

Musterlösung 2

Aufgabe 1 (Geschachtelte Klassen)

- i.) Nein, es besteht keine Kenntnis über die zugehörigen Konten ausgehend von einem Bankexemplar. Will man einen solchen Bezug herstellen, benötigt man eine Datenstruktur, welche die Konten für jede Bank verwaltet. Diese Beziehung zwischen Bankexemplar und zugehörigen Konten ist analog zur Beziehung der statischen Attribute und Methoden einer Klasse und ihren Exemplar: Die Exemplare kennen die statischen Attribute und Methoden, umgekehrt besteht aber kein Bezug.
- ii.) Ein New-Operator wird nicht benötigt, weil wir uns innerhalb einer Methode der Klasse Bank befinden. Hier wird immer implizit `this` als Bezugsexemplar verwendet (wie Sie es beim Zugriff auf Attribute bereits so gewohnt sind, dass sie gar nicht daran denken). Wollten wir ein Konto bei einer anderen Bank erstellen, dann müsste man die Exemplarvariable wieder voranstellen:

```
public void transferAccountToBank(Account a, Bank b){
    Account target = b.new Account();
    target.deposit(a.withdrawAll());
}
```

- b.) Sie finden eine Jar-Datei mit der Lösung unter:

<http://www.inf.fu-berlin.de/lehre/WS05/ALP3/loesungen/U2.A1.jar>

Vorteile:

Flexibelste Variante mit der höchsten Wiederverwendbarkeit.

Wechsel der Bank für ein Konto ist möglich.

Nachteile:

Zusätzliche Verweise müssen zur zugehörigen Bank angebracht werden.

Aufgabe 2 (Ausnahmebehandlung in Java)

Es empfiehlt sich **Ausnahmen dann zu deklarieren**, wenn das Auftreten der Ausnahme zur "normalen" Funktion der Methode gehört, d.h. ein Ereignis repräsentiert, mit welchem der aufrufende Code unmittelbar umgehen soll.

Ein Beispiel:

Die Methode `int readFromUser() throws Exception` soll beim Benutzer die Eingabe einer Zahl erfragen. Bricht der Benutzer die Operation ab oder gibt eine ungültige Zahl ein, dann wird eine deklarierte Ausnahme ausgelöst, da man erwarten kann, dass der aufrufende Programmteil mit einer solchen Möglichkeit umgeht (z.B. indem er die Wiederholung der Eingabe veranlasst oder einen Standardwert übernimmt).

Außergewöhnliche Ereignisse (d.h. solche die eher dem Sprachgebrauch einer Ausnahme entsprechen) sollten als **Laufzeitausnahmen** deklariert werden, da in den meisten Fällen der aufrufende Code nicht in der Lage sein wird, sinnvoll mit dem Fehlerereignis umzugehen.

Ein Beispiel:

Die Methode `pop()` der Klasse `Stack` verwendet eine Ausnahme `"EmptyStackException"` um das außergewöhnliche Ereignis anzuzeigen, dass jemand bei der Programmierung einen solchen Fehler gemacht hat, dass ein Aufruf von `pop()` auf einen leeren `Stack` ausgeführt wird. Man vergleiche dies zum vorherigen Beispiel: Es hat sich anscheinend eine Inkonsistenz in die Daten eingeschlichen oder es hat sich ein Fehler in das Programm eingeschlichen. Auf solch ein Problem kann der Aufrufer in den meisten Fällen nicht reagieren, weshalb es nicht sinnvoll ist die Ausnahme anzuzeigen. Eine saubere Programmarchitektur wird aber an wohldefinierten Grenzen (und nicht in jeder Methode selbst) des Programms diese Laufzeitausnahmen abfangen, um sie z.B. zu protokollieren oder Daten des Benutzers zu speichern.

In den Worten des [Java Tutorials](#):

"If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception."

Aufgabe 3 (Schnittstellen in der Praxis):

Laden Sie sich von der Homepage der Veranstaltung den Quelltext, des Malprogramms `JHotDraw` <http://www.inf.fu-berlin.de/lehre/WS05/ALP3/material/-JHotDrawSrc.zip> herunter und stöbern Sie nach folgenden Sachen durch den Code:

- a.) Die Klasse `DrawApplication` (die Hauptklasse des Malprogrammes also) implementiert (unter anderem) `DrawingEditor`, `PaletteListener` und `VersionRequester`. Die Aufgaben sind damit allerdings noch nicht so leicht ersichtlich. Ein bisschen Nachforschung in den Schnittstellen gibt uns vielleicht folgende Hinweise: Als `DrawingEditor` übernimmt die Klasse die Verwaltung von Ansichten, Werkzeugen und Figuren, als `PaletteListener` ist sie zuständig für die Auswahl von Farben durch den Benutzer und als `VersionRequester` legt die Klasse fest, welche Version andere Teile des Programmes haben müssen, um an der Anwendung teilzunehmen.
- b.) Die Schnittstelle `Tool` wird einer großen Anzahl von Klassen implementiert, z.B. `PolygonTool`, `AreaTool`, `ActionTool`. Es gibt so viele verschiedene Implementierungen, weil diese die allgemeine Schnittstelle anders ausfüllen. Eine gemeinsame Schnittstelle zu haben, bedeutet nämlich nicht, dass es für den Benutzer der Schnittstelle keine Unterschiede zwischen den Implementierungen gibt (man denke z.B. an die Maximalgröße

des Ringpuffers gegenüber dem unbeschränkten Geflecht), sondern nur dass der Zugriff auf diesen Teil der Implementierung durch die Schnittstelle einheitlich geregelt ist.

- c.) Insgesamt ist die Software sehr sauber aufgebaut und verwendet zu einem großen Teil Schnittstellen anstelle von Klassen. Die Klasse als kapselnde Einheit der Implementierung sollte nicht als Parametertyp oder Variablentyp verwendet werden, sobald das System eine minimale Größe übersteigt. Durch die Verwendung von Schnittstellen wird sichergestellt, dass immer ein klarer Aufgabenbezug (auch Rolle genannt) für ein Objektexemplar hergestellt wird.

Aufgabe 4 (Typverträglichkeit und Innere Klassen)

```
7 interface Socket {
8     int send(Buffer out, int howMuch);
9
10    int receive(Buffer in, int maxReceive);
11 }
12
13 class NetworkAdapter {
14     private int sendBytes;
15
16     private TcpSocket s = new TcpSocket();
17
18     class TcpSocket implements Socket {
19         int send(Buffer buf, int howMuch) {
20             // .. Interessiert uns hier nicht
21             return howMuch;
22         }
23
24         int receive(Buffer in, int max) {
25             // .. Hier uninteressant
26             return max;
27         }
28     }
29
30     static class UdpSocket {
31         private boolean recordStatistics;
32
33         int send(Buffer out, int howMuch) {
34             // .. Interessiert uns hier nicht
35             if (recordStatistics)
36                 sendBytes += howMuch;
37             return howMuch;
38         }
39
40         int receive(Buffer in, int maxReceive) {
41             // .. Hier uninteressant
42             return maxReceive;
43         }
44     }
45 }
46
47 class Main {
48     public static void main() {
49         NetworkAdapter a = new NetworkAdapter();
50         NetworkAdapter.TcpSocket b = new NetworkAdapter.TcpSocket();
51         NetworkAdapter.UdpSocket c = new NetworkAdapter.UdpSocket();
52         Socket i = new Socket();
53         i = c;
54         i = a.s;
55         i = (new NetworkAdapter()).new TcpSocket();
56     }
57 }
```

18	TcpSocket soll Socket implementieren. Alle Vereinbarungen in einer Schnittstelle sind implizit public, auch wenn, wie oben, kein Modifizierer verwendet wird. Beim Implementieren darf die Sichtbarkeit nicht eingeschränkt werden. send und receive müssten hier also public sein.
36	Hier wird auf das nicht-statische Attribut sendBytes der Klasse NetworkAdapter zugegriffen. Die Klasse UdpSocket ist statisch, es besteht

	also kein impliziter Bezug zwischen einem <code>UdpSocket</code> -Objekt und einem umschließenden <code>NetworkAdapter</code> -Objekt, wie es bei einer Attributklasse der Fall wäre. Auf <code>sendBytes</code> kann hier also nicht zugegriffen werden.
50	<code>TcpSocket</code> ist eine Attributklasse, es besteht immer ein impliziter Bezug zu einem Objekt der umschließenden Klasse <code>NetworkAdapter</code> . Um diesen Bezug herzustellen, muss ein <code>TcpSocket</code> -Objekt in einem nicht-statischen Kontext der Klasse <code>NetworkAdapter</code> erzeugt werden, oder die spezielle Syntax " <code>a.new TcpSocket()</code> " muss verwendet werden.
52	<code>Socket</code> ist eine Schnittstelle. Eine Schnittstelle enthält nur die Signaturen von Methoden, jedoch keine Implementierung. Von Schnittstellen kann daher kein Exemplar erzeugt werden, es muss ein Objekt einer Klasse erzeugt werden, die diese Schnittstelle implementiert.
53	Der statische Typ von <code>c</code> ist <code>NetworkAdapter.UdpSocket</code> , <code>i</code> hat den statischen Typ <code>Socket</code> , es besteht keine Verträglichkeit. <code>NetworkAdapter.UdpSocket</code> implementiert zwar genau die Methoden, die in <code>Socket</code> definiert sind, es fehlt jedoch das " <code>implements Socket</code> ", so dass <code>NetworkAdapter.UdpSocket</code> keine Implementierung von <code>Socket</code> ist und der Inhalt von <code>c</code> nicht an die <code>Socket</code> -Variable zugewiesen werden kann.
54	Das Attribut <code>s</code> der Klasse <code>NetworkAdapter</code> ist hier in Klasse <code>Main</code> nicht sichtbar, da es als <code>private</code> deklariert ist.

Aufgabe X (Java Wiederholung) [optional]

Ein paar Testfälle sowie eine undokumentierte Musterlösung findet ihr unter <http://projects.mi.fu-berlin.de/w/bin/view/SS/JavaWiederholung>.

Java 1.5 Exkurs-Folge 1 - Var-Args bzw. Static Import

Keine Musterlösung. Wenn einer von euch seine Musterlösung einschickt, dann werde ich diese gerne auf der Webseite zur Verfügung stellen.