Assignment 3

Abgabe bis zum 12. Mai 2020 (Dienstag) um 23:59 Uhr

Geben Sie Ihr Assignment bis zum 12. Mai 2020 (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.

Aufgabe 1: Fortführung: Einfach verkettete Listen

In dieser Aufgabe sollen Sie an Ihrer Abgabe aus der letzten Woche weiterarbeiten. Sollten Sie diese Aufgabe nicht bearbeitet haben oder sollten Sie mit Ihrer Lösung nicht zufrieden sein, finden Sie eine Musterlösung im Stud.IP.

a) Dokumentieren Sie Ihren Quellcode mit Javadoc Kommentaren. Versehen Sie dabei alle Klassen, Membervariablen und Methoden mit entsprechenden Javadoc Kommentaren. Alle Javadoc Kommentare sind in `Englisch` zu erstellen, da dies die Standardsprache für Code-Dokumentation ist. Halten Sie sich dabei an die folgenden Konventionen:

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**.

Vorlesung: apl. Prof. Matthias Becker becker@hci.uni-hannover.de
Übung: Patric Plattner patric.plattner@hci.uni-hannover.de
Organisation: M.Sc. Dennis Stanke dennis.stanke@hci.uni-hannover.de

- 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.

Hier ein Beispiel für Javadoc Kommentare:

```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 2020 May 04
 */
class Operators {
    /** Counts method calls on this class. */
    private 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 count.
     * @returns count
     */
    public static int getCount() {
```

```
59          return Operators.count_;
60      }
61  }
```

Alle Klassen/Methoden/Membervariablen aus den folgenden Teilaufgaben sollen auf dieselbe Art und Weise dokumentiert werden.

b) Implementieren Sie folgende Methoden auf entweder der `ListHead` oder der `ListNode` Klasse:

- `ListHead`: `void insert(int index, int i)` und `void insert(int index, ListNode node)` sollen die jeweiligen Elemente in die Liste an n-te Stelle einfügen. Beispiel:

```
1  //[listHead]->(a)->(b)->null
2  listHead.insert(1, 2)
3  //[listHead]->(a)->(2)->(b)->null
```

- `ListNode`: `void insertAfter(int i)` und `void insertAfter(ListNode node)`, die das Jeweilige Element nach dem Listenknoten einfügen. Insbesondere soll bei der zweiten Variante die gesamte Liste eingefügt werden. Beispiel:

```
1  //(a)->(b)->null       (c)->(d)->null
2  a.insertAfter(c)
3  //(a)->(c)->(d)->(b)->null
4  a.insertAfter(1)
5  //(a)->(1)->(c)->(d)->(b)->null
```

Überlegen Sie dabei, ob die jeweilige Methode `public`, `private` oder package private (kein Modifier) sein sollte. Schreiben Sie in das Javadoc Kommentar, warum Sie welche Sichtbarkeit gewählt haben. Gehen Sie auch **alle anderen** Methoden der Klassen durch und schauen Sie, ob diese `public`, `private` oder package private sein sollten und schreiben Sie Ihre Begründung in das Javadoc Kommentar.

Bedenken Sie dabei, dass die `ListHead` Klasse ein *high level interface* bieten soll und Methoden, die zu nahe an der Implementierung sind, dem Endnutzer nicht bereitgestellt werden sollen. Insbesondere geht es in dieser Aufgabe darum, dass Sie sich überlegt haben, wie das Interface der Klasse aussehen sollte und dass Ihre Entscheidungen gut begründet sind.

Bestehenskriterien

- Ihr Code ist, wie in dieser Aufgabe beschrieben, mit Javadoc Kommentaren versehen.
- Ihre Javadoc Kommentare enthalten eine Begründung bezüglich der Wahl der Sichtbarkeit der Methoden.
- Ihr Code kompiliert ohne Fehler.
- Die Funktionalität Ihrer Implementierung entspricht den Anforderungen der Aufgabe.

Aufgabe 2: Kara

In dieser Aufgabe werden die Sichtbarkeiten der einzelnen Methoden nicht vorgegeben. Ähnlich wie in Aufgabe 1 sollen Sie selber entscheiden, welche Methoden `public`, `private` oder package private (kein modifier) sein sollen. Ihr Quellcode soll mit Javadoc Kommentaren versehen werden, so wie in Aufgabe 1a) beschrieben. Auch hier sollen Sie ihre Begründung für die gewählte Sichtbarkeit in das Javadoc Kommentar schreiben.

Kara (https://www.swisseduc.ch/informatik/karatojava/kara/) ist ein Programm, das Programmiereinsteigern Grundlegende Programmierkonzepte (insbesondere endliche Automaten) näher bringen soll. Es handelt sich um ein Programm, in dem der Anwender einen Marienkäfer durch ein Labyrinth steuern soll, indem der Nutzer den Käfer wie einen endlichen Automaten mit den Aktionen 'vorwärts laufen', 'links drehen', 'rechts drehen', 'aufheben' und 'ablegen' vorprogrammiert. In dieser Aufgabe sollen Sie eine deutlich vereinfachte Version von Kara implementieren.

a) In dieser Aufgabe arbeiten Sie in der `Aufgabe2.zip` Datei. Entpacken Sie die ZIP Datei. Sie sollten eine Package Struktur mit (zum großen Teil leeren) Quelltextdateien vorfinden. Die `aufgabe2.util.Util` Klasse hat eine `static void clearScreen()` Methode, die die das Konsolenfenster leert. Die drei Enum Klassen sind ebenso vorgegeben. Die restlichen `.java` Dateien sollen wie folgt mit Funktionalität gefüllt werden.

Zunächst sollen Sie das Spielfeld implementieren. Dazu dient die `PlayingField` Klasse. Diese soll ein 10x10 Felder großes `PlayingFieldTile[][]` Array haben, welches das Spielfeld darstellt. Schreiben Sie Getter und Setter, die einzelne Felder des Arrays bearbeiten/zurückgeben.

Schreiben Sie einen Konstruktor `PlayingField(PlayingFieldTile[][] field)`, der das Programm mit `System.exit(1)` beendet, wenn die Dimensionen des Parameters nicht 10x10 sind. Schreiben Sie auch einen Konstruktor, der keine Eingaben entgegennimmt und ein leeres (GROUND) 10x10 Spielfeld generiert.

Schreiben Sie eine Methode `boolean canWalk(int x, int y)`, welche dann wahr zurückgibt, wenn die gegebenen Koordinaten auf dem Spielfeld liegen und diese begehbar sind.

Schreiben Sie eine Methode `String str()`, in welcher Sie Das Spielfeld als String ausgeben. Das Ausgabeformat bleibt Ihnen überlassen.

Erstellen sie in der Main Methode, welche in der `Main` Klasse sein soll, ein Spielfeld, befüllen sie dieses mit Hindernissen und geben sie dieses auf der Kommandozeile aus.

b) Sie haben nun die Spielkarte implementiert. Nun müssen Sie die Karte so abarbeiten, dass Sie einen Spieler auf die Karte setzen können.

Fügen Sie der `PlayingField` Klasse Membervariablen hinzu, mit denen Sie die Position und die Blickrichtung einer Spielfigur auf dem Spielfeld darstellen können. Schreiben Sie Getter und Setter für Position und Blickrichtung des Spielers. Der Setter für die Spielerposition soll Eingaben ignorieren, wenn diese einen Spieler auf eine unbegehbare Position legen soll. Fuer dir Richtung soll das vorgegebene `Direction` enum genutzt werden.

Passen Sie die Konstruktoren so an, dass Sie auch die Position und Blickrichtung des Spielers entgegennehmen. Passen Sie die `String str()` Methode so an, dass Sie auch die Spielerposition und Blickrichtung angibt (z.B. könnte der Spieler als Buchstabe N, S, E, W dargestellt werden)

Implementieren Sie nun folgende Methoden auf der `PlayingField` Klasse:

- `boolean playerCanWalk()`: Gibt wahr zurück, wenn der Spieler einen Schritt in die momentane Blickrichtung machen kann.

- `boolean playerWalk()`: Lässt den Spieler vorwärts laufen. Gibt wahr zurück, wenn der Spieler gelaufen ist, falsch, wenn dies nicht möglich war.
- `boolean playerTurnRight()`: Lässt den Spieler nach rechts drehen. Gibt wahr zurück, wenn der Spieler gedreht wurde, falsch, wenn dies nicht möglich war.
- `boolean playerTurnLeft()`: Lässt den Spieler nach links drehen. Gibt wahr zurück, wenn der Spieler gedreht wurde, falsch, wenn dies nicht möglich war.

Bearbeiten sie den Konstruktoraufruf in der Main Methode so, dass sie einen Spieler auf das Spielfeld setzen.

c) Nun, da das Spielfeld fertig implementiert ist, sollen Sie die `Simulator` Klasse implementieren. In dieser Klasse soll nun eine Abfolge an Aktionen und ein dazugehöriges Spielfeld gespeichert werden und wenn die `void execute()` Methode aufgerufen wird, sollen die Aktionen der Reihe nach ausgeführt werden und das Feld nach jedem Schritt auf die Konsole ausgegeben werden. Aktionen werden durch das `Action` Enum dargestellt, das in der `Aufgabe2.zip` enthalten ist.

Erstellen Sie die `Simulator` Klasse in der dazugehörigen Datei. Die Klasse soll eine `Action[]` Membervariable und eine `PlayingField` Membervariable haben. Erstellen Sie einen Konstruktor, der beide Felder per Eingabe initialisiert und einen Konstruktor, der nur das `PlayingField` entgegennimmt und das `Action[]` Array als leeres Array der Größe 100 initialisiert. Verketten Sie die Konstruktoren, wie Sie es in der Vorlesung gelernt haben. Schreiben Sie Getter und Setter für alle Membervariablen.

Schreiben Sie eine `void execute()` Methode, in der Sie die Aktionen des `Action[]` Arrays abgehen, die jeweilige Aktion auf dem Spielfeld ausführen und danach das Spielfeld ausgeben.

Schreiben Sie eine `bool addAction(Action action)` Methode, die die gegebene Action an letzter Stelle des Actionsarrays legt. Ihnen könnte dabei evtl. eine Membervariable, die trackt, wie viele Actions bereits im Array sind.

Erstellen sie einen `Simulator`, mit dem sie einen Spieler durch ein selbst erstelltes Labyrinth schicken. Führen sie die Simulation aus.

d) OPTIONAL Erweitern Sie das Programm so, dass Sie Gegenstände im Labyrinth platzieren können und der Spieler diese aufheben und ablegen kann. Ihre Vorgehensweise ist hier nicht genau vorgegeben. In zukünftigen Aufgaben werden die Aufgabenstellungen ein wenig offener und die genauen Details der Implementierung bleiben dann Ihnen überlassen und diese Aufgabe ist als eine kleine optionale Vorbereitung auf diese Aufgaben zu verstehen.

Bestehenskriterien

- Ihr Code ist, wie in dieser Aufgabe beschrieben, mit Javadoc Kommentaren versehen.
- Ihre Javadoc Kommentare enthalten eine Begründung bezüglich der Wahl der Sichtbarkeit der Methoden.
- Ihr Code kompiliert ohne Fehler.
- Ihre Ausgaben entsprechen den Ausgaben der Beispiele. Die genaue Formatierung kann von den Beispielen abweichen.
- Die zu implementierenden Methoden funktionieren auch mit anderen Eingaben 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 Operator.ADD:
        return this.left_ + this.right_;
      case Operator.SUBTRACT:
        return this.left_ - this.right_;
      case Operator.MULTIPLY:
        return this.left_ * this.right_;
      case Operator.DIVIDE:
        return this.left_ / this.right_;
    }
  }
}

class Debug {

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