Praktikum Grundlagen der Programmierung

Prof. Dr. Harald Räcke, R. Palenta, A. Reuss, S. Schulze Frielinghaus WS 2016/17

Übungsblatt 11 Abgabefrist: 23.01.2017 (vor 05:00 Uhr)

Σ Punkte: 19 | Σ Bonuspunkte: 2

Aufgabe 11.1 (P) Erweiterung geometrische Figuren

In dieser Aufgabe sollen die Klassen der geometrischen Formen von den beiden vorherigen Aufgabenblättern erweitert werden.

- Die Klassen vom Typ `Prisma` und `Grundflaeche` sollen untereinander vergleichbar gemacht werden, indem die Klassen das Interface `Comparable<Prisma>` bzw. `Comparable<Grundflaeche>` implementieren [1].
  - Prismen werden nach Größe des Volumens verglichen;
  - `Grundflaechen` werden nach Größe der Fläche verglichen.
  - Zur Vereinfachung ignorieren wir Abweichungen durch Gleitkommaarithmetik und gehen davon aus, dass solche nicht vorkommen. D.h. die Flächen zweier Grundflächen sind gleich, wenn die Differenz exakt 0 ist. Die gleiche Annahme treffen wir für das Volumen von Prismen.
- Implementieren Sie dazu die Methode `public int compareTo(Prisma o)` bzw. `public int compareTo(Grundflaeche o)` in der Klasse `Prisma` bzw. `Grundflaeche`. Halten Sie sich dabei an die Vorgaben der Java Dokumentation für das Interface `Comparable<T>` unter `https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html`. Beachten Sie insbesondere, dass ggf. eine `NullPointerException` geworfen wird.
- Testen Sie die Implementierung, indem Sie jeweils eine `LinkedList` mit `Grundflaeche`-Objekten und eine LinkedList mit `Prisma`-Objekten mittels `Collections.sort(List<T> list)` sortieren.
  Beispiel: `List<Prisma> pf = new LinkedList<Prisma>(); Collections.sort(pf);`
- Als zweiten Schritt sollen die Methoden `istQuadrat` und `zuQuadrat` in ein Interface `Quadrierbar` ausgelagert werden, welches nur von solchen `Grundflaechen` implementiert werden soll, die auch ein Quadrat darstellen können.
- Zuletzt erstellen wir ein weiteres Interface `Polygon` mit nur einer Methode `int getEckenAnzahl()`, welche die Zahl der in einer Grundfläche enthaltenen Ecken zurückgibt. Implementieren Sie das Interface `Polygon` für alle Grundflächen, die eine endliche Anzahl von Ecken besitzen.
- Testen Sie `Quadrierbar` und `Polygon`, indem Sie über eine `LinkedList` des Typen `Comparable`, die verschiedene `Prisma`- und `Grundflaeche`-Objekte enthält, iterieren und mittels `instanceof` die Verfügbarkeit der beider Interfaces abfragen und, falls vorhanden, die aus diesen Interfaces ermittelbaren Informationen ausgeben.
- Deklarieren Sie alle Objektvariablen als `final`, wo das möglich ist.

Hinweis: Zur Verwendung der genannten Klassen und Methoden aus der Java-Standardbibliothek benötigen Sie die folgenden Imports.

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
```

[1] siehe dazu `https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html`

Aufgabe 11.2 (P) Typisch generisch

Generics (zu deutsch: Generische Programmierung) sind ein Konzept in Java, welches erst in Version 1.5 hinzugekommen ist. D.h. die Idee dahinter war nie Teil der initialien Implementierung der Sprache. Dies merken wir an verschiedensten Stellen. Um abwärtskompatibel zu sein, also generischen Java-Code ab Version 1.5 mit nicht generischem Java-Code von Versionen 1.4 und früher zu verknüpfen, hat man sich entschieden, generische Datentypen zur Compile-Zeit durch nicht generische Datentypen zu ersetzen. Dieser Vorgang wird i.A. als „type erasure“ (zu deutsch: Typlöschung) bezeichnet. Schauen wir uns das an einem Beispiel an:

```
public class A<T> {

    public void foo(T x) { }
}

public class B extends A<String> {

    public void foo(String x) { }
}
```

In der abgeleiteten Klasse `B` schränken wir den generischen Datentyp `T` auf den konkreten Typ `String` ein. Des Weiteren überschreiben wir die Methode `foo`, die jetzt einen `String` erwartet. Betrachten Sie nun folgendes Codefragment:

```
B b = new B();
b.foo("Hello Student!");

A a = b;
a.foo(42);
```

Das Codefragment typt korrekt und lässt sich ohne Probleme (Warnungen/Fehler) vom Compiler übersetzen. Führen wir den Code aus, dann erhalten wir eine `ClassCastException`. Wieso? Schauen wir uns dazu mal den Code an, der nach der „type erasure“ entsteht:

```
public class A {

    public void foo(Object x) { }  // T wird durch Object automatisch
                                   // vom Compiler ersetzt
}

public class B extends A {         // <String> wurde entfernt

    public void foo(String x) { }

    // ...
}
```

Der Compiler hat den Typ `T` durch `Object` in der Klasse `A` ersetzt. Das bedeutet aber nun, dass die Methode `foo` aus der Klasse `B` nicht mehr die Methode `foo` aus der Klasse

`A` überschreibt, da die Methodensignaturen nicht mehr übereinstimmen. Damit nach der „type erasure“ die Methode `foo` wieder in der Klasse `B` überschrieben wird, fügt der Compiler automatisch eine weitere Methode `foo` hinzu. Der Code sieht schlussendlich wie folgt aus:

```
public class B extends A {          // <String> wurde entfernt

    public void foo(String x) { }

    public void foo(Object x) {     // diese Methode wird automatisch
        foo((String) x);            // vom Compiler erstellt
    }
}
```

Damit die Methode `foo` aus der Klasse `A` in der Klasse `B` wieder überschrieben wird, fügt der Compiler eine sogenannte „bridge“ Methode ein. In dieser wird ein Cast vorgenommen. D.h. das Programm typt korrekt liefert aber zur Laufzeit eine `ClassCastException`, da ein `Integer`-Objekt nicht in ein `String`-Objekt gecastet werden kann.

Aufgrund der „type erasure“ können wir in der Klasse `A` auch keine Methode `foo(Object x)` definieren, d.h. folgender Code compiliert nicht:

```
public class A<T> {

    public void foo(T x) { }

    public void foo(Object x) { }  // compiliert nicht
}
```

Denn nach der „type erasure“ würden ansonsten zwei Methoden mit der gleichen Signatur existieren.

Die offizielle Dokumentation[2] sagt dazu folgendes:
Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

Wir wissen nun, dass zur Kompilezeit alle generischen Datentypen ersetzt werden. Gibt es weitere Fallstricke?

[2] `https://docs.oracle.com/javase/tutorial/java/generics/erasure.html`

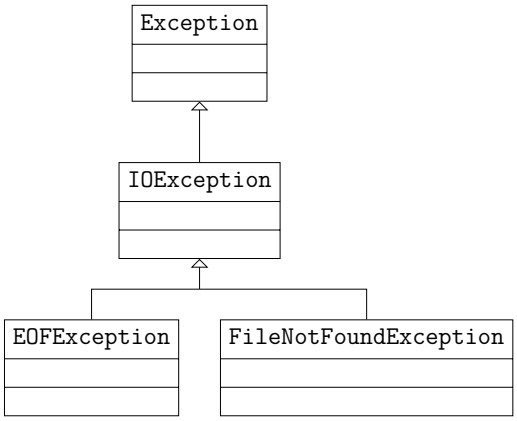
```java
A<Integer> a1 = new A<Integer>();
A<String>  a2 = new A<String>();

if (a1 instanceof A && a2 instanceof A)
    System.out.println("instanceof: yes");

if (a1.getClass() == a2.getClass())
    System.out.println("getClass(): yes");
```

In beiden Fällen evaluieren die Ausdrücke zu `true`. Dies ist insofern unschön, dass wir zur Laufzeit nicht mehr zwischen verschiedenen generischen Typen von der selben Klasse unterscheiden können. Das fällt uns besonderns negativ auf, wenn wir eine `equals`-Methode implementieren wollen. Hier können wir zum Beispiel nicht mehr zwischen einer Liste von Strings und einer Liste von Integern unterscheiden. D.h. `LinkedList<String>` und `LinkedList<Integer>` haben zur Laufzeit den gleichen Typ, nämlich `LinkedList`. Im Javasprech werden solche Typen auch *raw types*[3] genannt.

Quintessenz: All diese Probleme sind Folgen der Designentscheidung, dass Abwärtskompatibilität so wichtig für die Sprachdesigner war.

Aufgabe 11.3 (P) Exceptions

Betrachten Sie folgenden Ausschnitt aus der Klassenhierarchie von Exceptions in Java und folgende Java-Implementierung der Klasse `ExceptionTest`:




```java
import java.io.*;
public class ExceptionTest {
   public static void main (String[] args) {
      try {
        // ...
      }
      catch (EOFException e) {
         System.out.println("EOFException");
      }
      catch (IOException e) {
         System.out.println("IOException");
      }
      catch (Exception e) {
         System.out.println("Exception");
      }
      System.out.println("ENDE");
   }
}
```

1. An der durch „...“ gekennzeichneten Stelle im `try`-Block stehe ein Programmstück, durch das Exceptions vom Typ `EOFException`, `IOException` oder `FileNotFoundException` geworfen werden können.
   Was wird bei Ausführung der `main`-Methode ausgegeben, falls dabei im `try`-Block
   i) als erstes eine Ausnahme vom Typ `EOFException` geworfen wird,
   ii) als erstes eine Ausnahme vom Typ `FileNotFoundException` geworfen wird, oder

[3] `https://docs.oracle.com/javase/tutorial/java/generics/rawTypes.html`

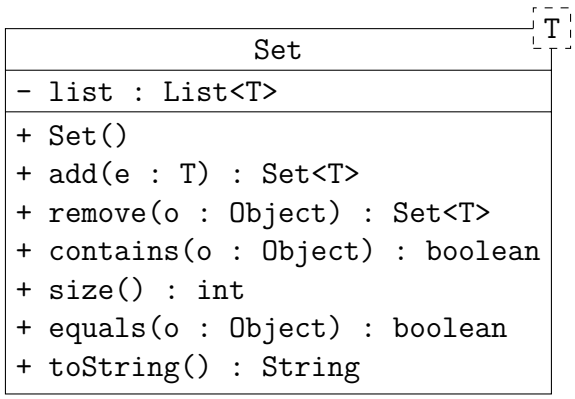
iii) gar keine Ausnahme geworfen wird?

2. Was wird bei Ausführung der `main`-Methode ausgegeben, falls dabei im `try`-Block als erste Ausnahme eine Division durch 0 auftritt?

Aufgabe 11.4 (H) Resistente Mengen [8 Punkte + 2 Bonuspunkte]

Verwenden Sie für diese Aufgabe nur die erlaubten Java-Methoden (siehe Anhang).

Implementieren Sie eine *unveränderliche* Datenstruktur, die das Verhalten einer Menge abbildet. Unveränderlich bedeutet, dass jede Membervariable `final` sein muss. Möchte man also ein Element `e` der Menge `E` hinzufügen, so darf/kann nicht die Menge `E` selber verändert werden, sondern es muss ein neues Objekt `E'` erzeugt werden, welches das neue Element `e` beinhaltet sowie alle alten Element der Menge `E`, i.e. `E'` $= \texttt{E} \cup \{\texttt{e}\}$. Um die Elemente einer Menge zu repräsentieren, verwenden wir eine Liste. Da wir gefordert hatten, dass die Menge unveränderlich sein muss, fordern wir auch, dass die Liste unveränderlich sein muss. D.h. alle Membervariablen der Liste müssen `final` sein. Für die Listenimplementierung dürfen Sie sich an Ihren eigenen Listenimplementierungen sowie an den Lösungsvorschlägen aus vorherigen Aufgaben orientieren. Die Menge selber soll Elemente von einem generischen Typ `T` enthalten und somit natürlich auch die Liste.



Im Folgenden sei `s` ein `Set`-Objekt von einem generischen Typ `T`:

- Der parameterlose Konstruktor erstellt ein Objekt welches eine leere Menge repräsentiert
- Die Methode `s.add(T e)` gibt ein `Set`-Objekt zurück, dass das Element `e` enthält sowie alle Elemente, die in `s` enthalten sind. Ist das Element `e` schon in `s` enthalten, dann soll `s` zurück gegeben werden. Ist `e` gleich `null`, dann soll eine `NullPointerException` geworfen werden.
- Die Methode `s.remove(Object o)` gibt ein `Set`-Objekt zurück, das alle Element von `s` enthält, außer dem Element `o`. Ist `o` gleich `null`, dann soll eine `NullPointerException` geworfen werden.
- Die Methode `s.contains(Object o)` gibt `true` zurück wenn das Element `o` in der Menge enthalten ist und andernfalls `false`.
- Die Methode `size` gibt die Kardinalität der Menge zurück.

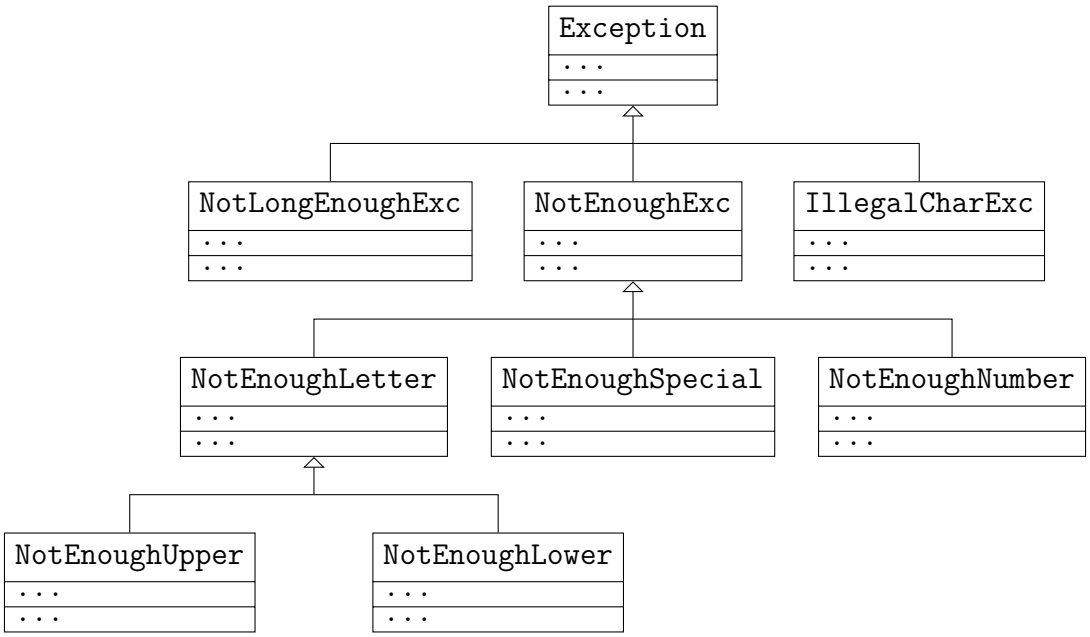
- Die Methode `s.equals(Object o)` erfüllt die üblichen Eigenschaften, die der Java-Standard fordert. Des Weiteren wird `true` zurück gegeben, wenn die Mengen `s` und `o` die gleichen Elemente enthält. Andernfalls wird `false` zurück gegeben. Beachten Sie die üblichen Eigenschaften einer Menge wie z. B. $e \in E \implies E = E \cup \{e\}$ oder $e \notin E \implies E = E \setminus \{e\}$.
- Die Methode `s.toString()` gibt einen String der Form $\{x_1, \ldots, x_n\}$ zurück, wenn die Menge `s` die Elemente x_i, $1 \leq i \leq n$ enthält und n die Kardinalität der Menge ist. Der zurück gegeben String enthält also eine String-Repräsentation aller enthaltener Elemente der Menge.

BONUSAUFGABE: Implementieren sie einen Iterator für die Menge, d.h. die Klasse `Set<T>` muss das Interface `Iterable<T>` implementieren (die Membervariablen des Iterators müssen nicht `final` sein).

Aufgabe 11.5 (H) Passwort-Exceptions [11 Punkte]

Verwenden Sie nur die erlaubten Java-Methoden.

In dieser Aufgabe wollen wir einige Exception-Klassen (Klassen, die von der Klasse `Exception` erben), die bei der Überprüfung der Gültigkeit eines Passworts verwendet werden können, implementieren. Dafür soll die folgende Klassenhierarchie umgesetzt werden, wobei `Exception`, die Klasse `Exception` der Java-Standardbibliothek ist.



Für ein Passwort können die folgenden Mindestanforderungen gestellt werden, die bei Missachtung zu einer entsprechenden Exception führen:

- Das Passwort muss eine Mindestlänge haben, andernfalls wird eine `NotLongEnoughExc`-Exception geworfen.
- Das Passwort muss eine Mindestanzahl an Großbuchstaben enthalten, andernfalls wird eine `NotEnoughUpper`-Exception geworfen.
- Das Passwort muss eine Mindestanzahl an Kleinbuchstaben enthalten, andernfalls wird eine `NotEnoughLower`-Exception geworfen.

- Das Passwort muss eine Mindestanzahl an Sonderzeichen enthalten, andernfalls wird eine `NotEnoughSpecial`-Exception geworfen.
- Das Passwort muss eine Mindestanzahl an Ziffern enthalten, andernfalls wird eine `NotEnoughNumber`-Exception geworfen.
- Das Passwort darf bestimmte Sonderzeichen *nicht* enthalten, andernfalls wird eine `IllegalCharExc`-Exception geworfen.

Für die konkrete Implementierung gelten folgende Anforderungen:

1. Die Klasse `NotLongEnoughExc` hat zwei private `int`-Variablen `should` und `is`. Diese repräsentieren die minimale Länge (`should`), die ein Passwort haben muss und die echt kleinere Länge (`is`), die das Passwort, das die Exception auslöst, hat. Die beiden Variablen werden im Konstruktor `public NotLongEnoughExc(int should, int is)` entsprechend gesetzt. Die `toString()`-Methode liefert eine Fehlermeldung in Form eines Strings, die unter Verwendung der beiden Membervariablen auf die Missachtung der Mindestlänge des Passworts hinweist.

2. Die Klasse `NotEnoughExc` hat zwei `int`-Variablen `should, is`. Diese repräsentieren die Mindestanzahl Zeichen einer bestimmten Kategorie, die ein Passwort enthalten muss und die echt kleinere Anzahl an Zeichen, die das Passwort, das die Exception auslöst, hat. Die beiden Variablen werden entpsrechend im Konstruktor `public NotEnoughExc(int should, int is)` gesetzt.

3. Die Klasse `NotEnoughLetter` hat einen Konstruktor `public NotEnoughLetter(int should, int is)`, der die beiden Variablen der Oberklasse `NotEnoughExc` sinngemäß initialisiert.

4. Die Klasse `NotEnoughUpper` hat einen Konstruktor `public NotEnoughUpper(int should, int is)`, der die beiden Variablen der Oberklasse `NotEnoughExc` sinngemäß initialisiert. Die Methode `toString()` liefert eine Fehlermeldung in Form eines Strings, die unter Verwendung der beiden Membervariablen auf die Missachtung der Mindestanzahl an Großbuchstaben im Passwort hinweist.

5. Die Klasse `NotEnoughLower` hat einen Konstruktor `public NotEnoughLower(int should, int is)`, der die beiden Variablen der Oberklasse `NotEnoughExc` sinngemäß initialisiert. Die Methode `toString()` liefert eine Fehlermeldung in Form eines Strings, die unter Verwendung der beiden Membervariablen auf die Missachtung der Mindestanzahl an Kleinbuchstaben im Passwort hinweist.

6. Die Klasse `NotEnoughSpecial` hat einen Konstruktor `public NotEnoughSpecial(int should, int is)`, der die beiden Variablen der Oberklasse `NotEnoughExc` sinngemäß initialisiert. Die Methode `toString()` liefert eine Fehlermeldung in Form eines Strings, die unter Verwendung der beiden Membervariablen auf die Missachtung der Mindestanzahl an Sonderzeichen im Passwort hinweist.

7. Die Klasse `NotEnoughNumber` hat einen Konstruktor `public NotEnoughNumber(int should, int is)`, der die beiden Variablen der Oberklasse `NotEnoughExc` sinngemäß initialisiert. Die Methode `toString()` liefert eine Fehlermeldung in Form eines Strings, die unter Verwendung der beiden Membervariablen auf die Missachtung der Mindestanzahl an Ziffern im Passwort hinweist.

8. Die Klasse `IllegalCharExc` hat eine private Variable `char used`, die ein Zeichen repräsentiert, das in einem Passwort nicht verwendet werden darf. Die Klasse hat einen Konstruktor `IllegalCharExc(char used)`, wobei das Zeichen `used` ein Zeichen ist, das im Passwort nicht enthalten sein darf. Im Konstruktor wird die Membervariable entsprechend initialisiert. Die Methode `toString()` liefert eine Fehlermeldung in Form eines Strings, die unter Verwendung der Membervariablen auf die Verwendung des nicht erlaubten Zeichens im Passwort hinweist. In der `toString()`-Methode soll auch eindeutig auf nicht darstellbare Zeichen hingewiesen werden: Stellen Sie sicher, dass es bei Verwendung der nicht darstellbaren Steuerzeichen `\n` , `\t` , `\r` , `\b` , `\f` eine verbale Beschreibung dieser Steuerzeichen gibt. Ein Backslash kann in einem `String` in Java durch `\\` dargestellt werden, z.B. liefert `System.out.println("line break (\\n)");` auf der Konsole die Ausgabe `line break (\n)`.

Schreiben Sie nun eine Klasse `Password` mit einem Konstruktor

```
public Password(int nrUpperShould, int nrLowerShould, int nrSpecialShould,
                int nrNumbersShould, int lengthShould, char[] illegalChars)
```

und einer Methode `public void checkFormat(String pwd)` sowie einer `main`-Methode. Dabei soll gelten:

a) Die Klasse `Password` hat die privaten `int`-Variablen `nrUpperShould`, `nrLowerShould`, `nrSpecialShould`, `nrNumbersShould`, `lengthShould`, die die Mindestanzahl an Großbuchstaben, Kleinbuchstaben, Sonderzeichen (alle Zeichen, die erlaubt sind und keine Groß- und Kleinbuchstaben (A-Z bzw. a-z) oder Ziffern sind) und Ziffern im Passwort sowie die Mindestlänge des Passworts repräsentieren. Außerdem gibt es die private `char[]`-Variable `illegalChars`, die alle Zeichen enthält, die nicht im Passwort enthalten sein dürfen. Alle Klassenvariablen sollen entsprechend im Konstruktor initialisiert werden.

b) Die öffentliche Methode `void checkFormat(String pwd)` soll den übergebenen String `pwd` auf die oben vorgestellten Kriterien eines Passworts, die durch den Konstruktor exakt festgelegt wurden, überprüfen. Wird ein Kriterium verletzt, soll eine entsprechende Exception geworfen werden. D.h. die Methode hat den Zusatz `throws IllegalCharExc, NotEnoughExc, NotLongEnoughExc`

c) In der `main`-Methode soll ein Objekt der Klasse `Password` erzeugt werden. Die konkreten Kriterien für Passwörter, also die Parameter bei der Erzeugung des Objekts dürfen frei gewählt werden. Überprüfen Sie einen `String` mithilfe der `checkFormat`-Methode des zuvor erzeugten `Password`-Objekts auf die dadurch festgelegten Kriterien eines Passworts. Wird eines der Kriterien verletzt, soll der Rückgabewert der `toString()`-Methode der entsprechenden Exception auf der Konsole ausgegeben werden.

Vermeiden Sie Codeduplikate so gut wie möglich durch Verwendung von `super(...)`. Sie können in allen zu implementierenden Klassen davon ausgehen, dass die Konstruktoren nur mit sinnvollen/gültigen Parametern aufgerufen werden. Alle vorgegebenen Membervariablen müssen als `final` deklariert werden.

Erlaubte Java-Methoden

MiniJava:

```
public static int readInt()
public static int readInt(String s)
public static int read()
public static int read(String s)
public static String readString()
public static String readString(String s)
public static void write(String output)
public static void write(int output)
public static int drawCard()
public static int dice()
```

returns an integer from the interval $[2, 11]$

returns an integer from the interval $[1, 6]$

Object bzw. allen anderen Klassen/Interfaces usw.:

```
public boolean equals(Object obj)
public final Class getClass()
public final void notify()
public final void notifyAll()
public final void wait() throws InterruptedException
```

String:

```
public char charAt(int index)
public boolean isEmpty()
public int length()
```

System:

`System.out.print(x)` prints the object `x` to the console

`System.out.println(x)` prints the object `x` to the console and terminates the line

Thread:

```
public void interrupt()
public final void join() throws InterruptedException
public void start()
public void run()
```

Selbstgeschriebene Methoden, die selber nur erlaubte Methoden verwenden, sind erlaubt, sofern dies nicht explizit in der Aufgabenstellung verboten wurde. Methoden, die im gegebenen Programmgerüst definiert wurden, dürfen verwendet werden.