Was ist so toll an generischen Klassen und Schnittstellen?

1 Sammeln in den schlechten alten Zeiten (bis Java 1.4)

In den schlechten alten Zeiten konnte man nur `Object`-Objekte sammeln. Wenn man andere Objekte (z. B. `String`- oder `Integer`-Objekte) in eine Sammlung einfügte, „vergaß“ der Ausführer den genaueren Typ dieser Objekte und „merkte sich nur“, dass es sich um `Object`-Objekte handelte. Holte man solche Objekte aus einer Sammlung heraus, musste man sie in aller Regel mit gefährlichen Cast-Befehlen „in ihren genauen Typ umdeuten“, etwa so:

Beispiel-01: Eine alte, nicht-generische Sammlung

```
1   List alt = new ArrayList();
2   ...
3   alt.add(new String("Hallo!"));
4   alt.add(new Integer(17));
5   alt.add(new String("Wie geht's?"));
6   ...
7   String  s1 = (String)  alt.get(0);
8   Integer i1 = (Integer) alt.get(1);
9   Double  d1 = (Double)  alt.get(2); // ClassCastException
10  ...
```

Der Fehler in Zeile 9 (das `String`-Objekt `"Wie geht's?"` soll als `Double`-Objekt „gedeutet werden“) wurde früher nicht schon bei der Übergabe des Programms („zur Compilezeit“) erkannt, sondern erst bei der Ausführung des Programms („zur Laufzeit“). Eine Ausnahme der Klasse `ClassCastException` wurde geworfen. Das war schlecht in den alten Zeiten.

2 Sammeln in den guten neuen Zeiten (ab Java 5.0)

Beim Erzeugen einer Sammlung kann man den Typ der Objekte, die man darin sammeln will, angeben (festlegen, einschränken). Versucht man dann aus Versehen, Objekte anderer Typen in die Sammlung einzufügen, erkennt der Ausführer schon bei der Übergabe des Programms einen formalen Fehler, und besteht darauf, dass der Programmierer diesen Fehler noch vor der ersten Ausführung des Programms beseitigt.

Beispiel-01: Eine genauer getypte Sammlung

```
1   List<Number> neu = new ArrayList<Number>();
2   ...
3   neu.add(new Integer(123));
4   neu.add(new Double (3.5));
5   neu.add(new Integer(456));
6   ...
7   Number n1 = neu.get(0);
8   Number n2 = neu.get(1);
9   Double d2 = neu.get(2); // Formaler Fehler, Programm wird abgelehnt
10  ...
```

Zur Erinnerung: Die abstrakte Klasse `Number` ist die direkte Oberklasse der Hüllklassen `Byte`, `Short`, `Integer`, `Long`, `Float` und `Double`, der Klassen `BigInteger` und `BigDecimal` und weiterer Klassen.

Man beachte: Beim Zugreifen auf die Komponenten der Sammlung `neu` (in den Zeilen 7 bis 8) sind keine gefährlichen Cast-Befehle nötig, weil der Ausführer „weiss und sicher ist, dass alle Komponenten der Sammlung `neu` zum Typ `Number` gehören“.

Auch eine Sammlung des Typs `List<Object>` ist noch ein bisschen *typsicherer* als eine Sammlung des alten, *rohen Typs* `List`, wie das folgende Beispiel deutlich machen soll:

Beispiel-02: Alte und neue Sammlungen

```
11  List alt1 = new ArrayList();
12  List alt2 = new ArrayList();
13  ...
14  alt1 = alt2; // Erlaubt, unabhaengig von den Inhalten von alt1 und alt2
15
16  List<Object> neu1 = new ArrayList<Object>();
17  List<String> neu2 = new ArrayList<String>();
18  ...
19  neu1 = neu2; // Verboten, unabhaengig von den Inhalten von neu1 und neu2
```

3 Vergleichen in den schlechten alten Zeiten (bis Java 1.4)

In den schlechten alten Zeiten machte man `Apfel`-Objekte *vergleichbar*, indem man in der Klasse `Apfel` die Schnittstelle `Comparable` implementierte. Diese Schnittstelle verlangt, dass man eine Vergleichsmethode namens `compareTo` mit einem **`Object`**-Parameter vereinbart. Mit dieser Methode konnte man dann aber einen `Apfel` nicht nur mit einem `Apfel` vergleichen, sondern auch mit einem `Birne`- oder einem `UBoot`-Objekt, etwa so:

Beispiel-01: Eine alte, unspezifische `comparTo`-Methode

```
1   class Apfel implements Comparable {
2      private double saftMenge;
3      ...
4      public int compareTo(Object ob) {
5         if (ob instanceof Apfel) {
6            Apfel that = (Apfel) ob;
7            return (int) (this.saftMenge - that.saftMenge);
8         } else {
9            return ... // oder: throw ...
10        }
11     }
12  } // class Apfel
13  // ---------------------------------------------------------
14  class Birne ... {
15     ...
16  } // class Birne
17  // ---------------------------------------------------------
18  ...
19  Apfel a1 = new Apfel(...);
20  Birne b1 = new Birne(...);
21  ...
22  if (a1.compareTo(b1)) { ... } // Apfel mit Birne vergleichen?
23  ...
```

Dass in Zeile 22 ein `Apfel` mit einer `Birne` verglichen wird (was vermutlich unsinnig ist), wurde damals nicht schon bei der Übergabe des Programms („zur Compilezeit“) als Fehler erkannt, sondern frühestens bei der Ausführung des Programms (falls in Zeile 9 eine Ausnahme geworfen wird).

4 Vergleichen in den guten neuen Zeiten (ab Java 5.0)

Sollen `Apfel`-Objekte miteinander vergleichbar sein, so kann man in der Klasse `Apfel` jetzt die genauere Schnittstelle `Comparable<Apfel>` implementieren. Die verlangt, dass man eine Vergleichsmethode `compareTo` mit einem **`Apfel`**-Parameter (und nicht wie früher mit einem `Object`-Parameter) vereinbart, etwa so:

Beispiel-01: Eine neue, spezifische `compareTo`-Methode

```
1   class Apfel implements Comparable<Apfel> {
2      private double saftMenge;
3      ...
4      public int compareTo(Apfel that) {
5         return (int) (this.saftMenge - that.saftMenge);
6      }
7
8   } // class Apfel
9   // ---------------------------------------------------------
10  class Birne ... {
11     ...
12  } // class Birne
13  // ---------------------------------------------------------
14
15  ...
16  Apfel a1 = new Apfel(...);
17  Birne b1 = new Birne(...);
18  ...
19  if (a1.compareTo(b1)) { ... } // Formaler Fehler
```

In diesem Beispiel erkennt der Ausführer schon bei der Übergabe des Programms („zur Compilezeit"), dass der Vergleich in Zeile 19 einen formalen Fehler enthält, und besteht darauf, dass der Programmierer diesen Fehler noch vor der ersten Ausführung des Programms korrigiert.

Wenn man sich ein bisschen Mühe gibt, kann man auch im neuen, generischen Java sehr merkwürdige (vermutlich falsche) Vergleichsfunktionen erzeugen lassen, aber das ist deutlich schwieriger, als im alten, nicht-generischen Java.

Beispiel-02: Eine merkwürdige, vermutlich falsche Vergleichsfunktion

```
20  class Apfel implements Comparable<Birne> {
21     private double saftMenge;
22     ...
23     public int compareTo(Birne that) {
24        return ...;
25     }
26     ...
```

Dass man in diesem Beispiel `Apfel`-Objekte mit `Birne`-Objekten vergleichen darf, kann durchaus sinnvoll sein (z. B. dann, wenn auch jedes `Birne`-Objekt ein Attribut `saftMenge` besitzt). Vermutlich falsch ist, dass man hier ein `Apfel`-Objekt *nicht* mit einem anderen `Apfel`-Objekt (und auch nicht mit sich selbst) vergleichen darf.

Regel: Eine Klasse darf nur *eine* Instanz einer generischen Schnittstelle implementieren, aber nicht mehrere.

Beispiel-03: Mit einer Klasse (z. B. `Apfel`) *mehrere Instanzen derselben generischen Schnittstelle* (z. B. der Schnittstelle `Comparable`) zu implementieren ist *verboten*:

```
27  class Apfel implements Comparable<Apfel>, Comparable<Birne> { ... } // No!!
```

Diese Regel hängt mit dem grundsätzlichen Modell zusammen, nach dem generische Einheiten in Java implementiert werden. Dieses Modell hat große Vorteile (es ist vor allem viel *sparsamer* als das wichtigste Konkurrenzmodell, welches z. B. in C++ und Ada verwendet wird), ist aber auch mit ein paar *Einschränkungen* verbunden (siehe etwa Beispiel-03).

In aller Regel („in 98% aller Fälle") implementiert eine Klasse `Apfel` die Schnittstelle `Comparable<Apfel>` (und nicht die Schnittstelle `Comparable<Birne>`). Es kann aber auch („in 2 % aller Fälle") sinnvoll sein, in der Klasse `Apfel` eine „größere" Schnittstelle als `Comparable<Apfel>` zu implementieren, wie im folgenden Beispiel:

Beispiel-04: Die Klasse `java.sql.Time` aus der Java-Standardbibliothek

```
28  package java.sql;
29  import  java.util.Date;
30
31  public class Time extends Date implements Comparable<Date> {
32     ...
33  } // class Time
```

`Date` ist die direkte Oberklasse der Klasse `Time`. Somit ist jedes `Time`-Objekt auch ein `Date`-Objekt (aber einige `Date`-Objekte sind keine `Time`-Objekte). Aus der Klausel `implements Comparable<Date>` (in Zeile 31) folgt: `Time`-Objekte kann man nicht nur mit `Time`-Objekten vergleichen, sondern sogar mit allen `Date`-Objekten (auch mit denen, die *keine* `Time`-Objekte sind).

5 Drei generische Schnittstellen

Jede der drei Schnittstellen `SammelbarA`, `SammelbarB` und `SammelbarC` hat einen Typparameter `S`. Welche Typen man für diesen Parameter angeben darf wird aber (von A nach C) auf immer kompliziertere Weise eingeschränkt.

Beispiel-01: Drei generische Schnittstellen mit je einem Parameter `S` (wie „Schlüsseltyp“)

```
1   interface SammelbarA<S> {
2      S getSlue();
3   }
4
5   interface SammelbarB<S extends Comparable<S>> {
6      S getSlue();
7   }
8
9   interface SammelbarC<S extends Comparable<? super S>> {
10     S getSlue();
11  }
```

Implementiert man eine `SammelbarA`-Schnittstelle, darf man für `S` jeden Referenztyp `RT` (aber keinen primitiven Typ) einsetzen.

Implementiert man eine `SammelbarB`-Schnittstelle, darf man für `S` nur einen solchen Referenztyp `RT` einsetzen, dessen Objekte mit Objekten des Typs `RT` vergleichbar sind (im Sinne der Schnittstelle `Comparable`). Das schließt allerdings einen Typ wie `Time` aus (siehe Beispiel-03 im vorigen Abschnitt), weil dessen Objekte nicht nur mit `Time`-Objekten, sondern sogar mit allen `Date`-Objekten vergleichbar sind (`Time`-Objekte sind mit „zuvielen anderen Objekten“ vergleichbar).

Implementiert man eine `SammelbarC`-Schnittstelle, darf man für `S` nur einen solchen Referenztyp `RT` einsetzen, dessen Objekte mit allen Objekte eines Obertyps `ORT` von `RT` vergleichbar sind (im Sinne der Schnittstelle `Comparable`). Dabei gilt die Konvention: Jeder Referenztyp `RT` gilt als *Obertyp von sich selbst*. Bei der Schnittstelle `SammelbarC` darf man für `S` z. B. den Typ `String` einsetzen (weil die Klasse `String` die Schnittstelle `Comparable<String>` implementiert) oder den Typ `Time` (weil die Klasse `Time` die Schnittstelle `Comparable<Date>` implementiert und `Date` eine Oberklasse von `Time` ist).

Aufgabe-01: Welche der drei Schnittstellen schränkt ihren Typparameter `S` *am stärksten* ein?

Aufgabe-02: Vereinbaren Sie eine Klasse `KA` (bzw. `KB`, `KC`), die die Schnittstelle `SammelbarA` (`SammelbarB`, `SammelbarC`) implementiert.

Lösungen zur Aufgabe-02 findet man (möglichst nach der Entwicklung einer eigenen Lösung) in der Beispieldatei `SammelbarA_C.java`.