Fakultät IV - Elektrotechnik und Informatik
Algorithmen und Datenstrukturen SoSe 2017
Brock / Mabrouk / Sieverling / Stahl

Aufgabenblatt 3

letzte Aktualisierung: 12. May, 09:53 Uhr

Ausgabe: 15.05.2017
Abgabe: 24.05.2017 19:59

Thema: *Interfaces, Vererbung, Exceptions*

Wichtige Ankündigungen

- Betrugsversuch (sowohl die "Geber" als auch die "Nehmer") bedeutet Nichtbestehen des Kurses.
- Wir verwenden Plagiatserkennungssoftware für Eure Programmieraufgaben. Bitte keinen Quelltext von anderen Personen oder Quellen verwenden.
- 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 könnt 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
`Blatt03/src/FastaParser.java`	Aufgabe 2.1
`Blatt03/src/FastaException.java`	Aufgabe 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 (20 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.
- toString: die Methode liefert eine String-Repräsentation des Objekts. Der String soll folgendes Format haben: "first, second" (beachtet bitte, dass dem Komma genau ein Leerzeichen folgt).

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) Einführung in Exception Handling (50 Punkte) In dieser Aufgabe schreibt Ihr einen Parser für das FASTA-Format. Das FASTA-Format wird für Proteinsequenzen genutzt. Es besteht aus zwei Elementen, dem Identifier und der Sequenz. Eine Identifierzeile beginnt mit einer spitzen Klammer (>, größer als) gefolgt von dem Identifier. Nach einer Identifierzeile folgen eine oder mehrere Zeilen, die die Sequenz enthalten. Eine gültige Sequenz besteht bei uns **nur** aus Buchstaben. Eine neue Proteinsequenz wird erneut mit einer Identifierzeile eingeleitet. Hier sind ein paar Beispiele.

```
1 >Protein1
2 OK
```

```
1 >Protein1
2 DASIST
3 AUCHOK
4 >Protein2
5 SOWIESOOK
```

```
1 >NICHT
2 >OK
```

Implementiert folgende Methoden der Klasse FastaParser:

- `readSequences(File f)` liefert eine ArrayList mit String Objekten. Ein String Objekt entspricht einer Sequenz.
- `handleException(File f)` liefert entweder eine String-Repräsentation der ArrayList mit Sequenzen von readSequence() oder im Falle einer Exception die dazugehörige Nachricht

Die Methode readSequence soll aussagekräftige und hilfreiche Fehlermeldungen liefern. Die folgenden Exceptions sollen explizit geworfen werden.

- Sequenzidentifier fehlt (**FastaException** mit der Nachricht: **Expected identifier**)
- Sequenz fehlt oder ist unzulässig (**FastaException** mit der Nachricht: **Expected sequence on line X**, ersetzt X mit der zugehörigen Zeilennummer in der FASTA Datei)

Der notwendige Code um die Datei einzulesen ist bereits vorgegeben, aber teilweise auskommentiert.

Die Methode handleException ruft readSequence auf und liefert eine String-Repräsentation des Ergebnisses, oder im Falle einer Exception, die zugehörige Nachricht (String, siehe getMessage()).

Die Klasse **FastaException** erbt von Exception und enthält unsere benutzerdefinierte Exception.

Bitte achtet darauf das die Ressource **RandomAccessFile** auch im Falle einer Exception ordnungsgemäß geschlossen wird, um Resource-Leaking zu vermeiden. In Java gibt es dafür den finally-Block und ab Java 7 try-with-resources.

Hinweis: Die String-Methode **trim()** kann benutzt werden um Leerzeichen am Anfang und Ende eines Strings zu entfernen. Hier gibt es eine Übersicht über weitere u.U. hilfreiche String-Methoden wie z.B. **substring**. Ob ein Zeichen ein Buchstabe ist, kann z.B. mit der Character-Methode **isLetter()** getestet werden. Exceptions sind normale Objekte mit Feldern und Methoden.

Verwendet für diese Aufgabe bitte die Vorgaben in der Datei `FastaParser.java` und `FastaException.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?

Bespricht auch die Möglichkeit, dass generische Interfaces nur für bestimmte Klassen implementiert sind (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.

```
1  public abstract class Shape implements Comparable<Shape>  {
2
3      /**
4       * if difference below tolerance parameter, shapes are considered equal
5       */
6      private static double tol = .1;
7
8      /**
9       * Calculates the area of a shape.
10      * @return the area the shape fills
```

```
11        */
12      abstract double calculateArea();
13
14      /**
15       * Scales the shape by a factor.
16       * @param factor the scaling factor
17       */
18      abstract void scale(double factor);
19
20
21      /**
22       * Compares two shapes using the area to compare.
23       *
24       * @param s the other Shape
25       */
26      @Override
27      public int compareTo(Shape s) {
28          // TODO
29          return 0;
30      }
31  }
```

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 (30 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 ist:
`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