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

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[1] 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 "$\{1, 2, 3\}$" zurückgegeben 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

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

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 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 1: 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.