Fakultät IV - Elektrotechnik und Informatik
Algorithmen und Datenstrukturen SoSe 2016
Brock / Deimel / Foucard / Mabrouk / Sieverling

Aufgabenblatt 3

letzte Aktualisierung: 15. May, 20:45 Uhr

Ausgabe: 15.05.2016
Abgabe: 25.05.2016 19:59

Thema: *Interfaces, Vererbung, Exceptions*

Wichtige Ankündigungen

- Zur Erinnerung: Unit tests dürfen geteilt werden. Lösungen dürfen auf keinen Fall geteilt werden! Wir testen auf Plagiate.
- Bitte achtet auf die verwendete Java Version. Programme die mit JDK 1.7 (Java Version 7) nicht kompilieren, sind gleichbedeutend mit einer fehlerhaften Abgabe!
- Bitte achtet ebenso auf die korrekte Kodierung des Programmcodes. Wir erwarten für Abgaben das UTF-8 Format. Bei der Verwendung von sprach- sowie länderspezifischen Kodierungen auf eurem Computer (speziell Windows ist hier betroffen) ist nicht sichergestellt, dass der Programmcode auf anderen Systemen kompilierbar ist. In Eclipse koennt ihr den Zeichensatz auf UTF-8 festlegen:
  Window->Preferences->General->Workspace->Text File Encoding-> Set to UTF-8
  Verzichtet zur Sicherheit auch auf Umlaute in den Kommentaren.

Abgabe

Die folgenden Dateien müssen für eine erfolgreiche Abgabe im svn Ordner
`Tutorien/txx/Studierende/deinname@TU-BERLIN.DE/Abgaben/`
eingecheckt sein:

Geforderte Dateien:	
`Blatt03/src/Pair.java`	Aufgabe 1.1, 2.1
`Blatt03/src/SteelFactory.java`	Aufgabe 3.4

Als Abgabe wird nur die neueste Version im svn gewertet.

1. Aufgabe: Generics

In manchen Situationen will man die Objektklasse nicht von vornherein festlegen. Ein klassischer Fall sind Mengen und Listen, deren Verhalten unabhängig von der Klasse der enthaltenen Elemente definiert ist. Für diese Zwecke bietet Java das Konzept der *Generics*.
Generische Typen sind Platzhalter für Variablentypen, die erst zur Laufzeit bestimmt sind. Der generische Typ wird in der Klassendeklaration mit eineckigen Klammern (`<T>`) angezeigt, der enthaltene Name ist dabei beliebig wählbar, solange er eindeutig ist.

In der Implementierung innerhalb der Klasse kann dann dieser generische Typ `T` wie ein gewöhnlicher Variablentyp verwendet werden, so können z.B. Variablen vom Typ `T` angelegt werden, oder Methodenparameter den Typ `T` besitzen.
Alternativ könnte man ohne Generics auch für jeden einzelnen möglichen Objekttyp eine separate Methode implementieren (Polymorphismus), allerdings muss dann sehr viel Programmcode dupliziert werden, und selbst kleinste Änderungen werden aufwändig und fehleranfällig.
Hinweis: Das Konzept der *Generics* in Java ist äquivalent zu dem Konzept der *Templates* in C++.

1.1. (Übung) Kleine Einführung in Java-Generics (30 Punkte) In der folgenden Aufgabe wollen wir eine Klasse `Pair<T>` implementieren, die genau zwei Objekte eines generischen Typs speichern kann. Die Attribute können von beliebigem Typ sein, müssen aber beide denselben Typ haben.

Eure Aufgabe ist es, folgende Methoden in der Klasse `Pair` zu implementieren:

- einen Konstruktor `Pair(T first, T second)`, der zwei Objekte übergeben bekommt und speichert (wie, ist euch überlassen).
- getters: `getFirst()` und `getSecond()`
- setters: `setFirst()` und `setSecond()`
- swapper: eine Methode `swap()`, die den Platz der beiden gespeicherten Objekte vertauscht.

Verwendet dafür bitte die Vorgaben in der Datei `Pair.java` und kommentiert euren Code sinnvoll.

2. Aufgabe: Exceptions

Ausnahmen (Exceptions) sind ein Konzept zur Vereinfachung des Programmflusses. Anstatt einer expliziten Abfrage auf mögliche Fehler (z.B. durch einen Rückgabewert oder einer globalen `error` Variable) werden Fehler durch das "Werfen" (throw) einer Exception angezeigt. Der normale Programmfluß wird sofort unterbrochen, und die Exception an die aufrufende Funktion hoch gerreicht. Ohne besondere Vorkehrungen wandert die Exception den gesamten Aufrufstapel (Call Stack) hoch und verursacht schlußendlich einen Programmabruch. Dieser Standardfall passiert ohne eine einzige zusätzliche Programmverzweigung, was die Lesbarkeit eines Programms deutlich erhöht.
Exceptions werden vor allem dann verwendet, wenn Zustände auftreten die im lokalen Kontext der Funktion nicht sinnvoll behandelt werden können, z.B. schlägt der Befehl `java.io.FileInputStream(``config.txt'')` fehl wenn "config.txt" nicht existiert. Manchmal können aber übergeordnete Funktionen sinnvoll reagieren, z.B. könnten alternative Dateinamen ausprobiert werden, oder Standardwerte gesetzt werden. Für solche Fälle gibt es die try-catch Syntax. Hier ein konzeptionelles Beispiel eines sinnvollen Exception Handlings:

```
1  try {
2     java.io.File file = new java.io.File("config.txt");
3     this.parseConfiguration(file);
4  } catch (IOException e) {
5     this.setDefaultConfiguration();
6  }
```

Wird innerhalb des try-Blocks eine Exception vom Typ IOException geworfen, führt das zur sofortigen Ausführung der Anweisungen im catch-Block.

2.1. (Übung) Kleine Einführung in Exception Handling (30 Punkte) Erweitert die Implementierung der Pair Klasse aus dem vorherigen Beispiel um die folgende Methode:

- `toString()` liefert einen String in genau der Form: "first, second" zurück (beachtet bitte, dass dem Komma genau ein Leerzeichen folgt).

Schreibt die Methode mit Hilfe von Exception Handling so um, dass die Methode immer eine String zurück gibt. Gebt im Falle einer Exception ersatzweise "<not implemented>, <not implemented>" zurück.

Hinweis: Praktisch jede Java-Klasse beerbt eine sinnvolle toString() Implementierung oder implementiert diese ohne Exceptions auszulösen. Testet euren Code deshalb mit der in den Unit Tests definierten Klasse `NotImplementedDataType`:

```
/**
 * Bogus class that triggers an exception when somebody tries
 * to convert it to a string
 *
 *
 **/
class NotImplementedDataType {
    @Override
    public String toString() {
        throw new RuntimeException("I am evil.");
    }
}
```

Testet aber auch den Normallfall!

Verwendet für diese Aufgabe bitte die Vorgaben in der Datei `Pair.java` und kommentiert euren Code sinnvoll.

3. Aufgabe: Interfaces

Interfaces in Java vereinbaren eine Ansammlung bestimmter Methoden (genau genommen deren Signaturen), welche von beliebig vielen Klassen implementiert werden kann. Auf diese Weise kann die Verwendung von Objekten vereinheitlicht werden, auch wenn diese keine gemeinsame Oberklasse haben.
Interfaces stellen in der Regel gut definierte, allgemeine Verhaltensweisen zur Verfügung, z.B. stellt das `Comparable` Interface Methoden zur Festellen der Gleichheit des Objektes mit einem anderen zur Verfügung. Andere Beispiele sind die Interfaces Clonable (Ein Objekt kann samt seines derzeitigen internen Zustandes dupliziert werden), List (Objekt verhält sich wie eine Liste), und Iterator (das Objekt generiert eine Abfolge von Elementen durch wiederholtes (iteratives) Aufrufen der Methode `next()`).

3.1. (Tut) Interfaces und abstrakte Klassen in Java Interfaces ähneln abstrakten Klassen in vielen Punkten. So können sie nicht instanziiert oder als Typ verwendet werden. Sie beinhalten abstrakte Methoden, also Methoden ohne "Rumpf". Es gibt jedoch auch einige wichtige Unterschiede zwischen abstrakten Klassen und Interfaces:

- Interfaces erben von Interfaces, abstrakte Klassen von abstrakten Klassen (jeweils per Schlüsselwort `extends`).
- Interfaces enthalten niemals die Programmlogik, während Abstrakte Klassen auch ausimplementierte Programmlogik beinhalten können. Interfaces enthalten auch keine Attribute.
- Klassen implementieren Interfaces (angezeigt durch das Schlüsselwort **`implements`**), aber beerben abstrakte Klassen (Schlüsselwort **`extends`**).
- Eine Klasse kann mehrere Interfaces implementieren, aber nur von einer abstrakten Klasse erben.
- Abstrakte Klassen können Interfaces implementieren, Interfaces aber nicht von abstrakten Klassen erben.

3.2. (Tut) Interfaces und Generics Warum sind Interfaces oft mit Hilfe von Generics definiert?

Besprecht auch die Möglichkeit, dass generische Interfaces nur für bestimmte Klassen zu implementiert sein (wie im folgenden Übungsbeispiel), oder für jegliche Klassen (z.B. für Listen). Wann ist letzteres möglich? Was gewinnt man dadurch?

3.3. (Tut) Implemetierung des Comparable Interface Ein häufig verwendetes Interface in Java ist das Interface `Comparable<T>`. Es definiert die Signatur der Methode **`int`** `compareTo(T o)` und ist dafür gedacht, beliebige Objekte, die das Interface implementieren, miteinander vergleichen zu können.

Implementiert das Interface `Comparable<Shape>` in der abstrakten Klasse `Shape`, dass zwei (zweidimensionale) Shapes aufgrund ihrer Fläche vergleicht. Die Vorgabe enthält für Unit tests zusätzlich die Unterklassen `CircleShape`, `RectangleShape`, `HexagonShape`, die die abstrakte `Shape` Klasse beerben und implementieren.

```
public abstract class Shape implements Comparable<Shape>  {

    /**
     * if difference below tolerance parameter, shapes are considered equal
     */
    private static double tol = .1;

    /**
     * Calculates the area of a shape.
     * @return the area the shape fills
     */
    abstract double calculateArea();

    /**
     * Scales the shape by a factor.
     * @param factor the scaling factor
     */
    abstract void scale(double factor);


    /**
     * Compares two shapes using the area to compare.
     *
     * @param s the other Shape
     */
    @Override
    public int compareTo(Shape s) {
        // TODO
        return 0;
    }
}
```

Achtung: Die Klassenvariable tol (Toleranz) gibt die absolute Differenz an, unterhalb welcher zwei Fließkommavariablen als gleich angesehen werden sollen. Das ist notwendig, um numerische Ungenauigkeiten zu ignorieren.

3.4. (Übung) Programmieren gegen Interfaces (40 Punkte) In dem Uni-Projekt "Entwicklung neuer Verwaltungssoftware" geht es unter anderem darum, das bereits bestehende System zur Firmenverwaltung zu verbessern. Dafür wurde von der verantwortlichen Gruppe bereits das Interface

`Enterprise` geschaffen, welches Methoden beschreibt, die eine konkrete Firma in dem System auf jeden Fall zur Verfügung stellen muss. `Enterprise` erbt zusätzlich von dem Interface `Comparable`, so dass es möglich ist, Firmen miteinander zu vergleichen.

Eure Aufgabe ist es, die neue Klasse `SteelFactory` zu entwickeln, welche das Interface `Enterprise` implementiert (definiert in enterprise.java). Stahlwerke sollen mit anderen Unternehmen anhand ihrer Mitarbeiteranzahl verglichen werden können.

Beachtet, dass ein negativer Rückgabewert gleichbedeutend mit einer geringeren Mitarbeiterzahl in Relation zum verglichenen Stahlwerk bedeutet:
`kleinesStahlwerk.compare(grossesStahlwerk) < 0`

Diese Semantik ist konform zu der Beschreibung des Interfaces. Der Name des Stahlwerkes ist eine beliebige Zeichenkette (`String`). Nutzt den Konstruktor für das Setzen des Namens.

Hinweis: Da Arbeiter nach und nach eingestellt werden können, muss die Liste aller Arbeiter von variabler Größe sein. Nutze dafür die vorgegebene Membervariable ArrayList `workers`.

- Implementiert den Konstruktor SteelFactory() und getName()
- Implementiert addWorker() und getWorkerCount()
- Implementiert das interface Enterprise