Assignment 3

Abgabe bis zum 4. Mai 2021 (Dienstag) um 23:59 Uhr

Geben Sie Ihr Assignment bis zum 4. Mai 2021 (Dienstag) um 23:59 Uhr auf folgender Webseite ab:
https://assignments.hci.uni-hannover.de
Ihre Abgabe muss aus einer einzelnen zip-Datei bestehen, die den Quellcode und ggf. ein PDF-Dokument bei Freitextaufgaben enthält. Beachten Sie, dass Dateien mit Umlauten im Dateinamen nicht hochgeladen werden können. Entfernen Sie die daher vor der Abgabe die Umlaute aus dem Dateinamen. Überprüfen Sie nach Ihrer Abgabe, ob der Upload erfolgreich war. Bei technischen Problemen, wenden Sie sich an Patric Plattner oder Dennis Stanke. Es wird eine Plagiatsüberprüfung durchgeführt. Gefundene Plagiate führen zum Ausschluss von der Veranstaltung. In dieser Veranstaltung wird ausschließlich die Java/JavaFX Version 11 verwendet. Code und Kompilate anderer Versionen sind nicht zulässig. Ihre Abgaben sollen die vorgegebenen Dateinamen verwenden. Wiederholter Verstoß gegen diese Regel kann zu Punktabzügen führen.

Info

Ab dieser Woche werden die Aufgabenstellungen offener sein, als in den Vorwochen. Wir wollen Ihnen die Möglichkeit geben, mehr eigene Problemlösungen zu finden, statt nach Anleitung zu Programmieren. Deshalb möchten wir Sie dazu ermutigen, sich zunächst eigene Gedanken zu einer Aufgabenstellung zu machen. Sollten Sie weiterhin Hilfe brauchen, wenden Sie sich bitte an das Stud.IP Forum der Übungsveranstaltung oder an die Discord Beratungstutoren.

Wir möchten ihnen nahelegen, Ideen für die Bearbeitung von offeneren Aufgaben im Forum zu diskutieren. Das Posten von Lösungen zu Fragen und Codeausschnitten generell ist verboten, lediglich das Besprechen von Lösungs**ansätzen** ist erlaubt.

Aufgabe 1: LadybugGame 2

In dieser Woche sollen Sie anfangen, die Bausteine, die Sie in der letzten Woche angelegt haben, zusammenzufügen, sodass Sie in Kürze eine brauchbare Version des Spiels programmieren können. Allerdings werden Sie diese Woche noch keine spielbare Version anfertigen, sondern nur die Vorarbeit leisten, sodass dies in der folgenden Woche besser gelingen wird.

Die Musterlösung der Ladybug Game Aufgabe der Vorwoche finden Sie im Stud.IP. Sie können entweder die Musterlösung erweitern, Ihre eigene Lösung der Vorwoche weiterentwickeln, oder Ihre eigene Lösung mithilfe der Musterlösung verbessern und darauf aufbauen.

a) Erstellen Sie ein `Direction` Enum, welches die Himmelsrichtungen darstellen soll. Schreiben Sie die `Ladybug` Klasse so um, dass die Blickrichtung mithilfe des `Direction` Enums kodiert wird.

Ergänzen Sie zu allen Klassen, Methoden und Member-Variablen JavaDoc Kommentare. Halten Sie sich dabei an die JavaDoc Konventionen in Anhang A: JavaDoc Konventionen. Beispiele für JavaDoc Kommentare können Sie hier sehen: Anhang B: JavaDoc Beispiel.

Versehen Sie auch alle Klassen, Methoden und Member-Variablen, die Sie in den folgenden Teilaufgaben verfassen mit JavaDoc Kommentaren.

b) Erstellen Sie eine Klasse `Game`, welche das Spielfeld darstellen soll. Die Klasse soll also ein `NxM` Spielfeld besitzen, welches aus beliebig vielen `Tree`s und genau einem `Ladybug` besteht. Sie können zunächst die `Clover` vernachlässigen, dürfen aber gerne auch `Clover` in das Spielfeld einarbeiten. Dies kann diese Woche noch ein wenig knifflig sein. Aus diesem Grund ist das Einbauen der `Clover` Klasse diese Woche optional.

`Game` soll auch eine Zielkoordinate haben, also eine Koordinate, die das Ziel des Labyrinths darstellen soll. Die Game Klasse soll aber noch nichts besonderes machen, wenn der `Ladybug` das Ziel erreicht.

Erstellen Sie angemessene Konstruktoren für `Game` und geeignete Methoden um das Spielfeld zu bearbeiten.

Implementieren Sie eine Methode, die den aktuellen Stand des Spielfeldes als String zurückgibt.

Sorgen Sie dafür, dass ein `Game` Objekt niemals in einem ungültigen Zustand ist. Das bedeutet, dass immer ein `Ladybug` auf dem Spielfeld sein soll und ein `Ladybug` niemals auf einem `Tree` liegen darf.

c) Erstellen Sie Methoden auf `Game` um den `Ladybug` durch das Spielfeld zu bewegen. Dabei sollen folgende Kommandos möglich sein:

- Der `Ladybug` dreht sich um 90° nach links.
- Der `Ladybug` dreht sich um 90° nach rechts.
- Der `Ladybug` bewegt sich ein Feld nach vorne.
- Der `Ladybug` signalisiert, ob sie sich nach Vorne bewegen kann.
- Der `Ladybug` signalisiert, ob sie am Ziel angekommen ist.

Folgende Kommandos sollen nur dann implementiert werden, wenn Sie in Aufgabe **b)** `Clover` in das Spielfeld eingebaut haben:

- Der `Ladybug` hebt den `Clover` unter sich auf, sofern dort einer ist.

Sie dürfen dabei beliebig Hilfsmethoden auf allen Klassen implementieren.

Sorgen Sie dafür, dass ein `Game` Objekt niemals in einem ungültigen Zustand ist. Das bedeutet, dass immer ein `Ladybug` auf dem Spielfeld sein soll und ein `Ladybug` niemals auf einem `Tree` liegen darf.

d) Entfernen Sie den Inhalt aus der `main` Methode, den Sie letzte Woche schreiben sollten. Erstellen Sie in der Main ein `Game` Objekt, welches mit einem simplen Labyrinth befüllt werden soll (entweder im Aufruf des Konstruktors oder im Nachhinein mithilfe der Setter aus Aufgabe **b)**). Die Gänge Ihres Labyrinths sollten immer nur ein Feld breit sein und alle Wände sollten miteinander verbunden sein (ausnahme dabei ist das Zielfeld, welches die Wand unterbrechen darf). Im Anhang C: Beispiel Labyrinth finden Sie ein Beispiel Labyrinth.

Nutzen Sie danach den Wall Follower Algorithmus um den `Ladybug` durch das Labyrinth zu bewegen. Hier finden Sie eine Beschreibung für den Wall Follower Algorithmus:

1. Wenn der `Ladybug` am Ziel angekommen ist ⟶ Fertig.
2. Der `Ladybug` checkt, ob rechts neben ihr ein `Tree` ist:
   (a) Wenn kein `Tree` neben der `Ladybug` ist, dann dreht sich der `Ladybug` nach rechts und geht ein Feld vorwärts ⟶ Zurück zu 1.
   (b) Wenn ein `Tree` neben dem `Ladybug` ist, dann checkt der `Ladybug`, ob vor ihr ein Baum ist:
      i. Wenn ein `Tree` vor der `Ladybug` ist, dann dreht sich der `Ladybug` so lange nach links, bis kein `Tree` vor ihr ist und geht ein Feld nach vorne ⟶ Zurück zu 1.
      ii. Wenn kein `Tree` vor der `Ladybug` ist, geht der `Ladybug` ein Feld nach vorne ⟶ Zurück zu 1.

Bestehenskriterien

- Ihr Code kompiliert ohne Fehler.
- Alle Klassen, Methoden und Member-Variablen sind mit Javadoc Kommentaren, wie in Anhang A: JavaDoc Konventionen beschrieben, versehen.
- `Game` kann das vorgegebene Spielfeld darstellen und Ausgeben.
- `Ladybug` kann wie beschrieben durch das Spielfeld navigiert werden.
- In der `main` wird ein Labyrinth erstellt, durch welches der `Ladybug` läuft.

Aufgabe 2: Wechselgeldrechner

In dieser Aufgabe sollen Sie verstärkt mit Enums arbeiten um einen Wechselgeldrechner zu programmieren.

a) Erstellen Sie ein Enum `EuroDenomination`, welches alle zurzeit im Umlauf befindliche Euro Münzen und Scheine - also {1, 2, 5, 10, 20, 50} Cent Münzen, {1, 2} Euro Münzen und {5, 10, 20, 50, 100, 200} Euro Scheine - als Enum Keywords enthält.

Erstellen Sie zwei Methoden auf diesem Enum:

- Eine Methode, die den *Cent*-Wert eines Zahlungsmittels zurückgibt.
- Eine Methode, die eine eindeutige Repräsentation des Zahlungsmittels als String zurückgibt.

b) Erstellen Sie eine Klasse `EuroCalculator`, welche eine statische Methode enthält, die folgende Parameter entgegennimmt

- Ein Array an `EuroDenomination`, welches das gegebene Bargeld bei einer Bezahlung darstellen soll
- Einen `int`, welcher die Kosten eines Einkaufs in *Cent* darstellen soll.

Die Methode soll ein Array an `EuroDenomination` zurückgegeben, welche das Wechselgeld darstellen soll. Das Wechselgeld, das zurückgegeben wird, soll dabei in möglichst großen Münzen/Scheinen berechnet werden.

Sollte zu wenig Geld gegeben werden, soll die Methode das Programm beenden.

c) Erstellen Sie eine `Product` Klasse, die einen Namen für das Produkt und einen Preis (in Cent) enthält. Erstellen Sie passende Getter/Setter und Konstruktoren.

Erstellen Sie eine `Application` Klasse, welche eine `main` Methode enthalten soll. In dieser sollen Sie eine numerierte Liste an `Product` mit ihren Namen und Preisen auf die Konsole ausgeben. Was für Produkte mit welchen Preisen Sie angeben, bleibt Ihnen überlassen.

Wir stellen ihnen eine Klasse namens `ConsoleInput` bereit, welche zwei Methoden enthält:

- `public static String getChoice(String[] arr, String msg)`, welche dem Nutzer fragt einen String aus `arr` zu wählen. `msg` ist eine Nachricht, die dem Nutzer angezeigt wird.
- `public static String[] get MultipleChoces(String[] arr, String msg)`, welche ähnlich wie `getChoiceFromArray()` funktioniert, allerdings mehrere

Nutzen Sie mindestens eine dieser Methoden und implementieren Sie einen Interaktiven Shop, in dem der Nutzer ein `Product` auswählen kann und dann angeben kann, mit welchem Bargeld er bezahlen möchte. Lesen Sie dazu das JavaDoc der `ConsoleInput` Klasse, die im `doc` Ordner in der Assignment3.zip datei ist, und derer Methoden. Das Javadoc können Sie einsehen, indem Sie die `index.html` Datei öffnen.

Lösungshinweise

- Probieren Sie die Methoden der `ConsoleInput` Klasse auf der jshell aus um herauszufinden, wie sie funktionieren, bzw. wie Sie diese in ihrem Programm verwerten können.
- Nutzen Sie ein `switch` Statement um zwischen den Ausgaben der `ConsoleInput` Methoden auszuwählen. Ein Beispiel dazu finden Sie im Anhang D: Beispiel zu `switch` Statements.
- Sie können das Programm mit `System.exit(1);` beenden.

Bestehenskriterien

- Ihr Code kompiliert ohne Fehler
- Alle Klassen, Methoden und Member-Variablen sind mit Javadoc Kommentaren, wie in Anhang A: JavaDoc Konventionen beschrieben, versehen.
- `EuroDenomination` enthält alle Gültigen Euro Bargeld Einheiten und deren Wert, und Bezeichnung.
- Die Wechselgeldmethode gibt immer das optimale Wechselgeld zurück (möglichst große Einheiten). Die Methode beendet das Programm, wenn zu wenig Geld gegeben wurde.
- Die `main` Methode implementiert einen Shop, so wie in der Aufgabe beschrieben.

Aufgabe 3: Debugging

In diesen Aufgaben werden Sie einen fehlerhaften Java Codeabschnitt bekommen. Diese Fehler können **syntaktisch** oder **semantisch** sein. Es kann sich dabei um Compiler- oder Laufzeitfehler handeln. Sie dürfen den Code kompilieren und ausführen um die Fehler zu finden. Arbeiten Sie in der Template-Datei `Debug.java`. Diese finden Sie im Stud.IP. Nutzen Sie Kommentare, um die Fehler zu Kennzeichnen. Gegeben ist der folgende Codeabschnitt:

Debug.java

```
enum Operator {
  ADD, SUBTRACT, MULTIPLY, DIVIDE
}

class Expression {
  double left_, right_;
  Operator op_;

  Expression(double left, double right, Operator op){
    this.left_  = left;
    this.right_ = right;
    this.op_    = op;
  }

  double evaluate() {
    switch (this.op_) {
      case ADD:
        return this.left_ + this.right_;
      case SUBTRACT:
        return this.left_ - this.right_;
      case MULTIPLY:
        return this.left_ * this.right_;
      case DIVIDE:
        return this.left_ / this.right_;
    }
  }
}

class Debug {

  public static void main(String[] args) {
    Operator[] ops = new Operator[5];
    ops[0] = ADD;
    ops[1] = SUBTRACT;
    ops[2] = MULTIPLY;
    ops[3] = DIVIDE;

    Expression[] exp = new Expression[ops.length];
    for (int i = 0; i < ops.length; ++i) {
      exp[i] = new Expression(2, 3, ops[i]);
    }

    for (int i = 0; i < ops.length; ++i) {
      System.out.println(exp[i].evaluate());
    }
  }
}
```

Der Codeabschnitt enthält zwischen 3-5 Fehler. Kompilieren und führen Sie den Code aus. Suchen Sie anhand der Fehlermeldungen des Compilers oder der Laufzeitfehlermeldungen Fehler im Code und korrigieren Sie diese. Kommentieren Sie am Ende der jeweiligen Zeile, was Sie korrigiert haben.

Erstellen Sie während des Fehlerkorrektur einen Blockkommentar unter dem Code, in dem Sie die Fehler dokumentieren. Schreiben Sie die Zeile(n) des Fehlers auf und beschreiben Sie den Fehler. Kopieren Sie die Fehlermeldung, sofern es eine gab, anhand welcher Sie den Fehler erkannt haben. Die Dokumentation soll wie in folgendem Beispiel aussehen:

```
1  public class Debug { //K: class falsch geschrieben (ckass)
2
3  ...
4
5  /*
6  ...
7  Zeile 1: class Keyword falsch geschrieben (ckass):
8  Fehlermeldung:
9  **************
10 Debug.java:1: error: class, interface, or enum expected
11 public ckass Debug {
12        ^
13 **************
14 Der Compiler erwartet eines der drei oben angegebenen Keywords, hat aber nur das falsch
       ↪ geschriebene ckass bekommen.
15 ...
16 */
```

Bestehenskriterien

- Ihr korrigierter Code funktioniert so wie in den Kommentaren im Code beschrieben.
- Alle korrigierten Fehler haben einen kurzen Kommentar, der beschreibt, warum die Stelle im alten Code nicht funktioniert hat.
- Alle Fehlermeldungen (Compiler und Laufzeit) sind in einem Blockkommentar `/* ... */` unter dem Code in der `Debug.java`.

Lösungshinweise

- Im Code vorgegebene Kommentare beschreiben immer das **korrekte** Verhalten des Programms.

Anhang A: JavaDoc Konventionen

Alle JavaDoc Kommentare sind in `Englisch` zu erstellen, da dies die Standardsprache für Code-Dokumentation ist.

Ein JavaDoc Kommentar zu einer Klasse soll immer folgende Informationen enthalten:

- Der erste Satz (*brief description*) soll eine kurze Beschreibung der Klasse enthalten.
- Der Text unter der *brief description* soll eine detailliertere Beschreibung dazu haben, was die Klasse genau darstellt, aber **keine Implementierungsdetails**.
- Der @author Tag soll den Namen der Person, die für diese Klasse zuständig ist, beinhalten.
- Der @version Tag soll normalerweise eine Versionsnummer und das Datum der letzten Änderung beinhalten. Da Versionsnummern in der Einzelübung nicht relevant sind, soll hier nur das Datum der letzten Änderung stehen.

Ein JavaDoc Kommentar zu einer Membervariable soll immer folgende Informationen enthalten:

- Eine kurze Beschreibung, was diese Variable darstellt.

Ein JavaDoc Kommentar zu einer Methode soll immer folgende Informationen enthalten:

- Der erste Satz (*brief description*) soll eine kurze Beschreibung der Funktionalität der Methode enthalten.
- Der Text unter der *brief description* soll (falls nötig) eine genauere Beschreibung der Funktionalität der Methode beinhalten. Dazu gehören ggf. mögliche unerwartete Nebenwirkungen der Methode, allerdings **keine Implementierungsdetails**.
- Der @author Tag soll den Namen der Person, die diese Methode zuletzt bearbeitet hat, beinhalten.
- Die @param Tags sollen die Eingabe parameter beschreiben.
- Der @return Tag soll den Rückgabewert beschreiben.

Anhang B: JavaDoc Beispiel

```java
package a.b.c;

import java.lang.Math;

/**
 * This class provides multiple Methods to calculate integers in arrays.
 * This class also tracks the amount of Method calls on this class.
 *
 * @author  Patric Plattner<patric.plattner@hci.uni-hannover.de>
 * @version 2021 April 01
 */
class Operators {
    /** Counts method calls on this class. Calls to getters of this Variable should be ignored.*/
    protected static int count_;


    /**
     * This Method adds the absolute values of given Values.
     * This Method has the side effect of altering the values of the input array to their Math.abs value.
     *
     * @author     Patric Plattner <patric.plattner@hci.uni-hannover.de>
     * @param  arr Array of integers to add up.
     * @return     Sum of all Math.abs values of the input array.
     */
    public static int addAbs(int[] arr) {
        Operators.count_++;
        int res = 0;
        for (int i = 0; i < arr.length; ++i) {
            arr[i] = Math.abs(arr[i]);
            res += arr[i];
        }
        return res;
    }


    /**
     * This Method multiplies the absolute values of given Values.
     * This Method has the side effect of altering the values of the input array to their Math.abs value.
     *
     * @author     Patric Plattner <patric.plattner@hci.uni-hannover.de>
     * @param  arr Array of integers to multiply.
     * @return     Product of all Math.abs values of the input array.
     */
    public static int mulAbs(int[] arr) {
        Operators.count_++;
        int res = 1;
        for (int i = 0; i < arr.length; ++i) {
            arr[i] = Math.abs(arr[i]);
            res *= arr[i];
        }
        return res;
    }

    /**
     * Getter for {@link a.b.c.Operators#count_}.
     * {@link a.b.c.Operators#count_} is a Variable that tracks how often Methods of this class were called.
     * @returns count
     */
    public static int getCount() {
        return Operators.count_;
    }
}
```

Anhang C: Beispiel Labyrinth

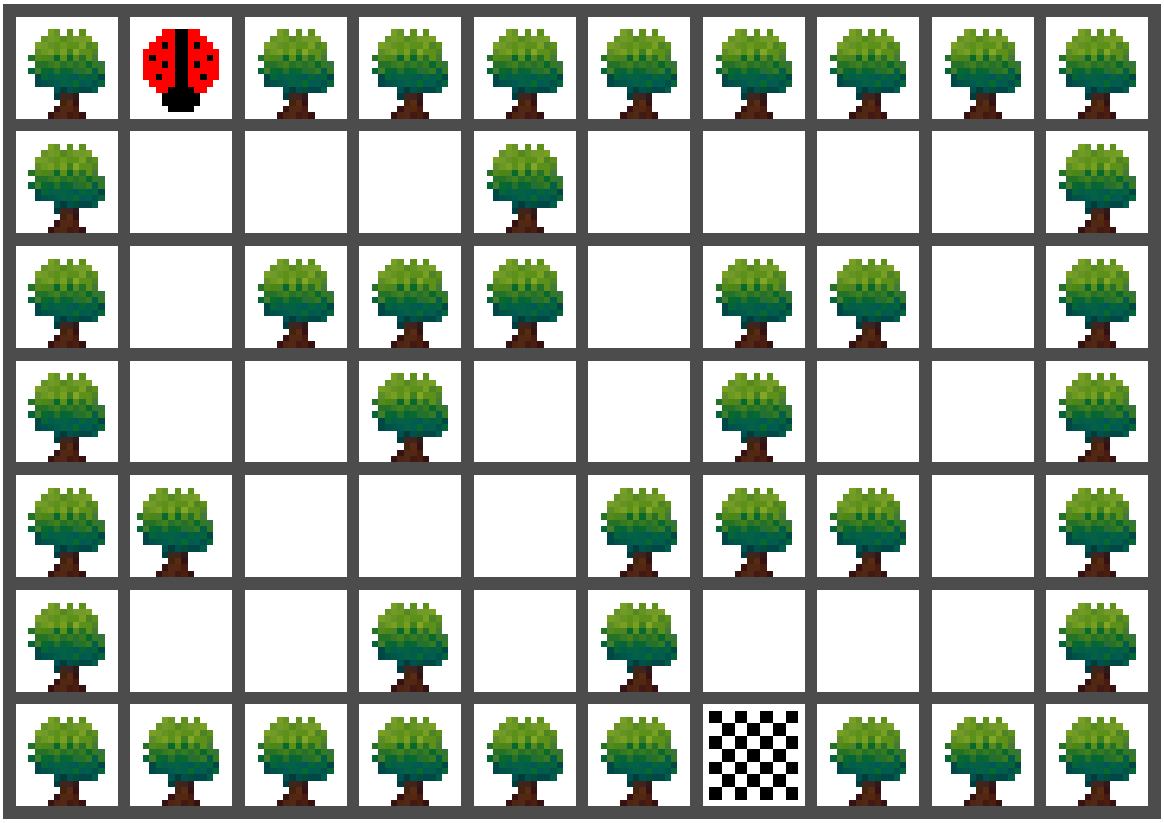

Abbildung 1: Hier sehen Sie ein Beispiel Labyrinth für Aufgabe 1.

Anhang D: Beispiel zu `switch` Statements

```
1  class Demo {
2    public static void main(String[] args) {
3      String switchString = "A";
4      //String switchString = "B";
5      switch(switchString) {
6      case "A":
7        System.out.println("A");
8        break;
9      case "B":
10       System.out.println("B");
11       break;
12     }
13   }
14 }
15 //Prints "A" if line 4 is commented out, prints "B" if line 3 is commented out.
```