Universität Augsburg
Institut für Informatik

Übung zur Vorlesung Informatik 2 SoSe 2024
Fakultät für Angewandte Informatik
Institut für Informatik
Prof R. Lorenz, L. Petrak, J. Linne, V. Le Claire, P. Schalk 31.05.2024

Lösungsvorschlag zu Übungsblatt 5

Abgabe: 10.06.2024, 10:00 Uhr (im Digicampus via VIPS: .java-Dateien für Code, .uxf für UML, .pdf für alles andere)

- Dieses Übungsblatt muss im Team abgegeben werden (Einzelabgaben sind nicht erlaubt!).
- Die **Zeitangaben** geben zur Orientierung an, wie viel Zeit für eine Aufgabe später in der Klausur vorgesehen wäre; gehen Sie davon aus, dass Sie zum jetzigen Zeitpunkt wesentlich länger brauchen und die angegebene Zeit erst nach ausreichender Übung erreichen.

* leichte Aufgabe / ** mittelschwere Aufgabe / *** schwere Aufgabe

Aufgabe 17 ** *(Polymorphismus / Schnittstellen, 20 Minuten)*

a) *(10 Minuten)*
Erklären Sie die Begriffe **Substitutionsprinzip**, **spätes Binden** und **Polymorphismus** anhand des Eingabeparameters und der Implementierung der `contains`-Methode der Klasse `ArrayList`.

Der Eingabeparameter `o` der `contains`-Methode ist vom Typ `Object`. Für diesen können Objekte beliebiger Klassen eingesetzt werden, denn jede Klasse ist Unterklasse von `Object` (Substitutionsprinzip). Es stehen im Rumpf der `contains`-Methode nur `Object`-Methoden für `o` zur Verfügung, da bei Übersetzung des Programms noch nicht bekannt sein muss, welche Objekte für `o` zur Laufzeit eingesetzt werden. Zum Beispiel wird in `contains` für `o` die `equals`-Methode benutzt, die für alle Objekte zur Verfügung steht. Falls zur Laufzeit ein Objekt für `o` eingesetzt wird, dessen Klasse die `equals`-Methode überschreibt, so wird die Implementierung der Klasse und nicht die Implementierung von `Object` benutzt (spätes Binden). Insgesamt spricht man bei diesem Prinzip von Polymorphismus, da hier die Implementierung der `equals`-Methode vielgestaltig ist und verwendet wird.

b) *(10 Minuten)*
Erklären Sie die Vorteile der Benutzung von Schnittstellen als Datentypen anhand des Eingabeparameters der `removeAll`-Methode der Klasse `ArrayList`.

Der Eingabeparameter der `removeAll`-Methode ist vom Typ `Collection` - für eingesetzte Objekte stehen damit nur die `Collection`-Methoden zur Verfügung, auch wenn der Typ eines eingesetzten Objekts noch wesentlich mehr Methoden hat. Ein solcher Typ können alle Klassen sein, die `Collection` implementieren, z.B. `ArrayList`, aber auch `TreeSet`. Von diesen anderen Methoden wird abstrahiert, denn sie werden nicht benötigt, um die Objekte der übergebenen Collection aus der Liste zu entfernen. So lassen sich für den Eingabeparameter viele unterschiedliche Objekte einsetzen - die Methode wird allgemeiner benutzbar. Die Implementierung der Methode ist nur abhängig von `Collection`, aber nicht von anderen Eigenschaften der eingesetzten Objekte und nicht davon, wie die Objekte die `Collection`-Methoden implementieren. Dadurch sinkt die Gefahr von Nebeneffekten zwischen Eigenschaften der eingesetzten Objekte und der Funktionsweise der Methode z.B. bei Änderungen am Code der Objekte). Ein weiterer Vorteil von der Verwendung von Schnittstellen ist, dass eine Klasse beliebig viele Schnittstellen implementieren kann, während eine Klasse immer nur von einer Oberklasse erben kann. So gibt es mehr Möglichkeiten, Methoden von verschiedenen Arten von Klassen verwendbar zu machen.

Aufgabe 18 ** *(Eigene Klasse und Containerklasse, 30 Minuten)*

In dieser Aufgabe sollen Sie eine eigene Datenklasse für die Verwaltung von Gepäckstücken implementieren. Dabei üben Sie auch das Werfen von Ausnahmen.

a) *(**, 24 Minuten)*
Implementieren Sie eine eigene Klasse `Luggage` für Gepäckstücke mit passenden set-, get- und check-Methoden:

- Ein Gepäckstück hat eine Flug-ID, einen Besitzer und ein Gewicht.
- Der Besitzer wird durch eine nicht-leere Zeichenkette dargestellt.
- Das Gewicht ist eine positive reelle Zahl.
- Die Flug-ID ist von der Form `"XXYY"`, wobei `X` für einen beliebigen Großbuchstaben und `Y` für eine beliebige Ziffer steht. Benutzen Sie einen geeigneten regulären Ausdruck[1], um diese Einschränkung darzustellen.
- Ist eine der Datenstruktur-Invarianten verletzt, soll die zugehörige set-Methode eine `IllegalArgumentException` werfen, in welcher der aufgetretene Fehler **kurz und knapp** beschrieben ist.
- Zwei `Luggage`-Objekte gelten als gleich, wenn ihre Flug-ID und ihr Besitzer übereinstimmen.
- Der Hashcode eines `Luggage`-Objekts soll aus seiner Flug-ID und dem Namen seines Besitzers berechnet werden. Überschreiben Sie dafür die `hashCode`-Methode und verwenden Sie dafür eine geeignete Klassenmethode der Klasse `Objects` im Paket `java.util`.

[1] Siehe https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/regex/Pattern.html. Für diese Aufgabe sind die Zusammenfassungen zu **Character classes** und **Greedy Quantifiers** vollkommen ausreichend.

Lösung:

```java
import java.util.Objects;

public class Luggage {
    private String flightID;
    private String owner;
    private double weight;

    public Luggage(String flightID, String owner, double weight) {
        setFlightID(flightID);
        setOwner(owner);
        setWeight(weight);
    }

    private static boolean checkFlightID(String flightID) {
        return flightID.matches("[A-Z]{2}\\d{2}");
    }

    private static boolean checkOwner(String owner) {
        return !owner.isEmpty();
    }

    private static boolean checkWeight(double weight) {
        return weight > 0;
    }
    public String getFlightID() {
        return flightID;
    }

    public void setFlightID(String flightID) {
        if (!checkFlightID(flightID)) {
            throw new IllegalArgumentException("Flightcode was " + flightID + "but needs to
            ↪  be of the form XXYY where X is any uppercase latin letter and y is any
            ↪  digit");
        }
        this.flightID = flightID;
    }

    public String getOwner() {
        return owner;
    }

    public void setOwner(String owner) {
        if (!checkOwner(owner)) {
            throw new IllegalArgumentException("Owner cannot be empty");
        }
        this.owner = owner;
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        if (!checkWeight(weight)) {
            throw new IllegalArgumentException("Weight must be positive");
        }
        this.weight = weight;
    }

    @Override
    public boolean equals(Object o) {
        if (o == null) return false;
        if (getClass() != o.getClass())
            return false;
        Luggage l = (Luggage) o;
        return l.getFlightID().equals(this.getFlightID()) &&
        ↪  l.getOwner().equals(this.getOwner());
    }
```

```
    @Override
    public int hashCode() {
        return Objects.hash(getFlightID(), getOwner());
    }
}
```

b) *(**, 8 Minuten)*
Implementieren Sie eine Klasse `LuggageContainer` zum Verwalten aller Gepäckstücke: Der Container bietet...

- mit `void addLuggage(Luggage l)` eine Möglichkeit an, ein `Luggage`-Objekt hinzuzufügen.
- mit `void removeLuggage(Luggage l)` eine Möglichkeit an, ein `Luggage`-Objekt zu entfernen.
- eine Mögichkeit an, über alle enthaltenen Elemente zu iterieren und implementiert dafür eine geeignete Schnittstelle.
- mit `Stream<Luggage> stream()` die Rückgabe eines Streams aller enthaltenen `Luggage`-Objekte an.

Lösung:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.stream.Stream;

public class LuggageContainer implements Iterable<Luggage> {
    private final ArrayList<Luggage> luggageList;

    public LuggageContainer() {
        luggageList = new ArrayList<>();
    }

    public void addLuggage(Luggage luggage) {
        luggageList.add(luggage);
    }

    public void removeLuggage(Luggage luggage) {
        luggageList.remove(luggage);
    }

    @Override
    public Iterator<Luggage> iterator() {
        return luggageList.iterator();
    }

    public Stream<Luggage> stream() {
        return luggageList.stream();
    }
}
```

c)
Testen Sie Ihre Implementierungen mit der im DigiCampus zur Verfügung gestellten Programmklasse `LuggageMain.java`.

Aufgabe 19 ** *(Exceptions abfangen und werfen, 25 Minuten)*

In den folgenden Teilaufgaben sollen Sie lernen, wie Sie Exceptions abfangen und werfen können.

a) *(**, 10 Minuten)*
Laden Sie die im Digicampus gegebene Klasse `FaultyCode` herunter. Sorgen Sie durch maximal spezifisches Abfangen aller auftretenden Exceptions dafür, dass das Programm fehlerfrei terminiert. Führen Sie dazu das Programm aus und verwenden Sie die ausgegebenen Stack Traces, um nach und nach alle Anweisungen, die Exceptions werfen, mit einem try-catch Block umgeben und mit geeigneten Methoden die Klasse und die enthaltene Fehlermeldung aller geworfenen Ausnahmen auf der Kommandozeile ausgeben. Umgeben Sie jeden Aufruf, der eine Exception wirft, mit einem eigenen try-catch Block.
Lassen Sie die Anweisungen des Programms sonst unverändert!

Lösung:

```
import java.util.ArrayList;
import java.util.InputMismatchException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class FaultyCodeSolution {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Test");
        try {
            System.out.println(list.get(-1));
        } catch (IndexOutOfBoundsException e) {
            System.out.println(e.getClass() + ": " + e.getMessage());
        }
        try {
            System.out.println(Integer.parseInt(list.get(0)));
        } catch (NumberFormatException e) {
            System.out.println(e.getClass() + ": " + e.getMessage());
        }
        String s1 = "abc";
        Scanner myScanner = new Scanner(s1);
        try {
            int x = myScanner.nextInt();
        } catch (InputMismatchException e) {
            System.out.println(e.getClass() + ": " + e.getMessage());
        }
        String s2 = null;
        String s3 = null;
        s2 = myScanner.next();
        try {
            s3 = myScanner.next();
        } catch (NoSuchElementException e) {
            System.out.println(e.getClass() + ": " + e.getMessage());
        }
        try {
            System.out.println(s2.toLowerCase() + " " + s3.toUpperCase());
        } catch (NullPointerException e) {
            System.out.println(e.getClass() + ": " + e.getMessage());
        }

    }
}
```

b) *(***, 15 Minuten)*
Schreiben Sie eine Programmklasse, die einen primitiven Taschenrechner implementiert. Der Taschenrechner soll in der Lage sein, zwei reellwertige Zahlen zu addieren, subtrahieren, multiplizieren und zu dividieren. Die Ein- und Ausgabe soll über die Kommandozeile erfolgen. Pro Zeile soll eine Operation eingegeben werden, die Verwendung von Klammern ist nicht erlaubt. Das Programm soll

- zu Beginn kurz über seine Benutzung informieren,
- solange zeilenweise Rechnungen einlesen, bis die Zeichenkette `"EXIT"` eingegeben wird,
- die Eingabe in zwei Zahlen und den verwendeten Operator aufspalten und die Rechnung mit ihrem Ergebnis ausgeben.

Beachten Sie außerdem folgende Vorgaben zur Implementierung:

- Implementieren Sie für die Berechnung des Ergebnisses eine eigene Klassenmethode, die auch das Ausgeben des Ergebnisses übernimmt.
- Verwenden Sie für das Aufteilen der Eingabe den regulären Ausdruck `"[\\x2B\\x2D\\x2A\\x2F]"`[2]
  zusammen mit einer geeigneten Objektmethode der Klasse `String`.
- Bei der Nutzereingabe können verschiedene Fehler auftreten. Behandeln Sie die folgenden Fehler jeweils mit dem Werfen der angegebenen Ausnahme:
  - Es wird kein gültiger Operator übergeben: `IllegalArgumentException`
  - Es werden zu viele gültige Operatoren übergeben `IllegalArgumentException`
  - Es werden keine gültigen Zahlen als Operanden übergeben: `NumberFormatException`

  Wenn möglich, reichen Sie die von API-Methoden geworfenen Ausnahmen weiter.
- Behandeln Sie die auftretenden Ausnahmen in der `main`-Methode maximal spezifisch und geben Sie eine jeweils eine passende Fehlermeldung auf dem Standard-Fehlerstrom aus.

Lösung:

```java
import java.util.Scanner;

public class Calculator {
    public static void main(String[] args) {
        Scanner myScanner = new Scanner(System.in);
        System.out.println("Please enter your calculation in the form of x op y, where x and
        ↪  y are real nubmers and op is either +, -, * or / .");
        System.out.println("Enter \"EXIT\" to stop at any time!");
        String input = myScanner.nextLine();
        while (!input.equals("EXIT")) {
            try {
                calculateOutput(input);
            } catch (NumberFormatException e) {
                System.err.println("Failed to parse your number input.");
            }
            catch (IllegalArgumentException e) {
                System.err.println(e.getMessage());
            }
            input = myScanner.nextLine();
        }
        myScanner.close();
    }

    public static void calculateOutput(String input) {
        String[] values = input.split("[\\x2B\\x2D\\x2A\\x2F]");
        if (values.length == 1) {
            throw new IllegalArgumentException("No valid operator was used");
        } else if (values.length > 2) {
            throw new IllegalArgumentException("Only one operator allowed, but " +
            ↪  (values.length - 1) + " were used.");
        }
```

[2] \x2B, 2D, 2A, 2F sind die Hexadezimalwerte der Zeichen `'+'`, `'-'`, `'*'`, `'/'`.

```
            double x = Double.parseDouble(values[0]);
            double y = Double.parseDouble(values[1]);
            char c = input.charAt(values[0].length());
            if (c == '+') {
                System.out.println(input + " = " + (x + y));
            } else if (c == '-') {
                System.out.println(input + " = " + (x - y));
            } else if (c == '*') {
                System.out.println(input + " = " + (x * y));
            } else if (c == '/') {
                System.out.println(input + " = " + (x / y));
            }
        }
    }
```

Aufgabe 20 ** *(Eigene Datenklassen implementieren, 30 Minuten)*

In dieser Aufgabe sollen Sie nach einer Systembeschreibung mehrere eigene Datenklassen implementieren und diese mit einer eigenen Programmklasse testen.

a) *(**, Klassen `RealEstate`, `Parcel` und `Apartment` nach Systembeschreibung implementieren, 22 Minuten)*
Implementieren Sie eigene Datenklassen `RealEstate`, `Parcel` und `Apartment` für eine Anwendung zum Anbieten von **Immobilien** (real estate), d.h. Grundstücke (parcel) und Apartments, gemäß folgender Systembeschreibung.

Es sollen Grundstücke und Apartments angeboten werden können. Für Grundstücke ist eine Größe anzugeben. Es dürfen nur Grundstücke, die nicht weniger als 250 *Quadratmeter haben, angeboten werden. Für Apartments ist die Wohnfläche anzugeben – diese darf nicht weniger als* 20 *und nicht mehr als* 120 *Quadratmeter betragen. Außerdem haben alle Immobilien einen Preis, der bei mindestens* 10 000 *Euro liegen muss, und bei dem nur Vielfache von* 1000 *erlaubt sind. Zudem soll es möglich sein, für jede Immobilie den Preis pro Quadratmeter zu berechnen. Schließlich ist für jeden Verkauf eine Maklerprovision in Höhe von* 3.45 *Prozent fällig. Es soll jeweils eine geeignete Möglichkeit geben, Grundstücke und Apartments anzulegen. Ungültige Preise sollen eine geprüfte Ausnahme vom Typ `IllegalPriceException` und ungültige Grundstücksgrößen und Wohnflächen eine geprüfte Ausnahme vom Typ `IllegalSizeException` auslösen (die Sie in der nächsten Teilaufgabe selbst implementieren).*

Hinweis: Da sich in der Systembeschreibung dazu keine Vorgaben befinden, müssen Sie sich hier nicht um eine spezifische Zeichenketten-Ausgabe und nicht um einen geeigneten Test auf Gleichheit kümmern.

Lösung:

```
public abstract class RealEstate {

    private static final double BROKER_PROVISION = 3.45;

    private double price;

    protected RealEstate(double price) throws IllegalPriceException {
        setPrice(price);
    }

    private static boolean checkPrice(double price) {
        return (price >= 10000 && price % 1000 == 0);
    }

    public void setPrice(double price) throws IllegalPriceException {
        if (!checkPrice(price))
```

```
            throw new IllegalPriceException("Price must be in multiples of 1000 and cant be
            ↪  lower than 10000");
        this.price = price;
    }

    public double getPrice() {
        return this.price;
    }

    public double getBrokerFee() {
        return 0.01 * BROKER_PROVISION * price;
    }

    public static double getProvision() {
        return BROKER_PROVISION;
    }

    public abstract double pricePerQm();

}
```

Apartment.java

```
public class Apartment extends RealEstate {

    private int area;

    public Apartment(double price, int area) throws IllegalSizeException,
    ↪  IllegalPriceException {
        super(price);
        setArea(area);
    }

    private static boolean checkArea(int area) {
        return (area >= 20 && area <= 120);
    }

    private void setArea(int area) throws IllegalSizeException {
        if (!checkArea(area))
            throw new IllegalSizeException("Area must be between 20 and 120 (both
            ↪  inclusive)");
        this.area = area;
    }

    public int getArea() {
        return this.area;
    }

    @Override
    public double pricePerQm() {
        return getPrice() / getArea();
    }

}
```

Parcel.java

```
public class Parcel extends RealEstate {

    private int size;

    public Parcel(double price, int size) throws IllegalSizeException, IllegalPriceException
    ↪  {
        super(price);
        setSize(size);
    }

    private static boolean checkSize(int size) {
        return (size >= 250);
    }

    private void setSize(int size) throws IllegalSizeException {
        if (!checkSize(size))
            throw new IllegalSizeException("Size can't be smaller than 250");
        this.size = size;
    }

    public int getSize() {
        return this.size;
    }

    @Override
    public double pricePerQm() {
        return getPrice() / getSize();
    }

}
```

b) *(*, Eigene Ausnahmeklassen implementieren, 4 Minuten)*
Implementieren Sie geprüfte Ausnahmeklassen `IllegalPriceException` und `IllegalSizeException` jeweils mit einem Konstruktor, der das Setzen einer individuellen Fehlermeldung erlaubt (weitere Informationen sollen nicht gespeichert werden).

Lösung:

```
@SuppressWarnings("serial")
public class IllegalSizeException extends Exception {

    public IllegalSizeException(String message) {
        super(message);
    }

}


@SuppressWarnings("serial")
public class IllegalPriceException extends Exception {

    public IllegalPriceException(String message) {
        super(message);
    }

}
```

c) *(*, Programmklasse `ImmoApp` implementieren, 4 Minuten)*
Implementieren Sie gemäß folgender Vorgaben eine eigene Programmklasse `ImmoApp` zum Test der vorher implementierten Klassen:

- Erstellen Sie ein Grundstück mit einem Preis von 500 000 Euro und einer Größe von 1000 Quadratmetern und geben Sie den daraus in der Klasse berechneten Preis pro Quadratmeter aus. Sorgen Sie hierbei für eine geeignete Ausnahmebehandlung. Für ungültige Größen soll dabei der Stapel der zur Ausnahme führenden Methodenaufrufe ausgegeben werden, für ungültige Preise nur die Fehlermeldung.
- Erstellen Sie ein Apartment mit einem Preis von 450 000 Euro und einer Wohnfläche von 150 Quadratmetern und geben Sie den daraus in der Klasse berechneten Preis pro Quadratmeter aus. Sorgen Sie hierbei für eine geeignete Ausnahmebehandlung. Für ungültige Größen soll dabei der Stapel der zur Ausnahme führenden Methodenaufrufe ausgegeben werden, für ungültige Preise nur die Fehlermeldung.

Lösung:

```
public class ImmoApp {
    public static void main(String[] args) {
        try {
            Parcel g = new Parcel(500000, 1000);
            System.out.println(g.pricePerQm());
        } catch (IllegalSizeException e) {
            e.printStackTrace();
        } catch (IllegalPriceException e) {
            System.err.println(e.getMessage());
        }

        try {
            Apartment a = new Apartment(450000, 150);
            System.out.println(a.pricePerQm());
        } catch (IllegalSizeException e) {
            e.printStackTrace();
        } catch (IllegalPriceException e) {
            System.err.println(e.getMessage());
        }
    }
}
```