser Aufgabe geht es um die Implementierung einer Datenstruktur für Mengen, welche in das bestehende Collection-Framework eingebettet werden soll. Sie benötigen dafür die Klassen `EmptySet`, `AddSet`, `RemoveSet`, `FunctionalSet`, `SimpleFunctionalSet` und `Main`, welche Sie als `.java` Dateien im Moodle-Lernraum herunterladen können.
Die in dieser Aufgabe zu betrachtende Mengenstruktur basiert auf einer Liste von Einfüge- (`Add`) und Löschoperationen (`Remove`) mit jeweils einem Element, die vom Ausgangspunkt einer leeren Menge (`Empty`) angewendet werden. Zum Beispiel lässt sich die Menge $\{1, 2, 3\}$ als die Liste `Add 3, Add 2, Add 1, Empty` darstellen. Will man nun das Element 2 aus der Menge löschen, so entfernt man nicht das zweite Element aus der Liste, sondern fügt ein weiteres `Remove` Element hinzu und erhält `Remove 2, Add 3, Add 2, Add 1, Empty`. Auf diese Weise erhält man eine Datenstruktur, bei der niemals Objekte entfernt werden (mit Ausnahme der `clear` Methode, welche die Liste wieder auf `Empty` setzen soll).

a) Die vorgegebene Klasse `FunctionalSet` implementiert bereits teilweise das `Set` Interface. Ergänzen Sie die Klasse um einen Konstruktor `FunctionalSet()` und eine geeignete `toString`-Methode. Der Konstruktor soll eine leere Menge erzeugen. Erweitern Sie das Interface außerdem um sinnvoll implementierte Methoden `boolean add(E e)`, `boolean remove(Object o)`, `boolean addAll (Collection <? extends E > c)` und `boolean removeAll(Collection <?> c)`. Hierbei gibt der Rückgabewert an, ob die Methode die Datenstruktur verändert hat. Schreiben Sie zuletzt noch eine Methode `void clear()`, die eine entsprechende Menge `this` leert.

b) Die Methode `iterator` benötigt die generische Klasse `FunctionalSetIterator<E>`, welche das Interface `Iterator<E>` aus dem Package `java.util` implementiert. Schreiben Sie diese generische Klasse.

Schlagen Sie für die zu implementierenden Methoden `hasNext`, `next` und `remove` die Funktionalitäten in der Java API für das Interface `Iterator` nach (die `remove` Operation soll durch Ihren Iterator unterstützt werden, die Methode `forEachRemaining` brauchen Sie hingegen nicht zu implementieren). Dies betrifft insbesondere auch die durch diese Methoden zu werfenden Exceptions.

c) Implementieren Sie in der Klasse `FunctionalSet` eine Methode `E min(java.util.Comparator<E> comp)`, die das kleinste in der Menge gespeicherte Element zurückliefert. Die Ordnung, die zum Vergleich zweier Elemente verwendet wird, ist durch den `Comparator comp` festgelegt. Wenn die Menge leer ist, soll die Methode eine `MinimumOfEmptySetException` werfen. Implementieren Sie zu diesem Zweck eine Klasse `MinimumOfEmptySetException`, die von `java.lang.RuntimeException` erbt.

[2] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html#sort(java.util.List)`
[3] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html#binarySearch(java.util.List,T)`
[4] `https://de.wikipedia.org/wiki/Timsort#Bekannte_Fehler`

d) Sie können die `main` Methode der Klasse `Main` nutzen, um Ihre Implementierung zu testen. Allerdings stürzt diese ab, wenn Sie z.B. `add k` oder `remove k` eingeben, da `k` keine Zahl ist und das Parsen von `k` folglich mit einer `java.lang.NumberFormatException` scheitert. Die Methode stürzt ebenfalls ab, wenn Sie `min` eingeben, ohne vorher Elemente zu der Menge hinzuzufügen (indem Sie z.B. `add 2` eingeben). In diesem Fall ist der Grund eine `MinimumOfEmptySetException`. Fangen Sie diese Exceptions mit `try-catch`, um Programmabstürze zu verhindern, und geben Sie stattdessen geeignete Fehlermeldungen aus.

Lösung:

Listing 1: FunctionalSetIterator.java

```
import java.util.*;

/**
 * @param <E> Element type.
 * Iterator through a functional set.
 */
public class FunctionalSetIterator<E> implements Iterator<E> {

    /**
     * The current simple functional set. It is always an add set with an unused (i.e., not contained in
     * the set named "used" below) element or an empty set.
     */
    private SimpleFunctionalSet<E> current;

    /**
     * The most recent element this iterator has returned.
     */
    private E recentElem;

    /**
     * Flag indicating whether the remove operation is applicable (needed since elements may be null, so
     * we cannot just check whether recentElem is null).
     */
    private boolean removable;

    /**
     * The functional set to which the current simple functional set belongs.
     */
    private final FunctionalSet<E> set;

    /**
     * A set of already seen elements (in add or remove sets).
     */
    private final Set<Object> used;

    /**
     * @param functionalSet The functional set containing the simple functional set to iterate over.
     * @param head The head of the simple functional set to iterate over.
     */
    public FunctionalSetIterator(
        FunctionalSet<E> functionalSet,
        SimpleFunctionalSet<E> head)
    {
        this.current = head;
        this.recentElem = null;
        this.removable = false;
        this.set = functionalSet;
        this.used = new FunctionalSet<Object>();
        this.forwardToNextUnusedSet();
    }

    @Override
    public boolean hasNext() {
        return !(this.current instanceof EmptySet);
    }

    @Override
    public E next() {
        if (this.hasNext()) {
            E elem = ((AddSet<E>) this.current).getElement();
            this.used.add(elem);
            this.recentElem = elem;
            this.removable = true;
            this.current = this.current.getRemainingSet();
            this.forwardToNextUnusedSet();
            return elem;
```

```
        } else {
            throw new NoSuchElementException();
        }
    }

    @Override
    public void remove() {
        if (this.removable) {
            this.set.remove(this.recentElem);
            this.removable = false;
        } else {
            throw new IllegalStateException(
                "The next method has not been called before this remove operation!");
        }
    }

    /**
     * Forwards the current set to the next remaining set which is no remove set and which is no add set
     * with an already used element. During forwarding, used objects are added to the corresponding set.
     */
    private void forwardToNextUnusedSet() {
        boolean loop = true;
        while (loop) {
            loop = false;
            while (this.current instanceof RemoveSet<E> rs) {
                this.used.add(rs.getObject());
                this.current = this.current.getRemainingSet();
            }
            if (this.current instanceof AddSet<E> as) {
                if(this.used.contains(as.getElement())) {
                    loop = true;
                    this.current = this.current.getRemainingSet();
                }
            }
        }
    }

}
```

Listing 2: FunctionalSet.java

```
import java.util.*;

/**
 * Functional data structure for a set.
 * @param <E> Element type.
 */
public class FunctionalSet<E> implements Set<E> {

    /**
     * The head of the list of operations representing the set.
     */
    private SimpleFunctionalSet<E> head;

    public String toString() {
        String res = "{";
        Iterator<E> it = iterator();
        while (it.hasNext()) {
            res = res + it.next();
            if (it.hasNext()) {
                res = res + ", ";
            }
        }
        return res + "}";
    }

    public E min(Comparator<E> comp) throws MinimumOfEmptySetException {
        if (isEmpty()) {
            throw new MinimumOfEmptySetException();
        } else {
            E res = null;
            for (E e: this) {
                if (res == null || comp.compare(e, res) < 0) {
                    res = e;
                }
            }
            return res;
        }
    }

    /**
     * Creates an empty functional set.
     */
```

```
public FunctionalSet() {
    this.head = new EmptySet<E>();
}

@Override
public boolean add(E e) {
    if (this.contains(e)) {
        return false;
    } else {
        this.head = new AddSet<E>(e, this.head);
        return true;
    }
}

@Override
public boolean addAll(Collection<? extends E> c) {
    boolean res = false;
    for (E elem : c) {
        res |= this.add(elem);
    }
    return res;
}

@Override
public void clear() {
    this.head = new EmptySet<E>();
}

@Override
public boolean contains(Object o) {
    return this.head.contains(o);
}

@Override
public boolean containsAll(Collection<?> c) {
    for (Object o : c) {
        if (!this.contains(o)) {
            return false;
        }
    }
    return true;
}

@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    } else if (o == null || o.getClass() != this.getClass()) {
        return false;
    } else {
        FunctionalSet<?> set = (FunctionalSet<?>)o;
        return this.containsAll(set) && set.containsAll(this);
    }
}

@Override
public int hashCode() {
    int res = 5;
    final int prime = 7;
    for (E elem : this) {
        res += prime * elem.hashCode();
    }
    return res;
}

@Override
public boolean isEmpty() {
    return this.size() == 0;
}

@Override
public Iterator<E> iterator() {
    return new FunctionalSetIterator<E>(this, this.head);
}

@Override
public boolean remove(Object o) {
    if (this.contains(o)) {
        this.head = new RemoveSet<E>(o, this.head);
        return true;
    } else {
        return false;
    }
}
```

```
    @Override
    public boolean removeAll(Collection<?> c) {
        boolean res = false;
        for (Object o : c) {
            res |= this.remove(o);
        }
        return res;
    }

    @Override
    public boolean retainAll(Collection<?> c) {
        List<E> store = new ArrayList<E>();
        boolean change = false;
        for (E elem : this) {
            if (c.contains(elem)) {
                store.add(elem);
            } else {
                change = true;
            }
        }
        if (change) {
            this.clear();
            for (E elem : store) {
                this.add(elem);
            }
            return true;
        } else {
            return false;
        }
    }

    @Override
    public int size() {
        int res = 0;
        for (Iterator<E> it = this.iterator(); it.hasNext(); it.next()) {
            res++;
        }
        return res;
    }

    @Override
    public Object[] toArray() {
        Object[] res = new Object[this.size()];
        int i = 0;
        for (E elem : this) {
            res[i] = elem;
            i++;
        }
        return res;
    }

    @SuppressWarnings("unchecked")
    @Override
    public <T>T[] toArray(T[] a) {
        final int size = this.size();
        final T[] res;
        if (a.length < size) {
            res = Arrays.copyOf(a, size);
        } else {
            res = a;
        }
        int i = 0;
        for (E elem : this) {
            try {
                res[i] = (T)elem;
            } catch (ClassCastException e) {
                throw new ArrayStoreException(
                    "Element " + elem + " cannot be cast to type of specified array!"
                );
            }
            i++;
        }
        return res;
    }

}
```

Listing 3: Main.java

```
import java.util.*;

public class Main {
```

```
    public static void main(String[] args) {
        FunctionalSet<Integer> s = new FunctionalSet<>();
        String line;
        do {
            line = SimpleIO.getString("Enter 'add i', 'remove i', 'min', or 'exit'!");
            String[] words = line.split(" ");
            switch (words[0]) {
                case "exit" -> {}
                case "add" -> {
                    s.add(Integer.parseInt(words[1]));
                    System.out.println(s);}
                case "remove" -> {
                    s.remove(Integer.parseInt(words[1]));
                    System.out.println(s);}
                case "min" ->
                    System.out.println(s.min(Comparator.<Integer>naturalOrder()));
                default ->
                    System.out.println("Unknown command.");
            }
        } while (!"exit".equals(line));
    }
}
```

Listing 4: MinimumOfEmptySetException.java

```
public class MinimumOfEmptySetException extends java.lang.RuntimeException{}
```

Tutoraufgabe 3 (Exceptions und Generics (Video)):

Diese Aufgabe behandelt erneut, wie bereits Aufgabe 4 auf dem vorherigen Blatt, einen Weihnachtsmarkt. Sie haben bereits auf Blatt 7 für Weihnachtsmärkte eine Klassenhierarchie entworfen. Auf diesem Blatt wird nun die tatsächliche Implementation behandelt. Verwenden Sie hierzu die Hilfsklassen `Zufall` und `SimpleIO`, die beide im Moodle-Lernraum zu finden sind.

- Ein Weihnachtsmarkt besteht aus verschiedenen Ständen. Ein Weihnachtsmarkt verfügt außerdem über eine Methode `run()`, die kein Ergebnis zurückgibt.
- Ein Stand kann entweder ein Weihnachtsartikelstand oder ein Lebensmittelstand sein. Jeder Stand hat eine*n Verkäufer*in, dessen Name von Interesse ist, und eine Anzahl von Besuchern*innen pro Stunde. Hierfür existiert sowohl ein Attribut `besucherProStunde` als auch eine Methode `berechneBesucherProStunde()`, um diese Anzahl neu zu berechnen. Ein Stand bietet außerdem die Methode `einzelkauf()`, welche den zu bezahlenden Preis (centgenau in Euro) zurückgibt.
- Ein Weihnachtsartikelstand hat eine Reihe an Artikeln.
- Ein Artikel hat einen Namen und einen Preis (centgenau in Euro).
- Ein Lebensmittelstand verkauft ein bestimmtes Lebensmittel.
- Ein Lebensmittel ist entweder ein Flammkuchen oder eine Süßware. Es bietet die Möglichkeit, über die Methoden `getPreisPro100g()` und `getName()`, den festen Preis pro 100 Gramm (centgenau in Euro) und den Namen abzurufen.
- Bei einer Süßware ist der Preis pro 100 Gramm (centgenau in Euro) und die Süßwarenart als String von Interesse.
- Bei einem Flammkuchen ist der Preis pro 100 Gramm (centgenau in Euro) von Interesse.
- Ein Süßwarenstand ist ein Lebensmittelstand.
- Im Gegensatz zu Flammkuchenständen, die einen festen Wasseranschluss benötigen, lassen sich Weihnachtsartikelstände und Süßwarenstände mit einer Methode `verschiebe(int)` ohne Rückgabe verschieben. Dies wird regelmäßig ausgenutzt, falls die Anzahl der Besucher*innen erhöht werden soll.

a) Klassenhierarchie: Siehe Blatt 7 Aufgabe 4

b) Implementieren Sie die Klassen entsprechend Ihrer Klassenhierarchie. Nutzen Sie dafür die Klassen `Zufall` und `SimpleIO` (aus dem Moodle-Lernraum). Fügen Sie jeder Klasse, falls notwendig, Getter und Setter sowie eine geeignete `toString`-Methode hinzu.

c) Die Klasse `Lebensmittelstand` soll einen generischen Typparameter `T` erhalten, welcher `Lebensmittel` oder eine Unterklasse von `Lebensmittel` sein kann. Das einzige Attribut `lebensmittel` der Klasse `Lebensmittelstand` soll vom Typ `T` sein. Der Konstruktor erhält einen Parameter vom Typ `T` und weist das Attribut entsprechend zu. Die Klasse `Suesswarenstand` ist Unterklasse von `Lebensmittelstand<Suessware>` und benötigt einen Konstruktor, welcher den `super`-Konstruktor mit einem `Suessware`-Objekt aufruft.

d) Initialisieren Sie die Attribute der Objekte Ihrer Klassen mithilfe von `Zufall.java` nach folgendem Schema:

- Der Konstruktur der Klasse `Weihnachtsmarkt` bekommt die Anzahl der Stände übergeben und legt ein Array mit entsprechend vielen Ständen an. Verwenden Sie die statische Methode `Zufall.zahl(int)` so, dass es jeweils etwa 33% Weihnachtsartikelstände, Süßwarenstände und Flammkuchenstände gibt. Hierbei gibt ein Aufruf `Zufall.zahl(i)` (für `i` größer 0) eine zufällige Zahl zwischen 0 und `i`-1 zurück. Ein Flammkuchenstand kann erzeugt werden, indem der `Lebensmittelstand`-Konstruktor mit einem `Flammkuchen`-Objekt aufgerufen wird.
- Der Name der Verkäuferin oder des Verkäufers eines Stands wird mit der statischen Methode `Zufall.name()` festgelegt.
- Die Anzahl der Besucher*innen pro Stunde bewegt sich zwischen 0 und 100 und soll mit der Methode `Zufall.zahl(int)` festgelegt werden. Die einzige Ausnahme bilden Weihnachtsartikelstände, bei denen sich die Anzahl der Besucher*innen pro Stunde durch die Addition von n Zufallszahlen zwischen 0 und 5 ergibt, wobei n die Anzahl der Artikel des Standes ist, die nicht `null` sind (Artikel werden später durch Verkaufen auf `null` gesetzt).
- Ein Weihnachtsartikelstand hat zwischen 1 und 20 Artikel. Sowohl die Anzahl der Artikel als auch die Artikel selbst sollen zufällig ausgewählt werden. Verwenden Sie hierfür die Methoden `Zufall.zahl(int)` und `Zufall.artikel()`, um die Anzahl, die Namen und die Preise zu bestimmen. Der Preis eines Artikels soll zwischen $0,01$ Euro und 10 Euro liegen.
- Bei einem Lebensmittel ergibt sich der Preis pro 100 Gramm als Zufallszahl zwischen $0,01$ Euro und 3 Euro.
- Zur Bestimmung der Süßwarenart soll die Methode `Zufall.suessware()` genutzt werden.

e) Implementieren Sie die in `Stand` als `abstract` markierte Methode `einzelkauf` in den Klassen `Weihnachtsartikelstand` und `Lebensmittelstand`. Weihnachtsartikelstände sollen alle erhältlichen Artikel auflisten und fragen, welchen Artikel die oder der Kunde*in kaufen möchte. Anschließend soll der gekaufte Artikel aus dem Sortiment gelöscht werden, indem der entsprechende Eintrag auf `null` gesetzt wird. Lebensmittelstände sollen nachfragen, wie viel Gramm die oder der Kunde*in haben möchte. Am Ende soll der Gesamtpreis zurückgegeben werden, den die oder der Kunde*in zahlen muss. Verwenden Sie hierbei die Methoden `SimpleIO.getInt(String)` und `SimpleIO.getBoolean(String)` zur Interaktion mit der oder dem Nutzer*in. Sie dürfen weitere private Hilfsmethoden anlegen, um den Code besser lesbar zu gestalten.

f) Implementieren Sie für verschiebbare Stände eine Methode `verschiebe(int standID)`, die die Anzahl der Besucher*innen pro Stunde neu berechnet und anschließend eine Meldung ausgibt, dass Stand `standID` verschoben wurde, zusammen mit der Information, von wie vielen Passanten*innen dieser Stand nun stündlich besucht wird.

Falls ein `Weihnachtsartikelstand` verschoben wird, so besteht die Möglichkeit, dass einer der Artikel beim Verschieben vom Stand fällt und kaputt geht. Der entsprechende Artikel soll aus dem Sortiment entfernt werden und es soll eine `SchadensfallException` geworfen werden, welche eine Nachricht enthält, die den Schadensfall beschreibt. Wählen Sie dazu beim Verschieben eines `Weihnachtsartikelstand`s zufällig einen Index aus dem `artikel`-Array aus. Ist dieser bereits ausverkauft (`null`), so passiert nichts. Ansonsten fällt dieser Artikel beim Verschieben vom Stand.

Legen Sie dazu die Klasse `SchadensfallException` an, welche von `Exception` erbt und einen Konstruktor enthält, der eine Fehlermeldung als `String` erhält und an den `super`-Konstruktor weitergibt. Fügen Sie außerdem eine entsprechende `throws`-Klausel für die `verschiebe`-Methode hinzu (auch am Interface `Verschiebbar`).

Sie dürfen weitere private Hilfsmethoden anlegen, um den Code besser lesbar zu gestalten.

g) Fügen Sie der Klasse `Weihnachtsmarkt` eine `main`-Methode hinzu. In dieser soll ein neuer Weihnachtsmarkt mit 5 Ständen erstellt werden und anschließend dessen `run`-Methode aufgerufen werden.

In der `run`-Methode beginnt die erste Runde, in der alle Stände des Weihnachtsmarktes aufgelistet werden und die oder der Nutzer*in gefragt wird, welchen Stand sie oder er besuchen möchte. An dem ausgewählten Stand soll die oder der Kunde*in solange Einzelkäufe tätigen können, bis sie oder er mit dem Einkauf fertig ist und den Stand verlässt. Am Ende soll der Gesamtpreis genannt werden, den die oder der Kunde*in zahlen muss.

Anschließend sollen alle verschiebbaren Stände, die von weniger als 30 Passanten*innen pro Stunde besucht werden, verschoben werden. Zum Ende einer Runde wird die oder der Nutzer*in gefragt, ob sie oder er den Weihnachtsmarkt verlassen möchte. Falls dies verneint wird, soll die nächste Runde beginnen.

Verwenden Sie die Methoden `SimpleIO.getInt(String)` und `SimpleIO.getBoolean(String)` zur Interaktion mit der oder dem Nutzer*in.

Sie dürfen weitere private Hilfsmethoden anlegen, um den Code besser lesbar zu gestalten.

Eine Lauf des Programms könnte beispielsweise die folgende Ausgabe erzeugen:

```
Der Weihnachtsmarkt besteht aus folgenden Staenden:

0: Lebensmittelstand fuer Flammkuchen:
Preis pro 100g: 0.91 Euro
Verkaeufer*in: Sarah
Besucher*innen pro Stunde: 21

1: Weihnachtsartikelstand:
Verkaeufer*in: Felix
Besucher*innen pro Stunde: 9

2: Lebensmittelstand fuer Suessware (Zuckerstange):
Preis pro 100g: 2.05 Euro
Verkaeufer*in: Mattis
Besucher*innen pro Stunde: 9

3: Weihnachtsartikelstand:
Verkaeufer*in: Per
Besucher*innen pro Stunde: 27

4: Lebensmittelstand fuer Suessware (Waffeln):
Preis pro 100g: 2.27 Euro
Verkaeufer*in: Fynn
Besucher*innen pro Stunde: 50

Welchen Stand moechten Sie besuchen?
0
Guten Tag!
Wie viel Gramm moechten Sie?
200
200 Gramm fuer Sie. Lassen Sie es sich schmecken!
Darf es sonst noch etwas sein?
false
1.82 Euro, bitte.
Stand 1 wurde verschoben und wird jetzt von 9 Passanten*innen pro Stunde besucht.
Dabei ist Artikel 1: Holzkrippe (3.7 Euro) leider vom Stand gefallen und kaputt gegangen.
Stand 2 wurde verschoben und wird jetzt von 26 Passanten*innen pro Stunde besucht.
```

```
Stand 3 wurde verschoben und wird jetzt von 33 Passanten*innen pro Stunde besucht.
Dabei ist Artikel 8: Tasse (9.1 Euro) leider vom Stand gefallen und kaputt gegangen.
Moechten Sie den Weihnachtsmarkt verlassen?
false
Der Weihnachtsmarkt besteht aus folgenden Staenden:

0: Lebensmittelstand fuer Flammkuchen:
Preis pro 100g: 0.91 Euro
Verkaeufer*in: Sarah
Besucher*innen pro Stunde: 21

1: Weihnachtsartikelstand:
Verkaeufer*in: Felix
Besucher*innen pro Stunde: 9

2: Lebensmittelstand fuer Suessware (Zuckerstange):
Preis pro 100g: 2.05 Euro
Verkaeufer*in: Mattis
Besucher*innen pro Stunde: 26

3: Weihnachtsartikelstand:
Verkaeufer*in: Per
Besucher*innen pro Stunde: 33

4: Lebensmittelstand fuer Suessware (Waffeln):
Preis pro 100g: 2.27 Euro
Verkaeufer*in: Fynn
Besucher*innen pro Stunde: 50

Welchen Stand moechten Sie besuchen?
1
Guten Tag!
Unsere Artikel sind:
0: Rucksack (7.01 Euro)
1: ausverkauft
2: Kette (9.18 Euro)
Welchen Artikel moechten Sie kaufen?
0
Rucksack wird eingepackt. Viel Spass damit!
Darf es sonst noch etwas sein?
false
7.01 Euro, bitte.
Stand 1 wurde verschoben und wird jetzt von 2 Passanten*innen pro Stunde besucht.
Stand 2 wurde verschoben und wird jetzt von 12 Passanten*innen pro Stunde besucht.
Moechten Sie den Weihnachtsmarkt verlassen?
false
Der Weihnachtsmarkt besteht aus folgenden Staenden:

0: Lebensmittelstand fuer Flammkuchen:
Preis pro 100g: 0.91 Euro
Verkaeufer*in: Sarah
Besucher*innen pro Stunde: 21

1: Weihnachtsartikelstand:
Verkaeufer*in: Felix
Besucher*innen pro Stunde: 2

2: Lebensmittelstand fuer Suessware (Zuckerstange):
Preis pro 100g: 2.05 Euro
Verkaeufer*in: Mattis
Besucher*innen pro Stunde: 12
```

```
3: Weihnachtsartikelstand:
Verkaeufer*in: Per
Besucher*innen pro Stunde: 33

4: Lebensmittelstand fuer Suessware (Waffeln):
Preis pro 100g: 2.27 Euro
Verkaeufer*in: Fynn
Besucher*innen pro Stunde: 50

Welchen Stand moechten Sie besuchen?
2
Guten Tag!
Wie viel Gramm moechten Sie?
20
20 Gramm fuer Sie. Lassen Sie es sich schmecken!
Darf es sonst noch etwas sein?
true
Wie viel Gramm moechten Sie?
30
30 Gramm fuer Sie. Lassen Sie es sich schmecken!
Darf es sonst noch etwas sein?
false
1.025 Euro, bitte.
Stand 1 wurde verschoben und wird jetzt von 2 Passanten*innen pro Stunde besucht.
Stand 2 wurde verschoben und wird jetzt von 2 Passanten*innen pro Stunde besucht.
Moechten Sie den Weihnachtsmarkt verlassen?
false
Der Weihnachtsmarkt besteht aus folgenden Staenden:

0: Lebensmittelstand fuer Flammkuchen:
Preis pro 100g: 0.91 Euro
Verkaeufer*in: Sarah
Besucher*innen pro Stunde: 21

1: Weihnachtsartikelstand:
Verkaeufer*in: Felix
Besucher*innen pro Stunde: 2

2: Lebensmittelstand fuer Suessware (Zuckerstange):
Preis pro 100g: 2.05 Euro
Verkaeufer*in: Mattis
Besucher*innen pro Stunde: 2

3: Weihnachtsartikelstand:
Verkaeufer*in: Per
Besucher*innen pro Stunde: 33

4: Lebensmittelstand fuer Suessware (Waffeln):
Preis pro 100g: 2.27 Euro
Verkaeufer*in: Fynn
Besucher*innen pro Stunde: 50

Welchen Stand moechten Sie besuchen?
3
Guten Tag!
Unsere Artikel sind:
0: Armband (4.87 Euro)
1: Rucksack (6.45 Euro)
2: Tasse (7.48 Euro)
3: Teelichtkarussel (8.85 Euro)
```

```
4: Uhr (4.03 Euro)
5: Kette (5.12 Euro)
6: Fensterbild (2.94 Euro)
7: Stofftier (0.88 Euro)
8: ausverkauft
Welchen Artikel moechten Sie kaufen?
2
Tasse wird eingepackt. Viel Spass damit!
Darf es sonst noch etwas sein?
true
Unsere Artikel sind:
0: Armband (4.87 Euro)
1: Rucksack (6.45 Euro)
2: ausverkauft
3: Teelichtkarussel (8.85 Euro)
4: Uhr (4.03 Euro)
5: Kette (5.12 Euro)
6: Fensterbild (2.94 Euro)
7: Stofftier (0.88 Euro)
8: ausverkauft
Welchen Artikel moechten Sie kaufen?
4
Uhr wird eingepackt. Viel Spass damit!
Darf es sonst noch etwas sein?
false
11.510000000000002 Euro, bitte.
Stand 1 wurde verschoben und wird jetzt von 3 Passanten*innen pro Stunde besucht.
Dabei ist Artikel 2: Kette (9.18 Euro) leider vom Stand gefallen und kaputt gegangen.
Stand 2 wurde verschoben und wird jetzt von 90 Passanten*innen pro Stunde besucht.
Moechten Sie den Weihnachtsmarkt verlassen?
true
```

Hinweise:

- Berücksichtigen Sie in der gesamten Aufgabe die Prinzipien der Datenkapselung und verwenden Sie Implementierungen in Oberklassen bzw. Interfaces soweit möglich.
- Vermeiden Sie betriebssystemspezifische Zeilenseparatoren wie `\n` bzw. `\r\n` in Strings. Verwenden Sie stattdessen `System.lineSeparator()`.

Lösung:

b) - g)

Listing 5: Artikel.java

```
public class Artikel {
    private final String name = Zufall.artikel();
    private final double preis = (Zufall.zahl(1000) + 1) / 100.0;

    public String getName() {
        return this.name;
    }

    public double getPreis() {
        return this.preis;
    }

    @Override
    public String toString() {
        return this.name + " (" + this.preis + " Euro)";
    }
}
```

Listing 6: Lebensmittel.java

```
public interface Lebensmittel {
    double getPreisPro100g();

    String getName();
}
```

Listing 7: Flammkuchen.java

```
public class Flammkuchen implements Lebensmittel {
    private final double preisPro100g = (Zufall.zahl(300) + 1) / 100.0;

    @Override
    public double getPreisPro100g() {
        return preisPro100g;
    }

    @Override
    public String getName() {
        return "Flammkuchen";
    }
}
```

Listing 8: Suessware.java

```
public class Suessware implements Lebensmittel {
    private final double preisPro100g = (Zufall.zahl(300) + 1) / 100.0;
    private final String suesswarenart = Zufall.suessware();

    @Override
    public double getPreisPro100g() {
        return preisPro100g;
    }

    @Override
    public String getName() {
        return "Suessware (" + suesswarenart + ")";
    }
}
```

Listing 9: Stand.java

```
public abstract class Stand {
    private final String verkaeufer;
    private int besucherProStunde;

    public Stand() {
        this.verkaeufer = Zufall.name();
        this.besucherProStunde = 0;
    }

    public int getBesucherProStunde() {
        return besucherProStunde;
    }

    public void berechneBesucherProStunde() {
        this.besucherProStunde = Zufall.zahl(101);
    }

    public void setBesucherProStunde(int besucherProStunde) {
        this.besucherProStunde = besucherProStunde;
    }

    public abstract double einzelkauf();

    public String toString() {
        return "Verkaeufer*in: " + this.verkaeufer + System.lineSeparator()
                + "Besucher*innen pro Stunde: " + this.besucherProStunde
                + System.lineSeparator();
    }
}
```

Listing 10: Weihnachtsartikelstand.java

```
public class Weihnachtsartikelstand extends Stand implements Verschiebbar {
    private Artikel[] artikel;

    public Weihnachtsartikelstand() {
        int anzahlArtikel = Zufall.zahl(20) + 1;
        this.artikel = new Artikel[anzahlArtikel];
        for (int i = 0; i < anzahlArtikel; i++) {
```

```
            this.artikel[i] = new Artikel();
        }
        this.berechneBesucherProStunde();
    }

    @Override
    public void berechneBesucherProStunde() {
        int result = 0;
        for (int i = 0; i < artikel.length; i++) {
            if (artikel[i] != null) {
                result += Zufall.zahl(6);
            }
        }
        this.setBesucherProStunde(result);
    }

    @Override
    public double einzelkauf() {
        praesentiereArtikel();
        return berechneArtikel(waehleArtikel());
    }

    private void praesentiereArtikel() {
        System.out.println("Unsere Artikel sind:");
        for (int i = 0; i < artikel.length; i++) {
            if (artikel[i] != null) {
                System.out.println(i + ": " + artikel[i].toString());
            } else {
                System.out.println(i + ": ausverkauft");
            }
        }
    }

    private int waehleArtikel() {
        int result = SimpleIO.getInt("Welchen Artikel moechten Sie kaufen?");
        System.out.println("Welchen Artikel moechten Sie kaufen?");
        System.out.println(result);
        return result;
    }

    private double berechneArtikel(int i) {
        if (i < artikel.length && artikel[i] != null) {
            System.out.println(artikel[i].getName()
                    + " wird eingepackt. Viel Spass damit!");
            double result = artikel[i].getPreis();
            artikel[i] = null;
            return result;
        } else {
            System.out.println("Diesen Artikel haben wir nicht!");
            return 0;
        }
    }

    @Override
    public void verschiebe(int standID) throws SchadensfallException {
        doVerschiebe(standID);
        schadensfall();
    }

    private void doVerschiebe(int standID) {
        this.berechneBesucherProStunde();
        System.out.println(
                "Stand " + standID + " wurde verschoben und wird jetzt von "
                        + getBesucherProStunde()
                        + " Passanten*innen pro Stunde besucht.");
    }

    private void schadensfall() throws SchadensfallException {
        int artikelIndex = Zufall.zahl(artikel.length);
        if (artikel[artikelIndex] != null) {
            String message = "Dabei ist Artikel " + artikelIndex + ": "
                    + artikel[artikelIndex].toString()
                    + " leider vom Stand gefallen und kaputt gegangen.";
            artikel[artikelIndex] = null;
            throw new SchadensfallException(message);
        }
    }

    @Override
    public String toString() {
        return "Weihnachtsartikelstand:" + System.lineSeparator()
                + super.toString();
    }
}
```

Listing 11: Lebensmittelstand.java

```
public class Lebensmittelstand<T extends Lebensmittel> extends Stand {
    private final T lebensmittel;

    public Lebensmittelstand(T lebensmittel) {
        this.lebensmittel = lebensmittel;
        this.berechneBesucherProStunde();
    }

    @Override
    public double einzelkauf() {
        int menge = SimpleIO.getInt("Wie viel Gramm moechten Sie?");
        System.out.println("Wie viel Gramm moechten Sie?");
        System.out.println(menge);
        System.out.println(
                menge + " Gramm fuer Sie. Lassen Sie es sich schmecken!");
        return menge / 100.0 * lebensmittel.getPreisPro100g();
    }

    @Override
    public String toString() {
        return "Lebensmittelstand fuer " + lebensmittel.getName() + ":"
                + System.lineSeparator()
                + "Preis pro 100g: " + lebensmittel.getPreisPro100g()
                + " Euro"
                + System.lineSeparator() + super.toString();
    }
}
```

Listing 12: Suesswarenstand.java

```
public class Suesswarenstand extends Lebensmittelstand<Suessware>
        implements Verschiebbar {
    public Suesswarenstand() {
        super(new Suessware());
    }

    @Override
    public void verschiebe(int standID) {
        this.berechneBesucherProStunde();
        System.out.println(
                "Stand " + standID + " wurde verschoben und wird jetzt von "
                        + getBesucherProStunde()
                        + " Passanten*innen pro Stunde besucht.");
    }
}
```

Listing 13: SchadensfallException.java

```
public class SchadensfallException extends Exception {
    public SchadensfallException(String message) {
        super(message);
    }
}
```

Listing 14: Verschiebbar.java

```
public interface Verschiebbar {
    void verschiebe(int standID) throws SchadensfallException;
}
```

Listing 15: Weihnachtsmarkt.java

```
public class Weihnachtsmarkt {
    private final Stand[] staende;

    public static void main(String[] args) {
        new Weihnachtsmarkt(5).run();
    }

    public Weihnachtsmarkt(int anzahlStaende) {
        this.staende = new Stand[anzahlStaende];
        for (int i = 0; i < this.staende.length; i++) {
            staende[i] = switch (Zufall.zahl(3)) {
            case 0 -> new Weihnachtsartikelstand();
            case 1 -> new Suesswarenstand();
            default -> new Lebensmittelstand<>(new Flammkuchen());
            };
        }
    }
```

```
    public void run() {
        do {
            praesentiereStaende();
            besucheStand(staende[waehleStand()]);
            verschiebeSchlechtBesuchteStaende();
        } while (!entscheideWeihnachtsmarktVerlassen());
    }

    private void praesentiereStaende() {
        System.out.println(
                "Der Weihnachtsmarkt besteht aus folgenden Staenden:");
        System.out.println();
        for (int i = 0; i < staende.length; i++) {
            System.out.println(i + ": " + staende[i].toString());
        }
    }

    private int waehleStand() {
        int besuchsstand = SimpleIO
                .getInt("Welchen Stand moechten Sie besuchen?");
        System.out.println("Welchen Stand moechten Sie besuchen?");
        System.out.println(besuchsstand);
        return besuchsstand;
    }

    private void besucheStand(Stand stand) {
        System.out.println("Guten Tag!");
        double preis = 0;
        do {
            preis += stand.einzelkauf();
        } while (!entscheideStandVerlassen());
        System.out.println(preis + " Euro, bitte.");
    }

    private boolean entscheideStandVerlassen() {
        boolean result = SimpleIO.getBoolean("Darf es sonst noch etwas sein?");
        System.out.println("Darf es sonst noch etwas sein?");
        System.out.println(result);
        return !result;
    }

    private void verschiebeSchlechtBesuchteStaende() {
        for (int i = 0; i < staende.length; i++) {
            if (staende[i] instanceof Verschiebbar) {
                if (staende[i].getBesucherProStunde() < 30) {
                    try {
                        ((Verschiebbar) staende[i]).verschiebe(i);
                    } catch (SchadensfallException e) {
                        System.out.println(e.getMessage());
                    }
                }
            }
        }
    }

    private boolean entscheideWeihnachtsmarktVerlassen() {
        boolean result = SimpleIO
                .getBoolean("Moechten Sie den Weihnachtsmarkt verlassen?");
        System.out.println("Moechten Sie den Weihnachtsmarkt verlassen?");
        System.out.println(result);
        return result;
    }
}
```

Aufgabe 4 (Collections, Exceptions und Generics): (2 + 4 + 8.5 + 1 + 9 + 6 + 3.5 + 8 + 1 + 3 + 1 + 3 = 50 Punkte)

In dieser Aufgabe geht es ebenfalls um die Implementierung von Datenstrukturen für Mengen.

a) Implementieren Sie eine nicht-abstrakte Klasse `mySets.MySetElement` (d.h., die Klasse mit Namen `MySetElement` soll sich im Paket `mySets` befinden). Diese soll einen Eintrag in einer Menge repräsentieren, die als einfach verkettete Liste implementiert ist. Zu diesem Zweck soll sie einen Typparameter `T` haben, der dem Typ der in der Menge gespeicherten Elemente entspricht. Außerdem soll sie über ein Attribut `next` vom Typ `MySetElement<T>` und ein Attribut `value` vom Typ `T` verfügen. Schreiben Sie

auch einen geeigneten Konstruktor für die Klasse. Die Klasse `MySetElement` soll nur innerhalb des Pakets `mySets` sichtbar sein.

b) Implementieren Sie eine nicht-abstrakte Klasse `mySets.MySetIterator`, die das Interface `Iterator` aus dem Paket `java.util` implementiert. Sie soll einen Typparameter `T` haben, der dem Typ der Elemente entspricht, über die iteriert wird. Außerdem soll sie über ein Attribut `current` vom Typ `MySetElement<T>` verfügen, dessen initialer Wert dem Konstruktor als einziges Argument übergeben wird. Die Methode `hasNext` soll `true` zurückgeben, wenn `current` nicht `null` ist. Die Methode `next` soll das in `current` gespeicherte Objekt vom Typ `T` zurückliefern und `current` auf das nächste `MySetElement` setzen, wenn `current` nicht `null` ist. Es soll eine `java.util.NoSuchElementException` geworfen werden, falls `current == null` gilt. Die Methode `remove` (die in der Java-API als optional gekennzeichnet ist) soll eine `java.lang.UnsupportedOperationException` werfen. Die Methode `forEachRemaining` brauchen Sie nicht zu implementieren. Die Klasse `MySetIterator` soll nur innerhalb des Pakets `mySets` sichtbar sein.

c) Implementieren Sie eine abstrakte Klasse `mySets.MyAbstractSet`, die eine Menge repräsentiert und einen Typparameter `T` hat, der dem Typ der in der Menge gespeicherten Elemente entspricht. Außerdem soll `MyAbstractSet` das Interface `java.lang.Iterable` implementieren. Die Methode `iterator` soll dabei eine geeignete Instanz von `MySetIterator` zurückliefern, die anderen Methoden aus `Iterable` müssen Sie nicht überschreiben. Darüber hinaus soll `MyAbstractSet` alle Methoden des Interfaces `java.util.Set` mit Ausnahme von

- allen Methoden, die bereits in `java.lang.Object` implementiert sind,
- allen Methoden, die eine Default-Implementierung haben,
- allen statischen Methoden und
- allen Methoden, die in der zugehörigen Dokumentation[5] als optional gekennzeichnet sind

auf sinnvolle Art und Weise implementieren. Für die beiden `toArray`-Methoden können Sie konkrete Implementierungen angeben, die nur eine `java.lang.UnsupportedOperationException` werfen.

Die Klasse `MyAbstractSet` soll eine Menge als einfach verkettete Liste von `MySetElements` implementieren. Zu diesem Zweck soll sie über ein Attribut `head` vom Typ `MySetElement<T>` verfügen, dessen initialer Wert dem Konstruktor als einziges Argument übergeben wird. Die Klasse `MyAbstractSet` soll nur innerhalb des Pakets `mySets` sichtbar sein.

d) Erweitern Sie die abstrakte Klasse `mySets.MyAbstractSet` um eine Methode `String toString()`. Die Methode soll eine Menge als String repräsentiert zurückgeben. Hierbei soll z.B. für die Menge $\{1, 2, 3\}$ von Zahlen vom Typ `Integer` der String "$\{1, 2, 3\}$" zurückgeben werden. Die Reihenfolge, in der die Elemente ausgeben werden, spielt keine Rolle und kann vernachlässigt werden.

e) Implementieren Sie eine nicht-abstrakte Klasse `mySets.MyMutableSet`, die von `MyAbstractSet` erbt. Diese soll das Interface `java.util.Set` implementieren. Folglich müssen nun alle optionalen Methoden des Interfaces `java.util.Set` implementiert werden, da diese von `MyAbstractSet` nicht zur Verfügung gestellt werden. Die optionale Methode `retainAll` dürfen Sie mit einer Methode überschreiben, die nur eine `java.lang.UnsupportedOperationException` wirft. Da Instanzen von `MyMutableSet` Mengen repräsentieren, müssen alle Methoden, die Elemente in die Menge einfügen, sicherstellen, dass die zugrunde liegende Liste duplikatfrei bleibt. Dabei werden zwei Objekte `x` und `y` als "gleich" betrachtet, falls `x.equals(y)` den Wert `true` zurückliefert. Falls ein Element in ein `MyMutableSet` eingefügt werden soll, das bereits enthalten ist, soll die Liste unverändert bleiben. Ein Konstruktor der Klasse `MyMutableSet` nimmt keine Argumente entgegen und ruft den (einzigen) Konstruktor der Oberklasse `MyAbstractSet` mit dem Argument `null` auf. Ein zweiter Konstruktor nimmt ein Element entgegen, ruft den Konstruktor der Oberklasse auf und erzeugt eine elementige Menge bestehend aus dem übergebenen Element. Die Klasse `MyMutableSet` soll öffentlich sichtbar sein.

f) Erweitern Sie die Klasse `MyMutableSet` um eine Methode `MyMutableSet<MyMutableSet<T>> powerset()`. Diese Methode soll die Potenzmenge der Menge `this` zurückgeben. Um Mengen vergleichen zu können, empfiehlt es sich, die `equals`-Methode zu überschreiben. Diese benötigen Sie implizit, damit Sie unter

[5] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Set.html`

anderem doppelte Einträge verhindern. Sie können die folgende `equals(Object other)`-Methode verwenden und in `MyAbstractSet` einfügen. Diese Methode überprüft nach einer Typ-Überprüfung, ob alle Elemente von `other` in `this` enthalten sind und ob die Mengen `other` und `this` aus der gleichen Anzahl von Elementen bestehen.

Listing 16: boolean equals(Object other) aus MyAbstractSet.java

```
@Override
public boolean equals(Object other) {
    if (other instanceof MyAbstractSet<?>) {
        return this.containsAll((Collection<?>) other)
            && this.size() == ((MyAbstractSet<?>) other).size();
    }
    return false;
}
```

g) Schreiben Sie unter Beachtung der Prinzipien der Datenkapselung eine Klasse `mySets.MyPair`, die ein Paar implementiert. Die Klasse soll zwei Typparameter `T` und `U` haben, einen Konstruktor haben welcher das Paar initialisiert, die `String toString()`-Methode geeignet überschreiben und für beide Elemente jeweils eine Getter-Methode beinhalten.

h) Schreiben Sie eine Methode `MyMutableSet<MyPair<T,T>> pairs()` in der Klasse `MyMutableSet`. Diese Methode soll eine Menge bestehend aus allen Paaren (t, t') für Elemente t und t' aus `this` zurückgeben. Schreiben Sie eine zweite Methode `MyMutableSet<MyPair<Integer,T>> enumerate()` ebenfalls in der Klasse `MyMutableSet`. Diese Methode zählt die Elemente in `this`, in einer beliebigen Reihenfolge beginnend bei 0, durch und gibt eine Menge von Paaren bestehend aus der eindeutigen Nummer und dem Element selbst zurück. Schreiben Sie eine letzte Methode `MyMutableSet<MyPair<MyMutableSet<T>,Integer>> numberOfSubsets()`. In dieser Methode sollen Sie die Potenzmenge der Menge `this` berechnen und für jedes Element der Potenzmenge bestimmen, wie viele *echte* Teilmengen dieses Elementes in der Potenzmenge enthalten sind. Erzeugen Sie jeweils ein Paar bestehend aus dem Element der Potenzmenge und der Anzahl der echten Teilmengen. Geben Sie schließlich die Menge dieser Paare zurück. Abschließend ergänzen Sie eine Main-Methode in der Klasse `MyMutableSet`. Erstellen Sie die leere Menge vom Typ `MyMutableSet<Integer>`, fügen Sie anschließend die 17 und die 23 hinzu. Geben Sie nun diese Menge aus. Geben Sie außerdem die Potenzmenge, die Paare, die nummerierte Menge und das Ergebnis der Methode `numberOfSubsets` aus.

Hinweise:

- Sie dürfen hier beliebige Methoden aus `java.lang.Math` verwenden.
- Die Ausgabe könnte wie folgt aussehen.
  `set: {23, 17}`
  `powerset: {{17, 23}, {17}, {23}, {}}`
  `pairs: {(17,17), (17,23), (23,17), (23,23)}`
  `enumerate: {(1,17), (0,23)}`
  `numberOfSubsets: {({},0), ({23},1), ({17},1), ({17, 23},3)}`
  Beachten Sie, dass die Reihenfolge der Elemente der jeweiligen Mengen natürlich beliebig sein kann.

i) Entwerfen Sie ein Interface `mySets.MyMinimalSet`. Implementierungen dieses Interfaces sollen unveränderliche Mengen repräsentieren. Das Interface soll die folgende Funktionalität zur Verfügung stellen:

- Es soll einen Typparameter `T` haben, der dem Typ der gespeicherten Elemente entspricht.
- Es soll eine Methode anbieten, um zu überprüfen, ob ein gegebenes Element Teil der Menge ist.
- Es soll eine Methode `void addAllTo(Collection<T> col)` zur Verfügung stellen, die alle Elemente des `MyMinimalSet`s zu der als Argument übergebenen `Collection` hinzufügt.
- Es soll das Interface `java.lang.Iterable` erweitern.

Das Interface `MyMinimalSet` soll öffentlich sichtbar sein.

j) Implementieren Sie eine nicht-abstrakte Klasse `mySets.MyImmutableSet`, die von `MyAbstractSet` erbt und das Interface `MyMinimalSet` implementiert. Diese soll einen Typparameter `T` haben, der dem Typ der gespeicherten Elemente entspricht. Ihr einziger Konstruktor nimmt den initialen Wert für das Attribut `head` der Oberklasse `MyAbstractSet` als Argument entgegen und ruft den (einzigen) Konstruktor von

`MyAbstractSet` entsprechend auf. Die Klasse `MyImmutableSet` soll nur innerhalb des Pakets `mySets` sichtbar sein.

k) Implementieren Sie in der Klasse `MyMutableSet` eine öffentliche Methode `freezeAndClear()` mit dem Rückgabetyp `MyMinimalSet<T>`. Diese soll eine Instanz von `MyImmutableSet` zurückliefern, die die gleiche Menge repräsentiert wie `this`. Außerdem soll sie das Attribut `this.head` auf `null` setzen. Das heißt, die Methode `freezeAndClear` erstellt aus einem `MyMutableSet` ein `MyImmutableSet`, leert die ursprüngliche Menge und liefert das neu erstellte `MyImmutableSet` zurück.

l) Wie Sie bereits in den vorherigen Teilaufgaben gesehen haben, sind alle Methoden, die Elemente zu `Collection`s hinzufügen, als optional gekennzeichnet. Tatsächlich gibt es in der Java-Standardbibliothek einige `Collection`s, die das Hinzufügen von Elementen nicht unterstützen. Diese `Collection`s werfen typischerweise eine `java.lang.UnsupportedOperationException`, wenn eine nicht unterstützte Methode aufgerufen wird. Dies kann dazu führen, dass die Methode `addAllTo` der Klasse `MyImmutableSet` mit einer `UnsupportedOperationException` scheitert. Wir wollen dieses Problem nun explizit machen, indem die Methode `addAllTo` ggf. anstelle einer `UnsupportedOperationException`, welche eine `RuntimeException` ist, eine normale `Exception` wirft, die in der Methodensignatur deklariert werden muss. Implementieren Sie dazu eine nicht-abstrakte Klasse `mySets.UnmodifiableCollectionException` als Unterklasse von `java.lang.Exception`. Ändern Sie die Signatur von `MyMinimalSet.addAllTo` so, dass eine `UnmodifiableCollectionException` geworfen werden darf. Fangen Sie in der Methode `addAllTo` der Klasse `MyImmutableSet` evtl. auftretende `UnsupportedOperationException`s und werfen Sie stattdessen eine `UnmodifiableCollectionException`.

Hinweise:

- Beachten Sie, dass das Paket `mySets` ausschließlich dazu dient, die in der Aufgabenstellung beschriebenen Implementierungen von Mengen zur Verfügung zu stellen. Diese sind eng miteinander verknüpft (z.B. verwenden sowohl `MyMutableSet` als auch `MyImmutableSet` die Klasse `MySetElement`). Daher bietet es sich in dieser Aufgabe an, nicht alle Attribute als `private` zu deklarieren. Dies erhöht die Lesbarkeit des Codes, da direkt auf die entsprechenden Attribute zugegriffen werden kann. Dabei ist es kein Verstoß gegen das Prinzip der Datenkapselung, solange es Klassen außerhalb des Pakets `mySets` nicht möglich ist, auf nicht-private Attribute zuzugreifen.
- `Collection<?>` in der Signatur der Methoden `removeAll` und `retainAll` steht für eine `Collection` beliebiger Elemente, d.h., `removeAll` kann z.B. mit einer `Collection<Integer>`, aber auch mit einer `Collection<String>` als Argument aufgerufen werden.
- `Collection<? extends E>` in der Signatur der Methode `addAll` steht für eine `Collection` von Elementen eines beliebigen Subtyps von `E`. Wenn `F` ein Subtyp von `E` ist, kann `addAll` also sowohl mit einer `Collection<E>` als auch mit einer `Collection<F>` als Argument aufgerufen werden.

Lösung:

Listing 17: MySetElement.java

```
package mySets;

class MySetElement<T> {

    MySetElement<T> next;
    T value;

    public MySetElement(MySetElement<T> next, T value) {
        this.next = next;
        this.value = value;
    }

}
```

Listing 18: MySetIterator.java

```
package mySets;

import java.util.*;
```

```
// no need to implement remove,
// as the default implementation already throws an UnsupportedOperationException
class MySetIterator<T> implements Iterator<T> {

    private MySetElement<T> current;

    public MySetIterator(MySetElement<T> current) {
        this.current = current;
    }

    @Override
    public boolean hasNext() {
        return current != null;
    }

    @Override
    public T next() {
        if (current == null) {
            throw new NoSuchElementException();
        }
        T res = current.value;
        current = current.next;
        return res;
    }


}
```

Listing 19: MyAbstractSet.java

```
package mySets;

import java.util.*;

abstract class MyAbstractSet<T> implements Iterable<T> {

    MySetElement<T> head;

    @Override
    public String toString() {
        String res = "{";
        MySetElement<T> current = head;
        while (current != null) {
            res += current.value;
            if (current.next != null) res += ", ";
            current = current.next;
        }
        return res + "}";
    }

    public MyAbstractSet(MySetElement<T> head) {
        this.head = head;
    }

    public int size() {
        int res = 0;
        for (T t: this) {
            res++;
        }
        return res;
    }

    public boolean isEmpty() {
        return head == null;
    }

    public boolean contains(Object o) {
        for (T t: this) {
            if (t.equals(o)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public Iterator<T> iterator() {
        return new MySetIterator<>(head);
    }

    public Object[] toArray() {
        throw new UnsupportedOperationException();
    }
```

```
    public <T> T[] toArray(T[] a) {
        throw new UnsupportedOperationException();
    }

    public boolean containsAll(Collection<?> c) {
        for (Object o: c) {
            if (!contains(o)) {
                return false;
            }
        }
        return true;
    }

    @Override
    public boolean equals(Object other) {
        if (other instanceof MyAbstractSet<?>) {
            return this.containsAll((Collection<?>) other) && this.size() == ((MyAbstractSet<?>) other).size();
        }
        return false;
    }

}
```

Listing 20: MyMutableSet.java

```
package mySets;

import java.util.*;

public class MyMutableSet<T> extends MyAbstractSet<T> implements Set<T> {

    public MyMutableSet() {
        super(null);
    }

    public MyMutableSet(T e) {
        super(null);
        head = new MySetElement<>(head, e);
    }

    @Override
    public boolean add(T e) {
        if (!contains(e)) {
            head = new MySetElement<>(head, e);
            return true;
        } else {
            return false;
        }
    }

    @Override
    public boolean remove(Object o) {
        if (head == null) {
            return false;
        }
        if (head.value.equals(o)) {
            head = head.next;
            return true;
        }
        MySetElement<T> previous = head;
        MySetElement<T> current = head.next;
        while (current != null) {
            if (current.value.equals(o)) {
                previous.next = current.next;
                return true;
            } else {
                previous = previous.next;
                current = current.next;
            }
        }
        return false;
    }

    @Override
    public boolean addAll(Collection<? extends T> c) {
        boolean changed = false;
        for (T t: c) {
            changed = changed | add(t);
        }
        return changed;
    }
```

```
@Override
public boolean retainAll(Collection<?> c) {
    throw new UnsupportedOperationException();
}

@Override
public boolean removeAll(Collection<?> c) {
    boolean changed = false;
    for (Object o: c) {
        changed = changed | remove(o);
    }
    return changed;
}

@Override
public void clear() {
    head = null;
}

public MyMinimalSet<T> freezeAndClear() {
    MyMinimalSet<T> res = new MyImmutableSet<>(head);
    clear();
    return res;
}

public MyMutableSet<MyMutableSet<T>> powerset() {
    MyMutableSet<T> emptyset = new MyMutableSet<T>();
    MyMutableSet<MyMutableSet<T>> res = new MyMutableSet<MyMutableSet<T>>(emptyset);

    /*
    * Wir fuegen die Elemente der Potenzmenge aufsteigend nach Kardinalitaet zu res hinzu.
    */
    for(int i = 0; i < this.size(); i++) {
        for(MyMutableSet<T> subset : res) {
            if(subset.size() == i) {
                for(T element : this) {
                    MyMutableSet<T> fresh_subset = new MyMutableSet<T>(element);
                    fresh_subset.addAll(subset);
                    res.add(fresh_subset);
                }
            }
        }
    }

    return res;
}

public MyMutableSet<MyPair<T,T>> pairs() {
    MyMutableSet<MyPair<T,T>> res = new MyMutableSet<MyPair<T,T>>();

    for(T e1 : this) {
        for(T e2 : this) {
            res.add(new MyPair<T,T>(e1, e2));
        }
    }

    return res;
}

public MyMutableSet<MyPair<Integer,T>> enumerate() {
    MyMutableSet<MyPair<Integer,T>> res = new MyMutableSet<MyPair<Integer,T>>();
    int i = 0;
    for(T e : this) {
        res.add(new MyPair<Integer,T>(i,e));
        i++;
    }
    return res;
}

public MyMutableSet<MyPair<MyMutableSet<T>,Integer>> numberOfSubsets() {
    MyMutableSet<MyPair<MyMutableSet<T>,Integer>> res = new MyMutableSet<MyPair<MyMutableSet<T>,Integer>>();

    for(MyMutableSet<T> subset : powerset()) {
        res.add(new MyPair<MyMutableSet<T>,Integer>(subset, (int) Math.pow(2, subset.size()) - 1));
    }

    return res;
}

public static void main(String[] args) {
    MyMutableSet<Integer> set = new MyMutableSet<Integer>();
    set.add(17);
    set.add(23);
```

```
        System.out.println(set);
        System.out.println(set.powerset());
        System.out.println(set.pairs());
        System.out.println(set.enumerate());
        System.out.println(set.numberOfSubsets());
    }
}
```

Listing 21: MyPair.java

```
package mySets;

public class MyPair<T,U> {
    private T first;
    private U second;

    public MyPair(T t, U u) {
        first = t;
        second = u;
    }

    public String toString() {
        return "(" + first + "," + second + ")";
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
        return second;
    }
}
```

Listing 22: MyMinimalSet.java

```
package mySets;

import java.util.*;

public interface MyMinimalSet<T> extends Iterable<T> {

    public boolean contains(Object o);
    public void addAllTo(Collection<T> col) throws UnmodifiableCollectionException;

}
```

Listing 23: MyImmutableSet.java

```
package mySets;

import java.util.*;

class MyImmutableSet<T> extends MyAbstractSet<T> implements MyMinimalSet<T> {

    public MyImmutableSet(MySetElement<T> head) {
        super(head);
    }

    @Override
    public void addAllTo(Collection<T> col)
            throws UnmodifiableCollectionException {
        try {
            for (T t: this) {
                col.add(t);
            }
        } catch (UnsupportedOperationException e) {
            throw new UnmodifiableCollectionException();
        }
    }
}
```

Listing 24: UnmodifiableCollectionException.java

```
package mySets;

public class UnmodifiableCollectionException extends Exception {}
```