Prof. Dr. J. Giesl
S. Dollase, M. Hark, D. Korzeniewski

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter oben links!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Freitag, den 18.01.2019 um 12 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor abgeben.
- In einigen Aufgaben müssen Sie in Java programmieren und `.java`-Dateien anlegen. **Drucken** Sie diese aus **und** schicken Sie sie per **E-Mail** vor Freitag, den 18.01.2019 um 12 Uhr an Ihre Tutorin/Ihren Tutor.
  Stellen Sie sicher, dass Ihr Programm von `javac` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.
- Benutzen Sie in Ihrem Code keine Umlaute, auch nicht in Strings und Kommentaren. Diese führen oft zu Problemen, da diese Zeichen auf verschiedenen Betriebssystemen unterschiedlich behandelt werden. Dadurch kann es dazu führen, dass Ihr Programm bei Ihrer Tutorin/Ihrem Tutor bei Verwendung von Umlauten nicht mehr von `javac` akzeptiert wird.
- Halten Sie sich beim Lösen von Programmieraufgaben an die auf der Website zur Vorlesung verfügbaren Codekonventionen. Verstöße, die zu unleserlichem Code führen, können zu Punktabzug führen.
- Dieses **Bonusblatt** besteht aus einer **freiwilligen Aufgabe**, welche Wissen vermittelt und abfragt, welches in der Vorlesung nicht vorkommt. Somit ist der Schwierigkeitsgrad etwas erhöht.
- Die in diesem Bonusblatt erreichten Punkte zählen für die Zeit nach Weihnachten als Bonuspunkte. Sie tragen nicht zur Summe der erreichbaren Punkte bei, die für die Klausurzulassung relevant ist, jedoch werden Ihnen die erreichten Punkte ganz normal gutgeschrieben.
- Die Lösung für dieses Bonusblatt soll zusammen mit der Lösung für Übungsblatt 9 abgegeben werden und wird in der Globalübung zu Übungsblatt 9 vorgestellt.

Aufgabe 1 (Funktionen höherer Ordnung in Java): (2 + 2 + 2 + 6 = 12 Punkte)

Inzwischen haben wir Funktionen höherer Ordnung in Haskell kennengelernt. Seit der Version 8 werden diese auch in Java, in Form von Lambda-Ausdrücken, direkt unterstützt. In dieser Bonusaufgabe soll es darum gehen, einen Eindruck davon zu bekommen wie Lambda-Ausdrücke in Java funktionieren.
In Haskell sind Funktionen direkt vom Typsystem unterstützt. Beispielsweise definiert folgender Code die Funktionen `add` und `multiply` des Typs `Int -> Int -> Int`, welche ihre beiden Parameter addieren bzw. multiplizieren.

```
add :: Int -> Int -> Int
add a b = a + b

multiply :: Int -> Int -> Int
multiply a b = a * b
```

Das Besondere an Funktionen höherer Ordnung ist nun, dass wir Funktionen als Werte betrachten können, welche in Variablen gespeichert werden können und die von Funktionen sowohl als Parameter entgegen genommen werden können und von ihnen zurückgegeben werden können. So können wir etwa eine Funktion `calculate` definieren, welche eine Funktion des Typs `Int -> Int -> Int` als Parameter bekommt und diese auf zwei konstante `Int`-Werte anwendet:

```
calculate :: (Int -> Int -> Int) -> Int
calculate f = f 14 28
```

Der Ausdruck `calculate add` wird also zu 42 ausgewertet, wohingegen der Ausdruck `calculate multiply` zu 392 ausgewertet wird.
In Haskell kennen wir außerdem die Lambda-Schreibweise für Funktionen. So ließe sich obige Berechnung auch durchführen, ohne zuvor die Funktionen `add` und `multiply` zu definieren. Der Ausdruck `calculate add` bzw. `calculate multiply` wird mit einem Lambda-Ausdruck als `calculate (\a b -> a + b)` bzw. `calculate (\a b -> a * b)` dargestellt.
Nun wollen wir diese Konzepte in die Java-Welt übertragen. Wir beginnen damit, die Funktionen `add` und `multiply` zu definieren.

```
public class Arithmetic {
    public static int add(int a, int b) {
        return a + b;
    }

    public static int multiply(int a, int b) {
        return a * b;
    }
}
```

Die Funktionen `add` und `multiply` können hier nicht direkt einer Variable zugewiesen werden. Dies liegt daran, dass Java Funktionen nicht direkt als Werte unterstützt. Um dies zu ermöglichen, bedienen wir uns eines Tricks: Eine Variable kann ein Objekt referenzieren und an einem Objekt können Funktionen aufgerufen werden. Die Idee ist also, Funktionen höherer Ordnung als Objekte eines speziell definierten Funktionentyps darzustellen.
Beginnen wir mit der Deklaration des Funtionentyps. Unser Ziel ist es, eine Variable deklarieren zu können, deren Typ die Funktionen sind, welche zwei `int`-Parameter erhalten und einen `int`-Wert zurückgeben. Für verschiedene Werte dieser Variable soll die Funktion verschieden implementiert werden können. Daher deklarieren wir keine konkrete Klasse, sondern ein Interface.

```
public interface IntFunction {
    int apply(int a, int b);
}
```

Nun implementieren wir die beiden Funktionen `add` und `multiply` erneut. Die neue Implementierung wird es ermöglichen, dass wir sie einer Variable vom Typ `IntFunction` zuweisen.

```
public class Adder implements IntFunction {
    @Override
    public int apply(int a, int b) {
        return Arithmetic.add(a, b);
    }
}

public class Multiplier implements IntFunction {
    @Override
    public int apply(int a, int b) {
        return Arithmetic.multiply(a, b);
    }
}
```

Um nun einer Variable `f` des Typs `IntFunction` die Funktion `add` bzw. `multiply` zuzuweisen, können wir Folgendes schreiben: `IntFunction f = new Adder()` bzw. `IntFunction f = new Multiplier()`.
Nun, da wir eine Funktion als Wert in einer Variable speichern können, können wir auch in Java die Funktion `calculate` deklarieren.

```
public static int calculate(IntFunction f) {
    return f.apply(14, 28);
}
```

Der Aufruf `calculate(new Adder())` würde also den Wert `42` zurückgeben, der Aufruf `calculate(new Multiplier())` hingegen den Wert `392`.
Bis hierhin haben wir kein Java-Feature genutzt, welches speziell auf Funktionen höherer Ordnung ausgerichtet ist. Dies ändert sich nun. In Java 8 wurde eine Notation für Lambda-Ausdrücke eingeführt, welche die doch recht wortreiche Deklaration der beiden Klassen `Adder` und `Multiplier` überflüssig macht. Anstatt also `IntFunction f = new Adder()` zu schreiben, können wir direkt `IntFunction f = (a, b) -> Arithmetic.add(a, b)` schreiben. Hier ist `(a, b) -> Arithmetic.add(a, b)` ein Lambda-Ausdruck, welcher zwei Parameter `a` und `b` erhält und welcher bei Ausführung den Funktionsaufruf `Arithmetic.add(a, b)` durchführt und das Ergebnis zurückgibt. Der Java-Compiler inferiert nun die Typen der Parameter und des Rückgabewertes des Lambda-Ausdrucks.
Der komplette Ausdruck `(a, b) -> Arithmetic.add(a, b)` hat jedoch selbst keinen festen Typ, da Java keine Funktionstypen unterstützt. Um trotzdem einen Typ zuordnen zu können, betrachtet der Java-Compiler den sogenannten Target-Type, also den Typ der Variable, welcher der Lambda-Ausdruck zugewiesen wird, in unserem Fall den Typ `IntFunction` der Variable `f`. Der Java-Compiler erkennt, dass `IntFunction` ein Interface mit nur einer nicht implementierten Methode ist. Er überprüft, ob die Signatur dieser unimplementierten Methode mit der Signatur des Lambda-Ausdrucks übereinstimmt. Ist dies der Fall, so ist das Programm typkorrekt und der Lambda-Ausdruck dient zum Einen dazu, implizit eine Unterklasse von `IntFunction` zu deklarieren (wie `Adder` und `Multiplier`) und zum Anderen dazu, an der Stelle des Lambda-Ausdrucks ein Objekt dieser implizit deklarierten Klasse zu erstellen.
Nun, da wir wissen wie Lambda-Ausdrücke in Java umgesetzt sind, wollen wir die Syntax noch einmal etwas genauer ansehen. Die Schreibweise `IntFunction f = (a, b) -> Arithmetic.add(a, b)` ist bereits sehr viel angenehmer als die explizite Implementierung der Klasse `Adder`. Trotzdem enthält sie weiterhin redundante Informationen. So haben wir explizit aufgeschrieben, dass der Lambda-Ausdruck zwei Parameter erhält und diese in derselben Reihenfolge an eine andere Funktion übergibt. Dieser Spezialfall lässt sich weiter verkürzen zu `IntFunction f = Arithmetic::add`, was schlicht die Methode `add` als Lambda-Ausdruck beschreibt. Da die Methoden `add` bzw. `multiply` nicht sonderlich kompliziert sind, können diese außerdem direkt als Lambda-Ausdrücke formuliert werden, ohne auf die Klasse `Arithmetic` zurückzugreifen: `IntFunction f = (a, b) -> a + b` bzw. `IntFunction f = (a, b) -> a * b`.
Hier also einige typkorrekte Aufrufe der `calculate`-Methode.

```
calculate(Arithmetic::add);
calculate((a, b) -> Arithmetic.add(a, b));
calculate((a, b) -> a + b);

calculate(Arithmetic::multiply);
calculate((a, b) -> Arithmetic.multiply(a, b));
calculate((a, b) -> a * b);
```

Nun sind wir aus der Programmierersicht bereits sehr nah an der Haskell Schreibweise. Ein Unterschied der bisher bleibt, ist, dass wir in Java weiterhin keine Funktionstypen haben. Während wir in Haskell einfach `Int -> Int -> Int` als Funktionstyp notieren können, so müssen wir in Java zunächst ein sogenanntes *Functional Interface* `IntFunction` mit nur einer abstrakten Methode deklarieren, um dieses als Funktionstyp nutzen zu können. Dies kann schnell dazu führen, dass eine schwer zu überblickende Menge von Functional Interfaces existiert, da jede Programmbibliothek ihre eigenen, zueinander inkompatiblen Functional Interfaces mitbringt. Doch auch hierfür gibt es Abhilfe, denn auch ein generisches Interface kann als Functional Interface dienen. Tatsächlich bietet Java im Package `java.util.function` einige häufig benötigte Functional Interfaces. Hier ein paar Beispiele:

- `Consumer<T>` mit der Funktion `void accept(T t)`, welche einen Parameter vom Typ `T` erhält und nichts zurückgibt.
- `Supplier<T>` mit der Funktion `T get()`, welche keinen Parameter erhält und einen Wert vom Typ `T` zurückgibt.
- `Function<T, R>` mit der Funktion `R apply(T t)`, welche einen Parameter vom Typ `T` erhält und einen Wert vom Typ `R` zurückgibt.
- `BiFunction<T, U, R>` mit der Funktion `R apply(T t, U u)`, welche zwei Parameter vom Typ `T` und `U` erhält und einen Wert vom Typ `R` zurückgibt.

Unser selbst definierter Typ `IntFunction` könnte also durch den Typ `BiFunction<Integer, Integer, Integer>` ersetzt werden. Dies würde uns jedoch dazu zwingen, Autoboxing für unsere `int`-Werte zu nutzen, da primitive Typen wie `int` nicht als generischer Typparameter genutzt werden können. Um dies zu vermeiden, hält Java außerdem einige spezialisierte Versionen der oben genannten Functional Interfaces vor. Hier einige Beispiele:

- `Predicate<T>` mit der Funktion `boolean test(T t)`, welche einen Parameter vom Typ `T` erhält und einen `boolean`-Wert zurückgibt.
- `ToIntBiFunction<T, U>` mit der Funktion `int applyAsInt(T t, U u)`, welche zwei Parameter vom Typ `T` und `U` erhält und einen `int`-Wert zurückgibt.
- `IntBinaryOperator` mit der Funktion `int applyAsInt(int left, int right)`, welche zwei Parameter vom Typ `int` erhält und einen `int`-Wert zurückgibt.

Wir sollten unser selbst definiertes Functional Interface `IntFunction` also nicht durch `BiFunction<Integer, Integer, Integer>` sondern durch `IntBinaryOperator` ersetzen. Unsere oben definierte `calculate`-Methode könnte nun also wie folgt definiert werden:

```
public static int calculate(IntBinaryOperator f) {
    return f.applyAsInt(14, 28);
}
```

Somit wird auch die eigene Deklaration unseres Functional Interface `IntFunction` überflüssig.
Nun wollen wir, ähnlich wie auf Übungsblatt 9, die Klasse `Optional<T>` implementieren, jedoch dieses Mal in Java. Java besitzt selbst bereits einen Typ `java.util.Optional`, jedoch wollen wir in dieser Aufgabe erneut eine alternative Version davon implementieren. Den folgenden Code können Sie von unserer Website herunterladen.

```
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Supplier;

public class Optional<T> {
    public static <T> Optional<T> empty() {
        return new Optional<>(null);
    }

    public static <T> Optional<T> present(T value) {
        if (value != null) {
            return new Optional<>(value);
        } else {
            throw new IllegalArgumentException("parameter cannot be null");
        }
    }

    private final T value;

    public Optional(T value) {
        this.value = value;
    }

    public boolean isPresent() {
        return value != null;
    }

    public T get() {
        if (value != null) {
            return value;
        } else {
```

```
            throw new IllegalStateException("cannot get empty value");
        }
    }

    public <R> Optional<R> map(Function<T, R> mapper) {
        // ...
    }

    public Optional<T> filter(Predicate<T> tester) {
        // ...
    }

    public <R> R fold(Function<T, R> presentMapper, Supplier<R> emptyReplacer) {
        // ...
    }
}
```

Wir geben zwei Methoden vor, mit denen ein `Optional`-Objekt erstellt werden kann. Der Aufruf `Optional.empty()` gibt ein leeres `Optional`-Objekt zurück. Der Aufruf `Optional.present("hallo")` gibt ein nicht-leeres `Optional`-Objekt zurück, das den `String`-Wert `"hallo"` enthält. Der Aufruf `Optional.present(null)` wirft einen Fehler. Hier einige Beispielaufrufe mit ihrem entsprechenden Ergebnis.

```
Optional.present("hallo").isPresent()
// -> true

Optional.empty().isPresent()
// -> false

Optional.present("hallo").get()
// -> "hallo"

Optional.empty().get()
// wirft einen Fehler
```

a) Implementieren Sie in der Klasse `Optional<T>` eine nicht-statische Methode

```
public <R> Optional<R> map(Function<T, R> mapper) {...}
```

welche die Funktion `mapper` auf den im aktuellen nicht-leeren `Optional`-Objekt enthaltenen Wert anwendet und das Ergebnis wieder in einem `Optional`-Objekt gewrappt zurückgibt. Ist das aktuelle `Optional`-Objekt leer, so soll hingegen ein leeres `Optional`-Objekt zurückgeben werden. Hier einige Beispielaufrufe mit ihrem entsprechenden Ergebnis.

```
Optional.present("hallo").map(v -> v + " welt")
// -> Optional.present("hallo welt")

Optional.empty().map(v -> v + " welt")
// -> Optional.empty()
```

b) Implementieren Sie in der Klasse `Optional<T>` eine nicht-statische Methode

```
public Optional<T> filter(Predicate<T> tester) {...}
```

welche den im aktuellen nicht-leeren `Optional`-Objekt enthaltenen Wert mit der übergebenen Funktion `tester` überprüft. Wenn die Prüfung fehlgeschlagen ist, soll ein leeres `Optional`-Objekt zurückgegeben werden und ansonsten soll das aktuelle `Optional`-Objekt zurückgegeben werden. Wenn das aktuelle `Optional`-Objekt leer ist, soll auch das leere `Optional`-Objekt zurückgegeben werden. Hier einige Beispielaufrufe mit ihrem entsprechenden Ergebnis.

```
Optional.present(10).filter(v -> v > 5)
// -> Optional.present(10)

Optional.present(2).filter(v -> v > 5)
// -> Optional.empty()

Optional.empty().filter(v -> v > 5)
// -> Optional.empty()
```

c) Implementieren Sie in der Klasse `Optional<T>` die folgende nicht-statische Methode:

```
public <R> R fold(Function<T, R> presentMapper,
                  Supplier<R> emptyReplacer) {...}
```

Falls das aktuelle `Optional`-Objekt nicht leer ist, soll der darin enthaltene Wert mit der übergebenen Funktion `presentMapper` abgebildet und das Ergebnis zurückgegeben werden. Ein leeres `Optional`-Objekt soll hingegen durch den von der übergebenen Funktion `emptyReplacer` zurückgegebenen Wert ersetzt werden. Hier einige Beispielaufrufe mit ihrem entsprechenden Ergebnis.

```
Optional.present(10).fold(v -> v * 2, () -> 42)
// -> 20

Optional.empty().fold(v -> v * 2, () -> 42)
// -> 42
```

Hinweise:

- Der zweite Parameter könnte hier auch eine Konstante des Typs `R` sein. Falls die Berechnung dieses Werts jedoch eine resourcenintensive Operation ist, so würde eben diese Operation immer ausgeführt werden, selbst wenn das Ergebnis niemals genutzt wird, weil `fold` nur für nicht-leere `Optional`-Objekte aufgerufen wird. Somit nutzen wir hier einen `Supplier<R>`, um den Wert nur dann zu berechnen, falls er tatsächlich benötigt wird. Dies ist vergleichbar mit Haskells Lazy Evaluation.

d) Gegeben sei die folgende Klasse `Article`:

```
public class Article {
    private final String name;
    private final int price;

    public Article(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public int getPrice() {
        return price;
    }

    public boolean isHumanEatable() {
        return !name.equals("Dog Food");
    }

    public Article adjustPrice() {
        if (price < 1000) {
            return new Article(name, price * 2);
```

```
        } else {
            return this;
        }
    }

    public String stringify() {
        return "The Article named '" + name + "' costs " + price + " cent.";
    }
}
```

Gegeben sei außerdem die folgende Klasse `Store`, welche ein Geschäft repräsentieren soll.

```
public class Store {
    public static void main(String[] args) {
        System.out.println(toPriceTag(new Article("Dog Food", 1000)));
        System.out.println(toPriceTag(new Article("Pizza", 1000)));
        System.out.println(toPriceTag(new Article("Pizza", 100)));
    }

    private static String toPriceTag(Article article) {
        // ...
    }
}
```

Implementieren Sie die Methode `toPriceTag`, welche den Text für das Preisschild eines Artikels generiert. Dazu nutzt sie die Funktionen `map`, `filter` und `fold` aus der Klasse `Optional` sowie `isHumanEatable`, `adjustPrice` und `stringify` aus der Klasse `Article`, um folgende Funktionalität zu implementieren.

Das Geschäft verkauft ausschließlich für Menschen genießbare Artikel. Hundefutter kann also nicht verkauft werden. Ein Artikel mit dem Namen `"Dog Food"` erhält also das Preisschild `"This Article is unavailable."` Für alle anderen Artikel wird der Preis verdoppelt, falls er kleiner als 10 Euro ist, und anschließend der Artikel mit der `stringify`-Funktion in einen `String` umgewandelt.

Der erste Schritt ist, den übergebenen Artikel in ein `Optional`-Objekt umzuwandeln. Nutzen Sie dazu die Funktion `Optional.present`.

Nun wollen wir, dass für Menschen nicht genießbare Ware unverkäuflich ist. Falls der übergebene Artikel also nicht für Menschen genießbar ist, so soll das `Optional`-Objekt durch das leere `Optional`-Objekt ersetzt werden. Wenden Sie dazu auf das aktuelle Zwischenergebnis die Funktion `filter` an und übergeben Sie dieser Funktion einen Lambda-Ausdruck, welcher `isHumanEatable` ausführt.

Der nächste Schritt ist, den Preis des ggf. leeren Artikels zu verdoppeln, falls dieser kleiner als 10 Euro ist. Wenden Sie hierzu auf das aktuelle Zwischenergebnis die Funktion `map` an und übergeben Sie dieser Funktion einen Lambda-Ausdruck, welcher `adjustPrice` ausführt.

Abschließend wollen wir den ggf. leeren Artikel zu einem `String` für das Preisschild umwandeln. Wenden Sie hierzu auf das aktuelle Zwischenergebnis die Funktion `fold` an und übergeben Sie dieser Funktion einen Lambda-Ausdruck, welcher auf einen nicht-leeren Artikel `stringify` anwendet und einen weiteren Lambda-Ausdruck, welcher das leere `Optional`-Objekt durch den Text `"This Article is unavailable."` ersetzt.

Die `main`-Funktion sollte nun folgendes ausgeben:

```
This Article is unavailable.
The Article named 'Pizza' costs 1000 cent.
The Article named 'Pizza' costs 200 cent.
```

Hinweise:

- Auch für nicht-statische Methoden kann die `::`-Syntax für Lambda-Ausdrücke genutzt werden. Dabei wird die Signatur der Methode vorne um einen Parameter erweitert. So ist die folgende Zuweisung typkorrekt: `Predicate<Article> tester = Article::isHumanEatable`.

Die nicht-statische Methode `isHumanEatable` wird also so behandelt, als ob sie einen Parameter vom Typ `Article` hätte.