Übungen – Zugriffsrechte und Exceptions

Aufgabe 1 – Zugriffsrechte in der Fahrzeug-Hierarchie

Die Klasse `Fahrzeug` hat zwei Konstruktoren, die beide bis jetzt noch `public` sind. Dabei ist der Konstruktor mit den zwei Argumenten nur für die erbenden Klassen gedacht, also für die Klassen, die von der Klasse **`Fahrzeug`** abgeleitet sind, wie die beiden Klassen `Motorrad` oder `Pkw`. Der andere Konstruktor ist sinnlos geworden, da wir die Klasse abstract definiert haben. Man könnte den übrig gebliebenen Konstruktor nicht `public` deklarieren, sondern `protected`.

Aufgabe 2 – Zugriffsrechte in der Waren-Hierarchie

Ach ja, wir haben gar keine Waren-Hierarchie. Aber nehmen wir mal an, wir erstellen eine Unterklasse von Ware, `AenderbareWare`, die auch in der Lage sein soll, nachträglich (also auch nach der Instanziierung) über eine zusätzliche Methode
`void setBezeichnung(String neueBezeichnung)`
die Warenbezeichnung ändern zu können. Die Oberklasse soll aber keine zusätzliche Schnittstelle bekommen. Wie realisieren Sie das?

Aufgabe 3 – Fragen über Fragen (nur nachdenken, nicht programmieren!)

Gegeben sind die beiden folgenden Übersetzungseinheiten.
Erste Übersetzungseinheit:

```
package com.x.eins;

public class Q
{
   private String eq1;
   String eq2;
   protected String eq3;
   public String eq4;
}
```

Zweite Übersetzungseinheit:

```
package com.x.eins;

class R extends Q
{
   private String er1;
   String er2;
   protected String er3;
   public String er4;
}
```

Was lässt sich in den folgenden Beispielen **nicht** compilieren?

Beispiel a:

```
package com.x.zwei;

public class S extends Q
{
   public void anzeigen()
   {
      System.out.println(eq1);
      System.out.println(eq2);
      System.out.println(eq3);
      System.out.println(eq4);
   }
}
```

Beispiel b:

```
package com.x.eins;

class T
{
   private Q q = new Q();

   public void anzeigen()
   {
      System.out.println(q.eq1);
      System.out.println(q.eq2);
      System.out.println(q.eq3);
      System.out.println(q.eq4);
   }
}
```

Beispiel c:

```
package com.x.zwei;

import com.x.eins.Q;

class T
{
   private Q q = new Q();

   public void anzeigen()
   {
      System.out.println(q.eq1);
      System.out.println(q.eq2);
      System.out.println(q.eq3);
      System.out.println(q.eq4);
   }
}
```

Beispiel d:

```
package com.x.zwei;

import com.x.eins.R;

public class U
{
   private R r = new R();

   public void anzeigen()
   {
      System.out.println(r.eq1);
      System.out.println(r.eq2);
      System.out.println(r.eq3);
      System.out.println(r.eq4);

      System.out.println(r.er1);
      System.out.println(r.er2);
      System.out.println(r.er3);
      System.out.println(r.er4);
   }
}
```

Beispiel e:

```
package com.x.zwei;

import com.x.eins.R;

public class V extends R
{
   public void anzeigen()
   {
      System.out.println(eq1);
      System.out.println(eq2);
      System.out.println(eq3);
      System.out.println(eq4);

      System.out.println(er1);
      System.out.println(er2);
      System.out.println(er3);
      System.out.println(er4);
   }
}
```

Beispiel f:

```
package com.x.zwei;

import com.x.eins.Q;

public class W extends Q
{
   private Q q = new Q();

   public void anzeigen()
   {
      System.out.println(eq1);
      System.out.println(eq2);
      System.out.println(eq3);
      System.out.println(eq4);

      System.out.println(q.eq1);
      System.out.println(q.eq2);
      System.out.println(q.eq3);
      System.out.println(q.eq4);
   }
}
```

Aufgabe 4 – Exceptions

Im Paket java.lang befinden sich einige Exception-Typen mit relativ flexiblen Einsatzmöglichkeiten. Z.B. kann eine `IllegalArgumentException` überall dort verwendet werden, wo einer Methode Parameter übergeben werden, die so nicht zulässig sind. Oder eine `IllegalStateException` kann anzeigen, dass eine Methode aufgerufen wurde, die im momentanen Zustand nicht richtig funktioniert.
Es gilt bei der Neuentwicklung von Java-Programmen immer die Abwägung zu treffen, ob man bestehende Exceptions einsetzen will oder neue, aussagekräftige eigene Exception-Typen definiert. Beachten Sie, dass beide Exception-Typen unchecked Exceptions sind. Auch diese Diskussion will normalerweise geführt werden: „setze ich unkontrollierte, unchecked Exceptions oder deklarierte checked Exceptions ein?“

4.1 – Exceptions verwenden

Überarbeiten Sie Ihre Klasse Ware, so dass sie sich mit Exceptions wehrt, wenn unsinnige oder ungültige Werte eingegeben werden.

4.2 – Exceptions provozieren und abfangen

Probieren Sie, Ware mit ungültigen Werten zu füttern und beobachten Sie die Reaktion.
Schreiben Sie Testcode in einer `main`-Methode, der die Ausnahmen provoziert und abfängt.

4.3 – Eigene Exceptions

Erstellen Sie eine unchecked Exception `WareException` und leiten Sie von dieser Exception weitere eigene, passend benannte Fehlerklassen ab und ersetzen Sie die in 4.1 gewählten System-Exceptions durch eigene.

Aufgabe 5 – Fehlerhafte Bestellprozesse

In der bisherigen Variante von Bestellung, die mit einer fixen Anzahl an Positionen arbeitet, müssen wir eigentlich immer beim Hinzufügen einer Bestellposition damit rechnen, dass die Bestellliste (das Array) voll ist. Diese Fehlersituation soll Nutzern unserer Klasse sehr transparent gemacht werden und daher entscheiden wir uns, dafür eine checked Exception zu verwenden.

5.1 – Checked Exception

Erstellen Sie eine eigene checked Exception-Klasse `BestellException`. Die `BestellException` bezieht sich immer auf eine `Bestellposition` und den Füllgrad des Arrays, daher ist es sinnvoll, beispielsweise den Füllgrad als Bestandteil der Exception-Message anzugeben und die betroffene Bestellposition per Methode am Exception-Objekt nachträglich abfragbar zu machen.

5.2 – Werfen der Exception

Sorgen Sie dafür, dass die Klasse Bestellung bei `nimmAuf()` nun diese Exception wirft und passen Sie den restlichen Code so an, dass er wieder sauber compiliert. Provozieren Sie in Ihrem bisherigen Code die Fehlersituation.

Aufgabe 6 – Tests

Schreiben Sie JUnit-Tests, die die veränderten Bedingungen aus Aufgabe 4 und 5 testet. Beachten Sie, dass das Framework Unterstützung* anbietet, um Methoden mit Exceptions zu testen.

*Hinweis: Die Unterstützung fällt je nach Version von JUnit komplett unterschiedlich aus. JUnit 4 bietet die Möglichkeit, bei einem Test zu deklarieren, welche Exception „erwartet“ wird für einen fehlerfreien Test. In JUnit 5 wurde diese Möglichkeit aus welchen Gründen auch immer komplett entfernt und es stehen nur Testmöglichkeiten über Java 8 Sprachmittel, die uns zum jetzigen Zeitpunkt noch nicht zur Verfügung stehen, bereit. Alternativ kann man sich aber mit einem Trick behelfen: schreiben Sie einen try-catch-Block und erzeugen Sie an den Stellen, die Sie mit einer Exception **nicht** erreichen sollten, mit Hilfe von `fail()` einen Testfehler.