Erweiterungen in Java 1.5

Sven Eric Panitz
TFH Berlin
Version 24. Juni 2004

The source of this paper is an XML-file. The sources are processed by the XQuery processor *quip*, XSLT scripts and LaTeX in order to produce the different formats of the paper.

Inhaltsverzeichnis

1 Einführung

Mit Java 1.5 werden einige neue Konzepte in Java eingeführt, die viele Programmierer bisher schmerzlich vermisst haben. Obwohl die ursprünglichen Designer über diese neuen Konstrukte von Anfang an nachgedacht haben und sie auch gerne in die Sprache integriert hätten, schaffen diese Konstrukte es erst jetzt nach vielen Jahren in die Sprache Java. Hierfür kann man zwei große Gründe angeben:

- Beim Entwurf und der Entwicklung von Java waren die Entwickler bei Sun unter Zeitdruck.
- Die Sprache Java wurde in ihren programmiersprachlichen Konstrukten sehr konservativ entworfen. Die Syntax wurde von C übernommen. Die Sprache sollte möglichst wenige aber mächtige Eigenschaften haben und kein Sammelsurium verschiedenster Techniken sein.

Seitdem Java eine solch starke Bedeutung als Programmiersprache erlangt hat, gibt es einen definierten Prozess, wie neue Eigenschaften der Sprache hinzugefügt werden, den *Java community process (JPC)*. Hier können fundierte Verbesserungs- und Erweiterungsvorschläge gemacht werden. Wenn solche von genügend Leuten unterstützt werden, kann ein sogenannter *Java specification request (JSR)* aufgemacht werden. Hier wird eine Expertenrunde gegründet, die die Spezifikation und einen Prototypen für die Javaerweiterung erstellt. Die Spezifikation und prototypische Implementierung werden schließlich öffentlich zur Diskussion gestellt. Wenn es keine Einwände von irgendwelcher Seite mehr gibt, wird die neue Eigenschaft in eine der nächsten Javaversionen integriert.

Mit Java 1.5 findet das Ergebnis einer Vielzahl JSRs Einzug in die Sprache. Die Programmiersprache wird in einer Weise erweitert, wie es schon lange nicht mehr der Fall war. Den größten Einschnitt stellt sicherlich die Erweiterung des Typsystems auf generische Typen dar. Aber auch die anderen neuen Kosntrukte, die verbesserte `for`-Schleife, Aufzählungstypen, statisches Importieren und automatisches Boxen, sind keine esoterischen Eigenschaften, sondern werden das alltägliche Arbeiten mit Java beeinflussen. In den folgenden Abschnitten werfen wir einen Blick auf die neuen Javaeigenschaften im Einzelnen.

2 Generische Typen

Generische Typen wurden im JSR014 definiert. In der Expertengruppe des JSR014 war der Autor dieses Skripts zeitweilig als Stellvertreter der Software AG Mitglied. Die Software AG hatte mit der Programmiersprache Bolero bereits einen Compiler für generische Typen implementiert[Pan00]. Der Bolero Compiler generiert auch Java Byte Code. Von dem ersten Wunsch nach Generizität bis zur nun bald vorliegenden Javaversion 1.5 sind viele Jahre vergangen. Andere wichtige JSRs, die in Java 1.5 integriert werden, tragen bereits die Nummern 175 und 201. Hieran kann man schon erkennen, wie lange es gedauert hat, bis generische Typen in Java integriert wurden.

Interessierten Programmierern steht schon seit Mitte der 90er Jahre eine Javaerweiterung mit generischen Typen zur Verfügung. Unter den Namen Pizza [OW97] existiert eine Javaerweiterung, die nicht nur generische Typen, sondern auch algebraische Datentypen mit *pattern matching* und Funktionsobjekten zu Java hinzufügte. Unter den Namen *GJ* für *Generic Java* wurde eine allein auf generische Typen abgespeckte Version von *Pizza* publiziert. *GJ* ist tatsächlich der direkte Prototyp für Javas generische Typen. Die Expertenrunde des JSR014 hat *GJ* als Grundlage für die Spezifikation genommen und an den grundlegenden Prinzipien auch nichts mehr geändert.

2.1 Generische Klassen

Die Idee für generische Typen ist, eine Klasse zu schreiben, die für verschiedene Typen als Inhalt zu benutzen ist. Das geht bisher in Java, allerdings mit einem kleinen Nachteil. Versuchen wir einmal, in traditionellem Java eine Klasse zu schreiben, in der wir beliebige Objekte speichern können. Um beliebige Objekte speichern zu können, brauchen wir ein Feld, in dem Objekte jeden Typs gespeichert werden können. Dieses Feld muß daher den Typ `Object` erhalten:

OldBox.java

```java
class OldBox {
  Object contents;
```



```java
  OldBox(Object contents){this.contents=contents;}
}
```

Der Typ `Object` ist ein sehr unschöner Typ; denn mit ihm verlieren wir jegliche statische Typinformation. Wenn wir die Objekte der Klasse `OldBox` benutzen wollen, so verlieren wir sämtliche Typinformation über das in dieser Klasse abgespeicherte Objekt. Wenn wir auf das Feld `contents` zugreifen, so haben wir über das darin gespeicherte Objekte keine spezifische Information mehr. Um das Objekt weiter sinnvoll nutzen zu können, ist eine dynamische Typzusicherung durchzuführen:

UseOldBox.java

```java
class UseOldBox{
  public static void main(String [] _){
    OldBox b = new OldBox("hello");
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
    System.out.println(((String) s).toUpperCase());
  }
}
```

Wann immer wir mit dem Inhalt des Felds `contents` arbeiten wollen, ist die Typzusicherung während der Laufzeit durchzuführen. Die dynamische Typzusicherung kann zu einem Laufzeitfehler führen. So übersetzt das folgende Programm fehlerfrei, ergibt aber einen Laufzeitfehler:

UseOldBoxError.java

```java
class UseOldBoxError{
  public static void main(String [] _){
    OldBox b = new OldBox(new Integer(42));
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
  }
}
```

```
sep@linux:~/fh/java1.5/examples/src> javac UseOldBoxError.java
sep@linux:~/fh/java1.5/examples/src> java UseOldBoxError
Exception in thread "main" java.lang.ClassCastException
        at UseOldBoxError.main(UseOldBoxError.java:4)
sep@linux:~/fh/java1.5/examples/src>
```

Wie man sieht, verlieren wir Typsicherheit, sobald der Typ `Object` benutzt wird. Bestimmte Typfehler können nicht mehr statisch zur Übersetzungszeit, sondern erst dynamisch zur Laufzeit entdeckt werden.

Der Wunsch ist, Klassen zu schreiben, die genauso allgemein benutzbar sind wie die Klasse `OldBox` oben, aber trotzdem die statische Typsicherheit garantieren, indem sie nicht mit dem allgemeinen Typ `Object` arbeiten. Genau dieses leisten generische Klassen. Hierzu ersetzen wir in der obigen Klasse jedes Auftreten des Typs `Object` durch einen Variablennamen. Diese Variable ist eine Typvariable. Sie steht für einen beliebigen Typen. Dem Klassennamen fügen wir zusätzlich in der Klassendefinition in spitzen Klammern eingeschlossen hinzu, daß diese Klasse eine Typvariable benutzt. Wir erhalten somit aus der obigen Klasse `OldBox` folgende generische Klasse `Box`.

Box.java

```
class Box<elementType> {
  elementType contents;
  Box(elementType contents){this.contents=contents;}
}
```

Die Typvariable `elementType` ist als allquantifiziert zu verstehen. Für jeden Typ `elementType` können wir die Klasse `Box` benutzen. Man kann sich unsere Klasse `Box` analog zu einer realen Schachtel vorstellen: Beliebige Dinge können in die Schachtel gelegt werden. Betrachten wir dann allein die Schachtel von außen, können wir nicht mehr wissen, was für ein Objekt darin enthalten ist. Wenn wir viele Dinge in Schachteln packen, dann schreiben wir auf die Schachtel jeweils drauf, was in der entsprechenden Schachtel enthalten ist. Ansonsten würden wir schnell die Übersicht verlieren. Und genau das ermöglichen generische Klassen. Sobald wir ein konkretes Objekt der Klasse `Box` erzeugen wollen, müssen wir entscheiden, für welchen Inhalt wir eine Box brauchen. Dieses geschieht, indem in spitzen Klammern dem Klassennamen `Box` ein entsprechender Typ für den Inhalt angehängt wird. Wir erhalten dann z.B. den Typ `Box<String>`, um Strings in der Schachtel zu speichern, oder `Box<Integer>`, um Integerobjekte darin zu speichern:

UseBox.java

```
class UseBox{
  public static void main(String [] _){
    Box<String> b1 = new Box<String>("hello");
    String s = b1.contents;
    System.out.println(s.toUpperCase());
    System.out.println(b1.contents.toUpperCase());

    Box<Integer> b2 = new Box<Integer>(new Integer(42));

    System.out.println(b2.contents.intValue());
  }
}
```

Wie man im obigen Beispiel sieht, fallen jetzt die dynamischen Typzusicherungen weg. Die Variablen `b1` und `b2` sind jetzt nicht einfach vom Typ `Box`, sondern vom Typ `Box<String>` respektive `Box<Integer>`.

Da wir mit generischen Typen keine Typzusicherungen mehr vorzunehmen brauchen, bekommen wir auch keine dynamischen Typfehler mehr. Der Laufzeitfehler, wie wir ihn ohne die generische Box hatten, wird jetzt bereits zur Übersetzungszeit entdeckt. Hierzu betrachte man das analoge Programm:

```
class UseBoxError{
  public static void main(String [] _){
    Box<String> b = new Box<String>(new Integer(42));
    String s = b.contents;
    System.out.println(s.toUpperCase());
  }
}
```


Die Übersetzung dieses Programms führt jetzt bereits zu einen statischen Typfehler:

```
sep@linux:~/fh/java1.5/examples/src> javac UseBoxError.java
UseBoxError.java:3: cannot find symbol
symbol  : constructor Box(java.lang.Integer)
location: class Box<java.lang.String>
    Box<String> b = new Box<String>(new Integer(42));
                    ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

2.1.1 Vererbung

Generische Typen sind ein Konzept, das orthogonal zur Objektorientierung ist. Von generischen Klassen lassen sich in gewohnter Weise Unterklassen definieren. Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein. So können wir unsere einfache Schachtelklasse erweitern, so daß wir zwei Objekte speichern können:

Pair.java

```
class Pair<at,bt> extends Box<at>{
  Pair(at x,bt y){
    super(x);
    snd = y;
  }

  bt snd;

  public String toString(){
    return "("+contents+","+snd+")";
  }
}
```

Die Klasse `Pair` hat zwei Typvariablen. Instanzen von `Pair` müssen angeben von welchem Typ die beiden zu speichernden Objekte sein sollen.

UsePair.java

```
class UsePair{
  public static void main(String [] _){
    Pair<String,Integer> p
     = new Pair<String,Integer>("hallo",new  Integer(40));

    System.out.println(p);
    System.out.println(p.contents.toUpperCase());
    System.out.println(p.snd.intValue()+2);
  }
}
```

Wie man sieht kommen wir wieder ohne Typzusicherung aus. Es gibt keinen dynamischen Typcheck, der im Zweifelsfall zu einer Ausnahme führen könnte.

```
sep@linux:~/fh/java1.5/examples/classes> java UsePair
(hallo,40)
HALLO
42
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können auch eine Unterklasse bilden, indem wir mehrere Typvariablen zusammenfassen. Wenn wir uniforme Paare haben wollen, die zwei Objekte gleichen Typs speichern, können wir hierfür eine spezielle Paarklasse definieren.

UniPair.java

```java
class UniPair<at> extends Pair<at,at>{
  UniPair(at x,at y){super(x,y);}
  void swap(){
    final at z = snd;
    snd = contents;
    contents = z;
  }
}
```

Da beide gespeicherten Objekte jeweils vom gleichen Typ sind, konnten wir jetzt eine Methode schreiben, in der diese beiden Objekte ihren Platz tauschen. Wie man sieht, sind Typvariablen ebenso wie unsere bisherigen Typen zu benutzen. Sie können als Typ für lokale Variablen oder Parameter genutzt werden.

UseUniPair.java

```java
class UseUniPair{
  public static void main(String [] _){
    UniPair<String> p
     = new UniPair<String>("welt","hallo");

    System.out.println(p);
    p.swap();
    System.out.println(p);
  }
}
```

Wie man bei der Benutzung der uniformen Paare sieht, gibt man jetzt natürlich nur noch einen konkreten Typ für die Typvariablen an. Die Klasse `UniPair` hat ja nur eine Typvariable.

```
sep@linux:~/fh/java1.5/examples/classes> java UseUniPair
(welt,hallo)
(hallo,welt)
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können aber auch Unterklassen einer generischen Klasse bilden, die nicht mehr generisch ist. Dann leiten wir für eine ganz spezifische Instanz der Oberklasse ab. So läßt sich z.B. die Klasse `Box` zu einer Klasse erweitern, in der nur noch Stringobjekte verpackt werden können:

StringBox.java

```java
class StringBox extends Box<String>{
  StringBox(String x){super(x);}
}
```

Diese Klasse kann nun vollkommen ohne spitze Klammern benutzt werden:

UseStringBox.java

```java
class UseStringBox{
  public static void main(String [] _){
    StringBox b = new StringBox("hallo");
    System.out.println(b.contents.length());
  }
}
```

2.1.2 Einschränken der Typvariablen

Bisher standen in allen Beispielen die Typvariablen einer generischen Klasse für jeden beliebigen Objekttypen. Hier erlaubt Java uns, Einschränkungen zu machen. Es kann eingeschränkt werden, daß eine Typvariable nicht für alle Typen ersetzt werden darf, sondern nur für bestimmte Typen.

Versuchen wir einmal, eine Klasse zu schreiben, die auch wieder der Klasse `Box` entspricht, zusätzlich aber eine `set`-Methode hat und nur den neuen Wert in das entsprechende Objekt speichert, wenn es größer ist als das bereits gespeicherte Objekt. Hierzu müssen die zu speichernden Objekte in einer Ordnungsrelation vergleichbar sein, was in Java über die Implementierung der Schnittstelle `Comparable` ausgedrückt wird. Im herkömmlichen Java würden wir die Klasse wie folgt schreiben:

CollectMaxOld.java

```java
class CollectMaxOld{
  private Comparable value;

  CollectMaxOld(Comparable x){value=x;}

  void setValue(Comparable x){
    if (value.compareTo(x)<0) value=x;
  }

  Comparable getValue(){return value;}
}
```

Die Klasse `CollectMaxOld` ist in der Lage, beliebige Objekte, die die Schnittstelle `Comparable` implementieren, zu speichern. Wir haben wieder dasselbe Problem wie in der Klasse `OldBox`: Greifen wir auf das gespeicherte Objekt mit der Methode `getValue` erneut zu, wissen wir nicht mehr den genauen Typ dieses Objekts und müssen eventuell eine dynamische Typzusicherung durchführen, die zu Laufzeitfehlern führen kann.

Javas generische Typen können dieses Problem beheben. In gleicher Weise, wie wir die Klasse `Box` aus der Klasse `OldBox` erhalten haben, indem wir den allgemeinen Typ `Object` durch eine

Typvariable ersetzt haben, ersetzen wir jetzt den Typ `Comparable` durch eine Typvariable, geben aber zusätzlich an, daß diese Variable für alle Typen steht, die die Untertypen der Schnittstelle `Comparable` sind. Dieses wird durch eine zusätzliche `extends`-Klausel für die Typvariable angegeben. Wir erhalten somit eine generische Klasse `CollectMax`:

CollectMax.java

```
class CollectMax <elementType extends Comparable>{
  private elementType value;

  CollectMax(elementType x){value=x;}

  void setValue(elementType x){
    if (value.compareTo(x)<0) value=x;
  }

  elementType getValue(){return value;}
}
```

Für die Benutzung diese Klasse ist jetzt für jede konkrete Instanz der konkrete Typ des gespeicherten Objekts anzugeben. Die Methode `getValue` liefert als Rückgabetyp nicht ein allgemeines Objekt des Typs `Comparable`, sondern exakt ein Objekt des Instanztyps.

UseCollectMax.java

```
class UseCollectMax {
  public static void main(String [] _){
    CollectMax<String> cm = new CollectMax<String>("Brecht");
    cm.setValue("Calderon");
    cm.setValue("Horvath");
    cm.setValue("Shakespeare");
    cm.setValue("Schimmelpfennig");
    System.out.println(cm.getValue().toUpperCase());
  }
}
```

Wie man in der letzten Zeile sieht, entfällt wieder die dynamische Typzusicherung.

2.2 Generische Schnittstellen

Generische Typen erlauben es, den Typ `Object` in Typsignaturen zu eleminieren. Der Typ `Object` ist als schlecht anzusehen, denn er ist gleichbedeutend damit, daß keine Information über einen konkreten Typ während der Übersetzungszeit zur Verfügung steht. Im herkömmlichen Java ist in APIs von Bibliotheken der Typ `Object` allgegenwärtig. Sogar in der Klasse `Object` selbst begegnet er uns in Signaturen. Die Methode `equals` hat einen Parameter vom Typ `Object`, d.h. prinzipiell kann ein Objekt mit Objekten jeden beliebigen Typs verglichen werden. Zumeist will man aber nur gleiche Typen miteinander vergleichen. In diesem Abschnitt werden wir sehen, daß generische Typen es uns erlauben, allgemein eine Gleichheitsmethode zu definieren, in der nur Objekte gleichen Typs miteinander verglichen werden können. Hierzu werden wir eine generische Schnittstelle definieren.

Generische Typen erweitern sich ohne Umstände auf Schnittstellen. Im Vergleich zu generischen Klassen ist nichts Neues zu lernen. Syntax und Benutzung funktionieren auf die gleiche Weise.

2.2.1 Äpfel mit Birnen vergleichen

Um zu realisieren, daß nur noch Objekte gleichen Typs miteinander verglichen werden können, definieren wir eine Gleichheitsschnittstelle. In ihr wird eine Methode spezifiziert, die für die Gleichheit stehen soll. Die Schnittstelle ist generisch über den Typen, mit dem vergleichen werden soll.

EQ.java

```
interface EQ<otherType> {
  public boolean eq(otherType other);
}
```

Jetzt können wir für jede Klasse nicht nur bestimmen, daß sie die Gleichheit implementieren soll, sondern auch, mit welchen Typen Objekte unserer Klasse verglichen werden sollen. Schreiben wir hierzu eine Klasse `Apfel`. Die Klasse Apfel soll die Gleichheit auf sich selbst implementieren. Wir wollen nur Äpfel mit Äpfeln vergleichen können. Daher definieren wir in der `implements`-Klausel, daß wir `EQ<Apfel>` implementieren wollen. Dann müssen wir auch die Methode `eq` implementieren, und zwar mit dem Typ `Apfel` als Parametertyp:

Apfel.java

```
class Apfel implements EQ<Apfel>{
  String typ;

  Apfel(String typ){
    this.typ=typ;}

  public boolean eq(Apfel other){
    return this.typ.equals(other.typ);
  }
}
```

Jetzt können wir Äpfel mit Äpfeln vergleichen:

TestEQ.java

```
class TestEq{
  public static void main(String []_){
    Apfel a1 = new Apfel("Golden Delicious");
    Apfel a2 = new Apfel("Macintosh");
    System.out.println(a1.eq(a2));
    System.out.println(a1.eq(a1));
  }
}
```

Schreiben wir als nächstes eine Klasse die Birnen darstellen soll. Auch diese implementiere die Schnittstelle `EQ`, und zwar dieses Mal für Birnen:

Birne.java
```java
class Birne implements EQ<Birne>{
  String typ;

  Birne(String typ){
    this.typ=typ;}

  public boolean eq(Birne other){
    return this.typ.equals(other.typ);
  }
}
```

Während des statischen Typchecks wird überprüft, ob wir nur Äpfel mit Äpfeln und Birnen mit Birnen vergleichen. Der Versuch, Äpfel mit Birnen zu vergleichen, führt zu einem Typfehler:

```java
class TesteEqError{
  public static void main(String []_){
    Apfel a = new Apfel("Golden Delicious");
    Birne b = new Birne("williams");
    System.out.println(a.equals(b));
    System.out.println(a.eq(b));
  }
}
```

Wir bekommen die verständliche Fehlermeldung, daß die Gleichheit auf Äpfel nicht für einen Birnenparameter aufgerufen werden kann.

```
./TestEQError.java:6: eq(Apfel) in Apfel cannot be applied to (Birne)
    System.out.println(a.eq(b));
                        ^
1 error
```

Wahrscheinlich ist es jedem erfahrenden Javaprogrammierer schon einmal passiert, daß er zwei Objekte verglichen hat, die er gar nicht vergleichen wollte. Da der statische Typcheck solche Fehler nicht erkennen kann, denn die Methode `equals` läßt jedes Objekt als Parameter zu, sind solche Fehler mitunter schwer zu lokalisieren.

Der statische Typcheck stellt auch sicher, daß eine generische Schnittstelle mit der korrekten Signatur implementiert wird. Der Versuch, eine Birneklasse zu schreiben, die eine Gleichheit mit Äpfeln implementieren soll, dann aber die Methode `eq` mit dem Parametertyp `Birne` zu implementieren, führt ebenfalls zu einer Fehlermeldung:

```java
class BirneError implements EQ<Apfel>{
  String typ;

  BirneError(String typ){
    this.typ=typ;}

```



```java
  public boolean eq(Birne other){
    return this.typ.equals(other.typ);
  }
}
```

Wir bekommen folgende Fehlermeldung:

```
sep@linux:~/fh/java1.5/examples/src> javac BirneError.java
BirneError.java:1: BirneError is not abstract and does not override abstract method eq(Apfel) in EQ
class BirneError implements EQ<Apfel>{
^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

2.3 Kovarianz gegen Kontravarianz

Gegeben seien zwei Typen `A` und `B`. Der Typ `A` soll Untertyp des Typs `B` sein, also entweder ist `A` eine Unterklasse der Klasse `B` oder `A` implementiert die Schnittstelle `B` oder die Schnittstelle `A` erweitert die Schnittstelle `B`. Für diese Subtyprelation schreiben wir das Relationssymbol $\sqsubseteq$. Es gelte also $A \sqsubseteq B$. Gilt damit auch für einen generischen Typ `C`: `C<A>`$\sqsubseteq$`C<B>`?

Man mag geneigt sein, zu sagen ja. Probieren wir dieses einmal aus:

```java
class Kontra{
  public static void main(String []_){
    Box<Object> b  = new Box<String>("hello");
  }
}
```

Der Javaübersetzer weist dieses Programm zurück:

```
sep@linux:~/fh/java1.5/examples/src> javac Kontra.java
Kontra.java:4: incompatible types
found   : Box<java.lang.String>
required: Box<java.lang.Object>
    Box<Object> b  = new Box<String>("hello");
                     ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

Eine `Box<String>` ist keine `Box<Object>`. Der Grund für diese Entwurfsentscheidung liegt darin, daß bestimmte Laufzeitfehler vermieden werden sollen. Betrachtet man ein Objekt des Typs `Box<String>` über eine Referenz des Typs `Box<Object>`, dann können in dem Feld `contents` beliebige Objekte gespeichert werden. Die Referenz über den Typ `Box<String>` geht aber davon aus, daß in `contents` nur Stringobjekte gespeichert werden.

Man vergegenwärtige sich nochmals, daß Reihungen in Java sich hier anders verhalten. Bei Reihungen ist die entsprechende Zuweisung erlaubt. Eine Reihung von Stringobjekten darf einer Reihung beliebiger Objekte zugewiesen werden. Dann kann es bei der Benutzung der Reihung von Objekten zu einen Laufzeitfehler kommen.

Ko.java

```
class Ko{
  public static void main(String []_){
    String [] x = {"hello"};
    Object [] b  = x;
    b[0]=new Integer(42);
    x[0].toUpperCase();
  }
}
```

Das obige Programm führt zu folgendem Laufzeitfehler:

```
sep@linux:~/fh/java1.5/examples/classes> java Ko
Exception in thread "main" java.lang.ArrayStoreException
        at Ko.main(Ko.java:5)
sep@linux:~/fh/java1.5/examples/classes>
```

Für generische Typen wurde ein solcher Fehler durch die Strenge des statischen Typchecks bereits ausgeschlossen.

2.4 Sammlungsklassen

Die Paradeanwendung für generische Typen sind natürlich Sammlungsklassen, also die Klassen für Listen und Mengen, wie sie im Paket `java.util` definiert sind. Mit der Version 1.5 von Java finden sich generische Versionen der bekannten Sammlungsklassen. Jetzt kann man angeben, was für einen Typ die Elemente einer Sammlung genau haben sollen.

ListTest.java

```
import java.util.*;
import java.util.List;
class ListTest{
  public static void main(String [] _){
    List<String> xs = new ArrayList<String>();
    xs.add("Schimmelpfennig");
    xs.add("Shakespeare");
    xs.add("Horvath");
    xs.add("Brecht");
    String x2 = xs.get(1);
    System.out.println(xs);
  }
}
```

Aus Kompatibilitätsgründen mit bestehendem Code können generische Klassen auch weiterhin ohne konkrete Angabe des Typparameters benutzt werden. Während der Übersetzung wird in diesen Fällen eine Warnung ausgegeben.

WarnList.java

```
import java.util.*;
import java.util.List;
```



```
class WarnTest{
  public static void main(String [] _){
    List xs = new ArrayList<String>();
    xs.add("Schimmelpfennig");
    xs.add("Shakespeare");
    xs.add("Horvath");
    xs.add("Brecht");
    String x2 = (String)xs.get(1);
    System.out.println(xs);
  }
}
```

Obiges Programm übersetzt mit folgender Warnung:

```
sep@linux:~/fh/java1.5/examples/src> javac WarnList.java
Note: WarnList.java uses unchecked or unsafe operations.
Note: Recompile with -warnunchecked for details.
sep@linux:~/fh/java1.5/examples/src>
```

2.5 Generische Methoden

Bisher haben wir generische Typen für Klassen und Schnittstellen betrachtet. Generische Typen sind aber nicht an einen objektorientierten Kontext gebunden, sondern basieren ganz im Gegenteil auf dem Milner-Typsystem, das funktionale Sprachen, die nicht objektorientiert sind, benutzen. In Java verläßt man den objektorientierten Kontext in statischen Methoden. Statische Methoden sind nicht an ein Objekt gebunden. Auch statische Methoden lassen sich generisch in Java definieren. Hierzu ist vor der Methodensignatur in spitzen Klammern eine Liste der für die statische Methode benutzten Typvariablen anzugeben.

Eine sehr einfache statische generische Methode ist eine *trace*-Methode, die ein beliebiges Objekt erhält, dieses Objekt auf der Konsole ausgibt und als Ergebnis genau das erhaltene Objekt unverändert wieder zurückgibt. Diese Methode `trace` hat für alle Typen den gleichen Code und kann daher entsprechend generisch geschrieben werden:

Trace.java

```
class Trace {

  static <elementType>  elementType trace(elementType x){
    System.out.println(x);
    return x;
  }

  public static void main(String [] _){
    String x = trace ((trace ("hallo")
                     +trace( " welt")).toUpperCase());

    Integer y = trace (new Integer(40+2));
  }
}
```

In diesem Beispiel ist zu erkennen, daß der Typchecker eine kleine Typinferenz vornimmt. Bei der Anwendung der Methode `trace` ist nicht anzugeben, mit welchen Typ die Typvariable `elementType` zu instanziieren ist. Diese Information inferriert der Typchecker automatisch aus dem Typ des Arguments.

3 Iteration

Typischer Weise wird in einem Programm über die Elemente eines Sammlungstyp iteriert oder über alle Elemente einer Reihung. Hierzu kennt Java verschiedene Schleifenkonstrukte. Leider kannte Java bisher kein eigenes Schleifenkonstrukt, das bequem eine Iteration über die Elemente einer Sammlung ausdrücken konnte. Die Schleifensteuerung mußte bisher immer explizit ausprogrammiert werden. Hierbei können Programmierfehler auftreten, die insbesondere dazu führen können, daß eine Schleife nicht terminiert. Ein Schleifenkonstrukt, das garantiert terminiert, kannte Java bisher nicht.

Beispiel:
In diesen Beispiel finden sich die zwei wahrscheinlich am häufigsten programmierten Schleifentypen. Einmal iterieren wir über alle Elemente einer Reihung und einmal iterieren wir mittels eines Iteratorobjekts über alle Elemente eines Sammlungsobjekts:

OldIteration.java

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

class OldIteration{

  public static void main(String [] _){
    String [] ar
      = {"Brecht","Horvath","Shakespeare","Schimmelpfennig"};
    List xs = new ArrayList();

    for (int i= 0;i<ar.length;i++){
      final String s = ar[i];
      xs.add(s);
    }

    for (Iterator it=xs.iterator();it.hasNext();){
      final String s = (String)it.next();
      System.out.println(s.toUpperCase());
    }
  }
}
```

Die Codemenge zur Schleifensteuerung ist gewaltig und übersteigt hier sogar die eigentliche Anwendungslogik.

Mit Java 1.5 gibt es endlich eine Möglichkeit, zu sagen, mache für alle Elemente im nachfolgenden Sammlungsobjekt etwas. Eine solche Syntax ist jetzt in Java integriert. Sie hat die Form:

`for` (*Type identifier* : *expr*){*body*}

Zu lesen ist dieses Kosntrukt als: für jedes *identifier* des Typs *Type* in *expr* führe *body* aus.[1]

Beispiel:
Damit lassen sich jetzt die Iterationen der letzten beiden Schleifen wesentlich eleganter ausdrücken.

NewIteration.java

```
import java.util.List;
import java.util.ArrayList;

class NewIteration{

  public static void main(String [] _){
    String [] ar
     = {"Brecht","Horvath","Shakespeare","Schimmelpfennig"};
    List<String> xs = new ArrayList<String>();

    for (String s:ar) xs.add(s);
    for (String s:xs) System.out.println(s.toUpperCase());
  }
}
```

Der gesamte Code zur Schleifensteuerung ist entfallen. Zusätzlich ist garantiert, daß für endliche Sammlungsiteratoren auch die Schleife terminiert.

Wie man sieht, ergänzen sich generische Typen und die neue `for`-Schleife.

3.1 Die neuen Schnittstellen Iterable

Beispiel:

ReaderIterator.java

```
package sep.util.io;

import java.io.Reader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Iterator;


```

[1] Ein noch sprechenderes Konstrukt wäre gewesen, wenn man statt des Doppelpunkts das Schlüsselwort `in` benutzt hätte. Aus Aufwärtskompatibilitätsgründen wird jedoch darauf verzichtet, neue Schlüsselwörter in Java einzuführen.

```
public class ReaderIterator
              implements Iterable<Character>
                        ,Iterator<Character>{
  private Reader reader;
  private int n;
  public ReaderIterator(Reader r){
    reader=new BufferedReader(r);
    try{n=reader.read();
    }catch(IOException _){n=-1;}
  }
  public Character next(){
    Character result = new Character((char)n);
    try{n=reader.read();
    }catch(IOException _){n=-1;}
    return result;
  }

  public boolean hasNext(){
    return n!=-1;
  }

  public void remove(){
    throw new UnsupportedOperationException();
  }

  public Iterator<Character> iterator(){return this;}
}
```

TestReaderIterator.java

```
import sep.util.io.ReaderIterator;
import java.io.FileReader;

class TestReaderIterator {
  public static void main(String [] args) throws Exception{
      Iterable<Character> it
        =new ReaderIterator(new FileReader(args[0]));
      for (Character c:it){
      System.out.print(c);
    }
  }
}
```

Beispiel:
Ein abschließendes kleines Beispiel für generische Sammlungsklassen und die neue `for`-Schleife. Die folgende Klasse stellt Methoden zur Verfügung, um einen String in eine Liste von Wörtern zu spalten und umgekehrt aus einer Liste von Wörtern wieder einen String zu bilden:

TextUtils.java

```
import java.util.*;
import java.util.List;
```



```

class TextUtils {

  static List<String> words (String s){
    final List<String> result = new ArrayList<String>();

    StringBuffer currentWord = new StringBuffer();

    for (char c:s.toCharArray()){
      if (Character.isWhitespace(c)){
        final String newWord = currentWord.toString().trim();
        if(newWord.length()>0){
          result.add(newWord);
          currentWord=new StringBuffer();
        }
      }else{currentWord.append(c);}
    }
    return result;
  }

  static String unwords(List<String> xs){
    StringBuffer result=new StringBuffer();
    for (String x:xs) result.append(" "+x);
    return result.toString().trim();
  }

  public static void main(String []_){
    List<String> xs = words("  the   world  is my Oyster  ");

    for (String x:xs) System.out.println(x);

    System.out.println(unwords(xs));
  }
}
```

4 Automatisches Boxen

Javas Typsystem ist etwas zweigeteilt. Es gibt Objekttypen und primitive Typen. Die Daten der primitiven Typen stellen keine Objekte dar. Für jeden primitiven Typen gibt es allerdings im Paket `java.lang` eine Klasse, die es erlaubt, Daten eines primitiven Typs als Objekt zu speichern. Dieses sind die Klassen `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` und `Character`. Objekte dieser Klassen sind nicht modifizierbar. Einmal ein Integer-Objekt mit einer bestimmten Zahl erzeugt, läßt sich die in diesem Objekt erzeugte Zahl nicht mehr verändern.

Wollte man in bisherigen Klassen ein Datum eines primitiven Typen in einer Variablen speichern, die nur Objekte speichern kann, so mußte man es in einem Objekt der entsprechenden Klasse kapseln. Man spricht von *boxing*. Es kam zu Konstruktoraufrufen dieser

Klassen. Sollte später mit Operatoren auf den Zahlen, die durch solche gekapselten Objekte ausgedrückt wurden, gerechnet werden, so war der primitive Wer mit einem Methodenaufruf aus dem Objekt wieder zu extrahieren, dem sogenannten *unboxing*. Es kam zu Aufrufen von Methoden wie `intValue()` im Code.

Beispiel:
In diesem Beispiel sieht man das manuelle Verpacken und Auspacken primitiver Daten.

ManualBoxing.java
```
package name.panitz.boxing;
public class ManualBoxing{
  public static void main(String [] _){
    int i1 = 42;
    Object o = new Integer(i1);
    System.out.println(o);
    Integer i2 = new Integer(17);
    Integer i3 = new Integer(4);
    int i4 = 21;
    System.out.println((i2.intValue()+i3.intValue())*i4);
  }
}
```

Mit Java 1.5 können die primitiven Typen mit ihren entsprechenden Klassen synonym verwendet werden. Nach außen hin werden die primitiven Typen auch zu Objekttypen. Der Übersetzer nimmt fügt die notwendigen *boxing*- und *unboxing*-Operationen vor.

Beispiel:
Jetzt das vorherige kleine Programm ohne explizite *boxing*- und *unboxing*-Aufrufe.

AutomaticBoxing.java
```
package name.panitz.boxing;
public class AutomaticBoxing{
  public static void main(String [] _){
    int i1 = 42;
    Object o = i1;
    System.out.println(o);
    Integer i2 = 17;
    Integer i3 = 4;
    int i4 = 21;
    System.out.println((i2+i3)*i4);
  }
}
```

5 Aufzählungstypen

Häufig möchte man in einem Programm mit einer endlichen Menge von Werten rechnen. Java bot bis Version 1.4 kein ausgezeichnetes Konstrukt an, um dieses auszudrücken. Man war gezwungen in diesem Fall sich des Typs `int` zu bedienen und statische Konstanten dieses

Typs zu deklarieren. Dieses sieht man auch häufig in Bibliotheken von Java umgesetzt. Etwas mächtiger und weniger primitiv ist, eine Klasse zu schreiben, in der es entsprechende Konstanten dieser Klasse gibt, die durchnummeriert sind. Dieses ist ein Programmiermuster, das Aufzählungsmuster.

Mit Java 1.5 ist ein expliziter Aufzählungstyp in Java integriert worden. Syntaktisch erscheint dieser wie eine Klasse, die statt des Schlüsselworts `class` das Schlüsselwort `enum` hat. Es folgt als erstes in diesen Aufzählungsklassen die Aufzählung der einzelnen Werte.

Beispiel:
Ein erster Aufzählungstyp für die Wochentage.

Wochentage.java
```
package name.panitz.enums;
public enum Wochentage {
  montag,dienstag,mittwoch,donnerstag
 ,freitag,sonnabend,sonntag;
}
```

Auch dieses neue Konstrukt wird von Javaübersetzer in eine herkömmlige Javaklasse übersetzt. Wir können uns davon überzeugen, indem wir uns einmal den Inhalt der erzeugten Klassendatei mit `javap` wieder anzeigen lassen:

```
sep@linux:fh/> javap name.panitz.enums.Wochentage
Compiled from "Wochentage.java"
public class name.panitz.enums.Wochentage extends java.lang.Enum{
    public static final name.panitz.enums.Wochentage montag;
    public static final name.panitz.enums.Wochentage dienstag;
    public static final name.panitz.enums.Wochentage mittwoch;
    public static final name.panitz.enums.Wochentage donnerstag;
    public static final name.panitz.enums.Wochentage freitag;
    public static final name.panitz.enums.Wochentage sonnabend;
    public static final name.panitz.enums.Wochentage sonntag;
    public static final name.panitz.enums.Wochentage[] values();
    public static name.panitz.enums.Wochentage valueOf(java.lang.String);
    public name.panitz.enums.Wochentage(java.lang.String, int);
    public int compareTo(java.lang.Enum);
    public int compareTo(java.lang.Object);
    static {};
}
```

Eine der schönen Eigenschaften der Aufzählungstypen ist, daß sie in einer `switch`-Anweisung benutzt werden können.

Beispiel:
Wir fügen der Aufzählungsklasse eine Methode zu, um zu testen ob der Tag ein Werktag ist. Hierbei läßt sich eine `switch`-Anweisung benutzen.

Tage.java
```
package name.panitz.enums;
public enum Tage {
  montag,dienstag,mittwoch,donnerstag
 ,freitag,sonnabend,sonntag;

  public boolean isWerktag(){
```

```
    switch (this){
      case sonntag    :
      case sonnabend  :return false;
      default         :return true;
    }
  }

  public static void main(String [] _){
    Tage tag = freitag;
    System.out.println(tag);
    System.out.println(tag.ordinal());
    System.out.println(tag.isWerktag());
    System.out.println(sonntag.isWerktag());
  }
}
```

Das Programm gibt die erwartete Ausgabe:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.Tage
freitag
4
true
false
sep@linux:~/fh/java1.5/examples>
```

Eine angenehme Eigenschaft der Aufzählungsklassen ist, daß sie in einer Reihung alle Werte der Aufzählung enthalten, so daß mit der neuen `for`-Schleife bequem über diese iteriert werden kann.

Beispiel:
Wir iterieren in diesem Beispiel einmal über alle Wochentage.

IterTage.java

```
package name.panitz.enums;
public class IterTage {
  public static void main(String [] _){
    for (Tage tag:Tage.values())
      System.out.println(tag.ordinal()+": "+tag);
  }
}
```

Die erwarttete Ausgabe ist:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.IterTage
0: montag
1: dienstag
2: mittwoch
3: donnerstag
4: freitag
5: sonnabend
6: sonntag
sep@linux:~/fh/java1.5/examples>
```


Schließlich kann man den einzelnen Konstanten einer Aufzählung noch Werte übergeben.

Beispiel:
Wir schreiben eine Aufzählung für die Euroscheine. Jeder Scheinkonstante wird noch eine ganze Zahl mit übergeben. Es muß hierfür ein allgemeiner Konstruktor geschrieben werden, der diesen Parameter übergeben bekommt.

Euroschein.java

```
package name.panitz.enums;
public enum Euroschein {
   fünf(5),zehn(10),zwanzig(20),fünfzig(50),hundert(100)
  ,zweihundert(200),tausend(1000);
  private int value;
  public Euroschein(int v){value=v;}
  public int value(){return value();}

  public static void main(String [] _){
    for (Euroschein schein:Euroschein.values())
      System.out.println
        (schein.ordinal()+": "+schein+" -> "+schein.value);
  }
}
```

Das Programm hat die folgende Ausgabe:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.Euroschein
0: fünf -> 5
1: zehn -> 10
2: zwanzig -> 20
3: fünfzig -> 50
4: hundert -> 100
5: zweihundert -> 200
6: tausend -> 1000
sep@linux:~/fh/java1.5/examples>
```

6 Statische Imports

Statische Eigenschaften einer Klasse werden in Java dadurh angesprochen, daß dem Namen der Klasse mit Punkt getrennt die gewünschte Eigenschaft folgt. Werden in einer Klasse sehr oft statische Eigenschaften einer anderen Klasse benutzt, so ist der Code mit deren Klassennamen durchsetzt. Die Javaentwickler haben mit Java 1.5 ein Einsehen. Man kann jetzt für eine Klasse alle ihre statischen Eigenschaften importieren, so daß diese unqualifiziert benutzt werden kann. Die `import`-Anweisung sieht aus wie ein gewohntes Paktimport, nur daß das Schlüsselwort `static` eingefügt ist und erst dem klassennamen der Stern folgt, der in diesen Fall für alle statischen Eigenschaften steht.

Beispiel:
Wir schreiben eine Hilfsklasse zum Arbeiten mit Strings, in der wir eine Methode zum umdrehen eines Strings vorsehen:

StringUtil.java
```
package name.panitz.staticImport;
public class StringUtil {
  static public String reverse(String arg) {
    StringBuffer result = new StringBuffer();
    for (char c:arg.toCharArray()) result.insert(0,c);
    return result.toString();
  }
}
```

Die Methode `reverse` wollen wir in einer anderen Klasse benutzen. Importieren wir die statischen Eigenschaften von `StringUtil`, so können wir auf die Qualifizierung des Namens der Methode `reverse` verzichten:

UseStringUtil.java
```
package name.panitz.staticImport;
import static name.panitz.staticImport.StringUtil.*;
public class UseStringUtil {
  static public void main(String [] args) {
    for (String arg:args)
     System.out.println(reverse(arg));
  }
}
```

Die Ausgabe dieses programms:

```
sep@linux:fh> java -classpath classes/ name.panitz.staticImport.UseStringUtil hallo welt
ollah
tlew
sep@linux:~/fh/java1.5/examples>
```

7 Variable Parameteranzahl

Als zusätzliches kleines Gimmik ist in Java 1.5 eingebaut worden, daß Methoden mit einer variablen Parameteranzahl definiert werden können. Dieses wird durch drei Punkte nach dem Parametertyp in der Signatur gekennzeichnet. Damit wird angegeben, daß eine bieliebige Anzahl dieser Parameter bei einem Methodenaufruf geben kann.

Beispiel:
Es läßt sich so eine Methode schreiben, die mit beliebig vielen Stringparametern aufgerufen werden kann.

VarParams.java
```
package name.panitz.java15;

public class VarParams{
  static public String append(String... args){
    String result="";
    for (String a:args)
      result=result+a;
```



```
    return result;
  }

  public static void main(String [] _){
    System.out.println(append("hello"," ","world"));
  }
}
```

Die Methode `append` konkateniert endlich viele `String`-Objekte.

Wie schon für Aufzählungen können wir auch einmal schauen, was für Code der Javakompilierer für solche Methoden erzeugt.

```
sep@linux:~/fh/java1.5/examples> javap -classpath classes/ name.panitz.java15.VarParams
Compiled from "VarParams.java"
public class name.panitz.java15.VarParams extends java.lang.Object{
    public name.panitz.java15.VarParams();
    public static java.lang.String append(java.lang.String[]);
    public static void main(java.lang.String[]);
}

sep@linux:~/fh/java1.5/examples>
```

Wie man sieht wird für die variable Parameteranzahl eine Reihung erzeugt. Der Javakompilierer sorgt bei Aufrufen der Methode dafür, daß die entsprechenden Parameter in eine Reihung verpackt werden. Daher können wir mit dem Parameter wie mit einer Reihung arbeiten.

8 Ein paar Beispielklassen

In Java gibt es keinen Funktionstyp als Typ erster Klasse. Funktionen können nur als Methoden eines Objektes als Parameter weitergereicht werden. Mit generischen Typen können wir eine Schnittstelle schreiben, die ausdrücken soll, daß das implementieren Objekt eine einstellige Funktion darstellt.

UnaryFunction.java
```
package name.panitz.crempel.util;

public interface UnaryFunction<arg,result>{
  public result eval(arg a);
}
```

Ebenso können wir eine Schnittstelle für konstante Methoden vorsehen:

Closure.java
```
package name.panitz.crempel.util;

public interface Closure<result>{
  public result eval();
}
```

Wir haben die generischen Typen eingeführt anhand der einfachen Klasse Box. Häufig benötigt man für die Werterückgabe von Methoden kurzzeitig eine Tupelklasse. Im folgenden sind ein paar solche Tupelklasse generisch realisiert:

Tuple1.java
```java
package name.panitz.crempel.util;

public class Tuple1<t1> {
  public t1 e1;
  public Tuple1(t1 a1){e1=a1;}
  String parenthes(Object o){return "("+o+")";}
  String simpleToString(){return e1.toString();}
  public String toString(){return parenthes(simpleToString());}
  public boolean equals(Object other){
    if (! (other instanceof Tuple1)) return false;
    return e1.equals(((Tuple1)other).e1);
  }
}
```

Tuple2.java
```java
package name.panitz.crempel.util;

public class Tuple2<t1,t2> extends Tuple1<t1>{
  public t2 e2;
  public Tuple2(t1 a1,t2 a2){super(a1);e2=a2;}
  String simpleToString(){
    return super.simpleToString()+","+e2.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple2)) return false;
    return super.equals(other)&& e2.equals(((Tuple2)other).e2);
  }
}
```

Tuple3.java
```java
package name.panitz.crempel.util;

public class Tuple3<t1,t2,t3> extends Tuple2<t1,t2>{
  public t3 e3;
  public Tuple3(t1 a1,t2 a2,t3 a3){super(a1,a2);e3=a3;}
  String simpleToString(){
    return super.simpleToString()+","+e3.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple3)) return false;
    return super.equals(other)&& e3.equals(((Tuple3)other).e3);
  }
}
```

Tuple4.java
```java
package name.panitz.crempel.util;

public class Tuple4<t1,t2,t3,t4> extends Tuple3<t1,t2,t3>{
```



```java
  public t4 e4;
  public Tuple4(t1 a1,t2 a2,t3 a3,t4 a4){super(a1,a2,a3);e4=a4;}
  String simpleToString(){
    return super.simpleToString()+","+e4.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple4)) return false;
    return super.equals(other)&& e4.equals(((Tuple4)other).e4);
  }
}
```

Zum Iterieren über einen Zahlenbereich können wir eine entsprechende Klasse vorsehen.

FromTo.java
```java
package name.panitz.crempel.util;

import java.util.Iterator;

public class FromTo implements Iterable<Integer>,Iterator<Integer>{
  private final int to;
  private int from;
  public FromTo(int f,int t){to=t;from=f;}
  public boolean hasNext(){return from<=to;}
  public Integer next(){int result = from;from=from+1;return result;}
  public Iterator<Integer> iterator(){return this;}
  public void remove(){new UnsupportedOperationException();}
}
```

Statt mit **null** zu Arbeiten, kann man einen Typen vorsehen, der entweder gerade einen Wert oder das Nichtvorhandensein eines Wertes darstellt. In funktionalen Sprachen gibt es hierzu den entsprechenden generischen algebraischen Datentypen:

HsMaybe.hs
```haskell
data Maybe a = Nothing|Just a
```

Dieses läßt sich durch eine generische Schnittstelle und zwei generische implementierende Klassen in Java ausdrücken.

Maybe.java
```java
package name.panitz.crempel.util;
public interface Maybe<a> {}
```

Nothing.java
```java
package name.panitz.crempel.util;
public class Nothing<a> implements Maybe<a>{

  public String toString(){return "Nothing("+")";}
  public boolean equals(Object other){
    return (other instanceof Nothing);
  }
}
```

Just.java
```java
package name.panitz.crempel.util;

public class Just<a> implements Maybe<a>{
  private a just;

  public Just(a just){this.just = just;}
  public a getJust(){return just;}

  public String toString(){return "Just("+just+")";}
  public boolean equals(Object other){
    if (!(other instanceof Just)) return false;
    final Just o= (Just) other;
    return just.equals(o.just);
  }
}
```

Die jetzt folgende Klasse möge der leser bitte ignorieren. Sie ist aus rein technischen internen Gründen an dieser Stelle.

CrempelTool.java
```java
package name.panitz.crempel.tool;
public interface CrempelTool{String getDescription();void startUp();}
```

9 Aufgaben

Aufgabe 1

a) Schreiben Sie eine Schnittstelle `Smaller`, in der es eine Methode `le` mit zwei generischen Parametern gleichen Typs gibt.

Lösung

Smaller.java
```java
package name.panitz.aufgaben;
public interface Smaller<at> {
  public boolean le(at x1,at x2);
}
```

b) Schreiben Sie eine Klasse, die `Smaller<Integer>`, so implementiert, daß gerade Zahlen kleiner als ungerade Zahlen sind.

Lösung

SmallEven.java
```java
package name.panitz.aufgaben;
public class SmallEven implements Smaller<Integer> {
  public boolean le(Integer x1,Integer x2){
    if (x1%2 != x2%2){return x1%2==0;};
```



```java
    return x1<=x2;
  }
}
```

c) Schreiben Sie eine Klasse, die `Smaller<String>`, so implementiert, daß kurze Zeichenketten kleiner als lange sind.

Lösung

SmallShort.java
```java
package name.panitz.aufgaben;
public class SmallShort implements Smaller<String> {
  public boolean le(String x1,String x2){
    return x1.length()<=x2.length();
  }
}
```

d) Schreiben Sie den Methodenkopf für eine statische generische Methode `bubbleSort` zum Sortieren von Reihungen, die zwei Parameter hat: eine zu sortierende Reihung und ein passendes Objekt des Typs `Smaller`.

Lösung

Hier mit dem Methodenrumpf, der aber nicht verlangt war:

Bubble.java
```java
package name.panitz.aufgaben;
public class Bubble {
  public static <at> void bubbleSort(at[] ar,Smaller<at> rel){
    boolean toBubble = true;
    int end=ar.length;
    while (toBubble){
      toBubble = bubble(ar,rel,end);
      end=end-1;
    }
  }

  static <at> boolean bubble(at[] ar,Smaller<at> rel,int end){
    boolean result = false;
    for (int i=0;i<end;i=i+1){
      try {
        if (!rel.le(ar[i],ar[i+1])){
          at o = ar[i];
          ar[i]=ar[i+1];
          ar[i+1]=o;
          result=true;
        }
      }catch (ArrayIndexOutOfBoundsException _){}
    }
    return result;
```

```
  }
}
```

e) Schreiben Sie zwei Beispielaufrufe der Methode `bubbleSort` mit `SmallerEven` bzw. `SmallerShort`.

Lösung

TestBubble.java

```
package name.panitz.aufgaben;
public class TestBubble{
  public static void main(String [] _){
    String [] xs = {"1","4444","22","55555","","333"};
    System.out.print("[");
    for (String x:xs)System.out.print(","+x);
    System.out.println("]");
    Bubble.bubbleSort(xs,new SmallShort());
    System.out.print("[");
    for (String x:xs)System.out.print(","+x);
    System.out.println("]");

    int [] ys_ = {2,3,4,5,6,7,78,8,8};
    Integer [] ys = new Integer[9];
    for (int i=0;i<9;i++) ys[i]=new Integer(ys_[i]);
    System.out.print("[");
    for (Integer x:ys)System.out.print(","+x);
    System.out.println("]");
    Bubble.bubbleSort(ys,new SmallEven());
    System.out.print("[");
    for (Integer x:ys)System.out.print(","+x);
    System.out.println("]");
  }
}
```

Literatur

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.

[Pan00] Sven Eric Panitz. Generische Typen in Bolero. *Javamagazin*, 4 2000.