

## 45 Neuerungen unter J2SE V5

### 45.1 Aufzählungstypen (Enumerations)

Unter J2SE V5 werden neuerdings Aufzählungstypen zur Verfügung gestellt. Es handelt sich nicht nur um "glorifizierte Integer", sondern um eine echte, gut durchdachte Spracherweiterung, nach Auffassung vieler Java-Kenner die wichtigste des neuen Java Releases. Enumerations können als normale Klassen aufgefasst werden. Sie enthalten statische Instanzvariablen, die Referenzen auf Objekte sind, welche die einzelnen Werte der Aufzählung repräsentieren.

Die Aufzählungswerte besitzen einen Namen und einen Integer-Index, der von Null in der Reihenfolge in der Deklaration aufsteigt. Man kann Enums sowohl in switch- wie in for-Konstrukten verwenden. Sie lassen sich auch direkt mit == bzw. != vergleichen.

Folgendes Beispiel zeigt die wesentlichen Eigenschaften.

```
// EnumEx1.java

import static ch.aplu.util.Console.*;

class EnumEx1
{
    // Es werden 4 Objekte, naemlich die 4 Jahreszeiten erzeugt
    // Deklaration darf sich nicht in einer Methode befinden
    private
        enum Jahreszeiten {FRUEHLING, SOMMER, HERBST, WINTER};

    EnumEx1()
    {
        // zufaellig 0, 1, 2, 3
        int zufallszahl = (int)(4 * Math.random());

        // values() liefert einen Array mit den Refs zu
        // allen Jahreszeiten
        // Referenz zu einer zufälligen Jahreszeit:
        Jahreszeiten jahreszeit =
```

```
Jahreszeiten.values()[zufallszahl];

// switch
switch(jahreszeit)
{
    case FRUEHLING:
        println("sanfter Fruhlingswind");
        break;

    case SOMMER:
        println("Flimmernde Sommerhitze");
        break;

    case HERBST:
        println("Farbiges Herbstlaub");
        break;

    case WINTER:
        println("Verschneite Winterlandschaft");
        break;
}

// Vergleich mit ==
if ( jahreszeit == Jahreszeiten.WINTER)
    println("ich gehe skilaufen");
else
    println("ich gehe baden");

// Indexwert, hier 0,1,2,3
println("Index: " + jahreszeit.ordinal());

// Vergleich mit compareTo(): Differenz der Indexwerte
println("compareTo ... WINTER:" +
        jahreszeit.compareTo(Jahreszeiten.WINTER));

// Durchlaufen aller Werte
println("Die Jahrezeiten sind:");
for (Jahreszeiten jz : Jahreszeiten.values())
{
    print(jz.name()); // Oder jz.toString() oder nur jz
    if (jz == Jahreszeiten.HERBST)
        println(" (Farbiges Herbstlaub)");
    else
        println();
}
}

public static void main(String[] args)
```

```
{
    new EnumEx1();
}
}
```

Ereignisgesteuerte Programme, die als Automat (state machine) aufgebaut sind und eine Ereignisschleife besitzen, spielen eine außerordentlich wichtige Rolle. Dabei kann der Zustand mehrere Zustandswerte durchlaufen. In vielerlei Hinsicht ist es besser, den Zustand als Aufzählungstyp und nicht als `int`, `String` oder als Interface mit Konstanten zu implementieren. Insbesondere wird das Programm dadurch übersichtlicher und weniger fehleranfällig, da sich Schreibfehler in der Wahl der Konstanten bereits bei der Compilation bemerkbar machen.

Das folgende Programm stellt ein Musterbeispiel für ein ereignisgesteuertes Programm mit einer Ereignisschleife dar. Es enthält die drei Buttons *Run*, *Stop* und *Quit*, mit denen der Zustand geändert wird.

```
// EnumEx2.java

import java.awt.event.*;
import javax.swing.*;
import ch.aplu.turtle.*;

public class EnumEx2
{
    private class ButtonActionAdapter implements ActionListener
    {
        public void actionPerformed(ActionEvent evt)
        {
            Object source = evt.getSource();
            if (source == runButton)
                state = States.RUNNING;
            if (source == stopButton)
                state = States.STOPPED;
            if (source == quitButton)
                state = States.QUITTING;
        }
    }

    private enum States {STOPPED, RUNNING, QUITTING};
    private States state = States.STOPPED;

    private JButton runButton = new JButton("Run");
    private JButton stopButton = new JButton("Stop");
    private JButton quitButton = new JButton("Quit");

    private Turtle t = new Turtle();
```

```
public EnumEx2()
{
    ButtonActionAdapter adapter = new ButtonActionAdapter();
    runButton.addActionListener(adapter);
    stopButton.addActionListener(adapter);
    quitButton.addActionListener(adapter);

    t.getPlayground().add(runButton);
    t.getPlayground().add(stopButton);
    t.getPlayground().add(quitButton);
    t.getPlayground().validate();

    while (state != States.QUITTING) // Ereignisschleife
    {
        switch(state)
        {
            case STOPPED:
                Thread.yield();
                break;

            case RUNNING:
                t.forward(10).left(10);
                break;
        }
    }
    System.exit(0);
}

public static void main(String[] args)
{
    new EnumEx2();
}
}
```

Wie bekannt, sollte man in der "engen" Schleifenwiederholung im Zustand STOPPED mit `Thread.yield()` oder mit `Thread.sleep()` den Prozessor zur anderweitigen Verwendung freigeben. Auf vielen Plattformen führt dies zu einer merklichen Verbesserung der Antwortzeiten.

Für die Verwendung desselben Aufzählungstyps in mehreren Quelldateien ist es wichtig zu wissen, dass dieser automatisch `static` ist. Man kann also auf seine Werte durch Vorstellen des Klassennamens der Klasse, in der er deklariert ist, zugreifen. Dies gilt allerdings nur, falls er nicht, wie im vorherigen Beispiel, `private` deklariert ist.

## 45.2 for-each-Struktur

Zur Vereinfachung der Schreibweise beim Durchlaufen von Collections und Arrays wurde ein neues Sprachkonstrukt im Zusammenhang mit der for-Schleife eingeführt. Mit der neuen for-Schleife können allerdings weder Elemente gelöscht noch verändert werden. Wir zeigen dies am Beispiel eines Arrays.

```
// ForEachEx1.java

import static ch.aplu.util.Console.*;

class ForEachEx1
{
    ForEachEx1()
    {
        int[] a = new int[5];

        // Fuellen des Arrays, for-each ungeeignet
        for (int i = 0; i < a.length; i++)
            a[i] = i * i;

        // Durchlauf mit for-each
        for (int z : a)
            println(z);

        // Exakt das gleiche wie
        for (int i = 0; i < a.length; i++)
        {
            int z = a[i];
            println(z);
        }
    }

    public static void main(String[] args)
    {
        new ForEachEx1();
    }
}
```

Man liest diese for-each-Schleife am besten mit „für jedes Element z in a...“. z ist also nicht etwa der Arrayindex, sondern vielmehr eine im for-Körper lokale Variable, mit der durch den Array iteriert wird. Der Arrayindex bleibt dabei versteckt. Man beachte, dass es mit for-each nicht möglich ist, dem Arrayelement einen Wert zuzuweisen. Syntaktisch richtig ist zwar

```
for (int z : a)
    z = 2;
```

Dies wird aber in

```
for (int i = 0; i < a.length; i++)
{
    int z = a[i];
    z = 2;
}
```

übersetzt, wodurch offensichtlich `z`, aber nicht `a[i]` der Wert 2 zugewiesen wird.

Eine gewisse Eleganz ist for-each-Struktur nicht abzuspüren, wie wir am folgenden Beispiel erkennen.

```
// ForEachEx2.java

import static ch.aplu.util.Console.*;

class ForEachEx2
{
    ForEachEx2()
    {
        String[] farben = new String[] {"rot", "gruen", "blau"};
        for (String s : farben)
            println(s);
    }

    public static void main(String[] args)
    {
        new ForEachEx2();
    }
}
```

## 45.3 printf() und die Klasse Formatter

In den Klassen `PrintStream` und `PrintWriter` wurde mit J2SE V5 die Methode `printf()` hinzugefügt, welche die Formatierung von Textzeilen wesentlich vereinfacht, insbesondere im Zusammenhang mit dem Ausschreiben von tabellenartig ausgerichteten Zahlenreihen. Die Syntax wurde weitgehend von der gleichlautenden Methode in der Programmiersprache C übernommen. `printf()` besitzt als ersten Parameter einen Formatstring, und anschließend eine beliebige Anzahl von auszuschreibenden Werten (**format specifier**). `printf()` macht also weitgehend von `varargs` Gebrauch. Die Formatierung erfolgt intern mit der Klasse `java.util.Formatter`, in deren Dokumentation das etwas gewöhnungsbedürftige Verfahren gut beschrieben ist. Die neuen Formatierungsmöglichkeiten werden vor allem in drei Gebieten eingesetzt, nämlich beim Ausschreiben

1. auf die Betriebssystem-Console bzw. in das Console-Fenster der Klasse `ch.aplu.util.Console`
2. in eine Textdatei
3. in ein GUI-Element (TextField, Label, TextArea, Titlebar, usw.) Im ersten Fall verwendet man `System.out.printf()`, im zweiten erstellt man mit

```
out = new PrintWriter(  
    new BufferedWriter(  
        new FileWriter(file)));
```

einen `PrintWriter` und schreibt mit `out.printf()` in die Datei. Im dritten Fall erzeugt man mit `String.format()` einen formatierten String, den man beliebig weiter verwenden kann.

Im Formatstring können Textausgaben mit Formatangaben beliebig kombiniert werden. Das `%`-Zeichen leitet eine Formatangabe ein. Von den nachfolgenden Zeichen werden so viele als Formatzeichen aufgefasst, bis das Format spezifiziert ist. Nachfolgend werden die Zeichen wieder als anzuzeigenden Text aufgefasst (außer es folgt gerade wieder ein `%`). `PrintfEx1.java` zeigt eine typische Formatierung.

```
// PrintfEx1.java  
  
class PrintfEx1  
{  
    PrintfEx1()  
    {  
        int a = 12;  
        int b = 15;  
        System.out.printf("Der Ggt von (%d, %d) ist %d", a, b,  
ggt(a, b));  
    }  
  
    int ggt(int a, int b)  
    {  
        if (b == 0)  
            return (a);  
        return (ggt(b, a % b));  
    }  
  
    public static void main(String[] args)  
    {  
        new PrintfEx1();  
    }  
}  
  
class PrintfEx1  
{  
    PrintfEx1()
```

```

{
    int a = 12;
    int b = 15;
    System.out.
        printf("Der Ggt von (%d, %d) ist %d", a, b, ggt(a, b));
}

int ggt(int a, int b)
{
    if (b == 0)
        return (a);
    return (ggt(b, a % b));
}

public static void main(String[] args)
{
    new PrintfEx1();
}
}

```

Für viele Anwendungen wichtig ist die Möglichkeit, die Feldbreite anzugeben und doubles auf eine vorgegebenen Anzahl Nachkommastellen zu runden. Einen Zeilenumbruch muss man explizit angeben, entweder als `\n` oder mit dem plattformunabhängigen `%n` (je nachdem wird `\r\n` oder nur `\n` erzeugt). In `PrintfEx2.java` werden wichtige Formatierungen verwendet, wobei die Begrenzer `|` lediglich die Sichtbarkeit der Feldlänge verbessern.

```

// PrintfEx2.java
// Verwendung von %n an Stelle von \n

// System.out.println() mit out.println() vereinfachen:
import static java.lang.System.out;

class PrintfEx2
{
    public static void main(String[] args)
    {
        int a = 1023;
        long b = 123456789012345L;
        short c = 123;
        char d = 'X';
        float e = 3.14159f;
        double f = 3.1415926535898;          // 13 Nachkommastellen

        out.
            printf("a = |%d|%n",a); // int (dezimal)
        out.
            printf("a = |%o|%n",a); // int (oktal)
    }
}

```



```

out.
    printf("a = |%x|%n",a); // int (hex)
out.
    printf("a = |%7d|%n",a); // int Feldw. 7,rechtsbuendig
out.
    printf("a = |%-7d|%n",a); // int Feldw. 7,linksbuendig
out.
    printf("b = |%20d|%n",b); // long Feldw.20,rechtsbuendig
out.
    printf("c = |%d|%n",c); // short
out.
    printf("d = |%c|%n",d); // char
out.
    printf("e = |%f|%n",e); // float Gleitkommadarstellung
out.
    printf("e = |%e|%n",e); // float Exponentialdarstellung
out.
    printf("f = |%15.3f|%n",f); // double Feldw. 15,
                                // Nachkommast. 3 gerundet
out.
    printf("f = |%-15.5f|%n",f); // dasselbe linksbuendig
out.
    printf("f = |%15.5e|%n",f); // Exponentialdarstellung
}
}

```

Die Formatierung ist insbesondere für Tabellendarstellungen sehr praktisch. Im folgenden Beispiel wird davon Gebrauch gemacht, dass auch die Klasse `ch.aplu.Console` die (statische) Methode `printf()` enthält. Importiert man die Klasse statisch, so kann beim Aufruf sogar der Klassenname weggelassen werden. Das Programm ist bewusst weitgehend ohne Verwendung der OOP geschrieben.

```

// PrintfEx3.java
import static java.lang.Math.*;
import static ch.aplu.util.Console.*;

class PrintfEx3
{
    public static void main(String[] args)
    {
        printf("Matrix mit ints, Vorzeichen immer anzeigen%n%n");
        int[] z = new int[3];

        printf("%10s %10s %10s%n",
            "Spalte_0", "Spalte_1", "Spalte_2");
        printf("%32s%n", "-----");
        for (int i = 0; i < 5; i++)

```

```

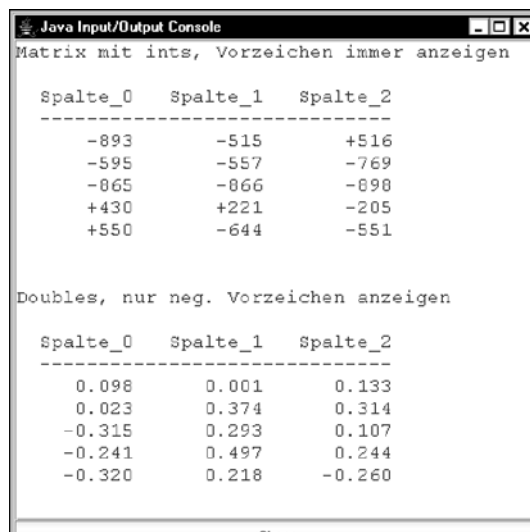
{
    for (int j = 0; j < 3; j++)
        z[j] = -1000 + (int)(2001*random());
    printf("%+10d %+10d %+10d\n", z[0], z[1], z[2]);
}

printf("\n\nDoubles, nur neg. Vorzeichen anzeigen\n\n");
double[] a = new double[3];

printf("%10s %10s %10s\n",
       "Spalte_0", "Spalte_1", "Spalte_2");
printf("%32s\n", "-----");
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 3; j++)
        a[j] = -0.5 + random();
    printf("%10.3f %10.3f %10.3f\n", a[0], a[1], a[2]);
}
}
}

```

Das Resultat ist erfreulich (Fig. 45.1).



```

Java Input/Output Console
Matrix mit ints, Vorzeichen immer anzeigen

Spalte_0    Spalte_1    Spalte_2
-----
-893        -515        +516
-595        -557        -769
-065        -066        -098
+430        +221        -205
+550        -644        -551

Doubles, nur neg. Vorzeichen anzeigen

Spalte_0    Spalte_1    Spalte_2
-----
0.098       0.001       0.133
0.023       0.374       0.314
-0.315      0.293       0.107
-0.241      0.497       0.244
-0.320      0.218       -0.260

```

Fig. 45.1: Ausgabe von PrintfEx3

## 45.4 Autoboxing/Unboxing

Vor allem aus Gründen der Effizienz gibt es in Java neben den Referenztypen (Objekten) noch Basistypen (primitive Datentypen, einfache Datentypen). Dies führt zu einer gewissen Uneinheitlichkeit, da sich die beiden Typenarten in vielen Beziehungen unterschiedlich verhalten (beispielsweise bei Zuweisungen oder Vergleichen). Oft ist es nötig, aus dem Basistyp einen entsprechenden Referenztyp zu erstellen. Dazu dienen die Hüllklassen (Wrapperklassen). Aus einem Basistyp erstellt man den entsprechenden Referenztyp üblicherweise durch eine Instanzierung, beispielsweise

```
Integer iObj = new Integer(123);
```

und aus dem Referenztyp den entsprechenden Basistyp mit der Methode `xxxValue()`, beispielsweise

```
int i = iObj.intValue();
```

Das neue Sprachfeature vereinfacht diese Konversionen, indem sie vom Compiler automatisch durchgeführt werden. Der Übergang von Basistyp zum Referenztyp nennt man Autoboxing:

```
Integer iObj = 123;
```

und vom Referenztyp zum Basistyp Unboxing (manchmal auch Autounboxing):

```
int i = iObj;
```

Im folgenden Beispiel sieht man auch, dass der Compiler, wenn nötig, das Unboxing von Boolean durchführt.

```
// AutoboxEx1.java
import ch.aplu.util.*;

class AutoboxEx1 extends Console
{
    AutoboxEx1()
    {
        Integer iObj = 123; // Autoboxing, same as
        // Integer z = new Integer(123);
        println(iObj);

        Object o = 123; // Autoboxing
        println(o);

        int i = iObj; // Unboxing, same as
        // int i = iObj.intValue();
        println(i);
    }
}
```

```
    Boolean isOk = true; // Autoboxing
    if (isOk)
        println("nur legal in Java 5");
}

public static void main(String[] args)
{
    new AutoboxEx1();
}
}
```

Besondere Vorsicht muss man beim Vergleich walten lassen, da man durch Autoboxing/Unboxing die Unterschiede von Referenz- und Basistypen leicht übersieht, was zu bösen Laufzeitfehlern führen kann. Von einem "unified type-system" kann also keine Rede sein. Es gilt immer noch die wichtige Regel, dass man Objekte nie mit == bzw. != vergleichen darf. Als besonders tückisch stellt sich heraus, dass in einem gewissen Bereich der numerischen Werte (üblicherweise zwischen -128 und 127) auch der Vergleich mit == zum richtigen Wahrheitswert führen kann, weil der Compiler aus Gründen der Effizienz einen internen Speicher (cache) anlegen kann, in dem Objekte mit den gleichen Werten beim Autoboxing auch dieselbe Referenz erhalten.

```
// AutoboxEx2.java

import ch.aplu.util.*;

class AutoboxEx2 extends Console
{
    AutoboxEx2()
    {
        Integer z1 = 123;
        Integer z2 = 123;
        if (z1 == z2)
            println("z1 == z2 is true");
        else
            println("z1 == z2 is false");
        // Gibt true, da Java in der jetzigen Version von
        // -128..127 einen Cache aufweist

        Integer z3 = 123456;
        Integer z4 = 123456;
        if (z3 == z4)
            println("z3 == z4 is true");
        else
            println("z3 == z4 is false");
        // Gibt false, da ausserhalb Cache

        // Ist ok:
    }
}
```

```
        if (z3.equals(z4))
            println("z3.equals(z4) is true");
        else
            println("z3.equals(z4) is false");
    }

    public static void main(String[] args)
    {
        new AutoboxEx2();
    }
}
```

Bereits bisher standen Basistypen durch die Erweiterung/Verkürzung in einer Beziehung zueinander, welche bei der Zuweisung, dem Vergleich und dem Überladen eine wichtige Rolle spielen:

byte -> short -> int -> long -> float -> double

Durch Autoboxing/Unboxing werden nun die Regeln, mit welchen beim Aufruf von überladenen Methoden nach der passenden (spezifischen) Methode gesucht wird, noch einmal komplizierter und noch weniger durchsichtig. Dies ist auch deswegen so, weil alle Wrapperklassen die gemeinsame Basisklasse `Number` besitzen. Das folgende bekannte Beispiel zeigt, dass das Auffinden der spezifischen Methode trotzdem einigermaßen vernünftig ist (es gibt natürlich exakte Regeln dafür). Kommentiert man `foo(int i, int j)` aus, so compiliert das Programm wegen dem Autoboxing immer noch.

```
// AutoboxEx3.java

import static ch.aplu.util.Console.*;

class AutoboxEx3
{
    AutoboxEx3()
    {
        foo(1, 1);
        foo(1L, 1);
        foo(1, 1, 1);
        foo((short)1, 1);
        foo(1, 1L);
    }

    void foo(int i, int j)
    {
        println("foo(int, int)");
    }

    void foo(Integer i, int j)
```

```
{
    println("foo(Integer, int)");
}

void foo(Long i, int j)
{
    println("foo(Long, int)");
}

void foo(Integer... i)
{
    println("foo(Integer...)");
}

void foo(Number... i)
{
    println("foo(Number...)");
}

public static void main(String[] args)
{
    new AutoboxEx3();
}
}
```

Es stellt sich die berechnete Frage, ob Autoboxing kombiniert mit Unboxing nicht mehr neue Probleme mit sich bringt als löst.

## 45.5 Die Klasse StringBuilder

Es ist bekannt, dass die Entwickler von Java bei der Implementierung von Strings nicht die glücklichste Hand hatten. Dies zeigt sich vor allem auch an der relativ schlechten Performanz der Stringklassen betreffend Speicherbedarf und Verarbeitungsgeschwindigkeit. Mit J2SE V5 wird eine neue Klasse `StringBuilder` zur Verfügung gestellt, die wegen des Verzichtes auf die Thread-Synchronisation etwas performanter als die Klasse `StringBuffer` ist und diese in Single-Thread-Applikationen ersetzen soll. Da `StringBuilder` exakt die gleichen Methoden wie `StringBuffer` aufweist, sollte man in Zukunft bei Single-Thread-Applikationen immer an Stelle von `StringBuffer` die neue Klasse `StringBuilder` verwenden. Der Geschwindigkeitsvorteil ist relativ bescheiden, so dass es sich nicht lohnen dürfte, bestehenden Code abzuändern und neu zu compilieren. Im folgenden Beispiel wird ein Benchmark-Vergleich beim Füllen eines `StringBuffers` und eines `StringBuilders` mit der Methode `add()` durchgeführt.

```
// StringBuilderEx1.java

import ch.aplu.util.*;

public class StringBuilderEx1
{
    private static StringBuffer getStringBuffer(String text,
                                                int rounds)
    {
        StringBuffer sbuf = new StringBuffer(text.length() *
                                                rounds);
        for (int i = 0; i < rounds; i++)
            sbuf.append(text);
        return sbuf;
    }

    private static StringBuilder getStringBuilder(String text,
                                                int rounds)
    {
        StringBuilder sbd = new StringBuilder(text.length() *
                                                rounds);
        for (int i = 0; i < rounds; i++)
            sbd.append(text);
        return sbd;
    }

    public static void main(String[] args)
    {
        String text = "foobar";
        int rounds = 1000000;
        LoResTimer timer = new LoResTimer();
        double totTime1 = 0;
        double totTime2 = 0;
        Console.printf(
            "      StringBuffer (s)   StringBuilder (s)\n");
        for (int i = 0; i < 5; i++)
        {
            timer.start();
            StringBuffer sbuf = getStringBuffer(text, rounds);
            timer.stop();
            double time1 = timer.getTime() / 1E6;
            totTime1 += time1;

            timer.start();
            StringBuilder sbd = getStringBuilder(text, rounds);
            timer.stop();
            double time2 = timer.getTime() / 1E6;
            totTime2 += time2;
        }
    }
}
```

```

        Console.printf("%3d. %10.3f %18.3f\n", i + 1, time1,
                        time2);
    }
    Console.printf("\nMittelwerte:\n");
    Console.printf("      %10.3f %18.3f (%2.0f%%)\n",
                  totTime1 / 5, totTime2 / 5,
                  totTime2 / totTime1 * 100);
}
}

```

Die Resultate sind natürlich stark vom verwendeten Rechner und Betriebssystem abhängig. Typischerweise ergibt sich aber eine Laufzeit-Reduktion bei der Verwendung von `StringBuilder` auf rund 60%.

	StringBuffer (s)	StringBuilder (s)
1.	0.360	0.311
2.	0.340	0.151
3.	0.330	0.140
4.	0.331	0.260
5.	0.331	0.140
Mittelwerte:		
	0.338	0.200 (59%)

## 45.6 Generische Datentypen und Methoden (Generics)

Eine der Grundaufgaben der Informatik besteht darin, allgemein gültige algorithmisierbare Abläufe zu erkennen und als Programm derart zu formulieren, dass dieses in vielen verschiedenen konkreten Situationen anwendbar ist. Dieser Denkprozess des Abstrahierens treffen wir auch in der Mathematik und allen Naturwissenschaften an, sind doch mathematische und physikalische Theorien so allgemein formuliert, dass sie in vielen konkreten Situationen einsetzbar sind. Beispielsweise behalten die Naturgesetze unter wohldefinierten Voraussetzungen ihre Gültigkeit, unabhängig davon, wo, wann und von wem sie überprüft werden.

Es ist deshalb verständlich, dass man in der Informatik Programmstrukturen schafft, die weitgehende Abstraktionsstufen zulassen. So werden beim prozeduralen Programmieren möglichst in sich geschlossene Prozeduren (Funktionen, in Java: Methoden) geschrieben, die



sich dadurch auszeichnen, in vielen konkreten Situationen anwendbar zu sein. Dies erreicht man vor allem durch eine gut durchdachte Parametrisierung mit einer günstigen Wahl der Parameterliste und des Rückgabewerts.

Die nächste Abstraktionsstufe erfolgt mit der OOP: Man deklariert Klassen bzw. eine Klassenhierarchie, aus denen sich in möglichst vielen konkreten Situationen Objekte mit gewünschten Eigenschaften und Verhalten erzeugen lassen. Dabei werden Werkzeuge bereit gestellt, um neben den Datentypen, die in der Programmiersprache eingebaut sind, eigene neue Datentypen zu schaffen.

Es ist nun ziemlich naheliegend, die Möglichkeiten zur Abstraktion noch etwas zu erweitern, indem man Strukturen zulässt, bei denen auch Datentypen bei der Deklaration der Struktur als Parameter aufgefasst werden. Diese Datentypen werden erst bei der Verwendung der Struktur durch aktuelle Datentypen ersetzt, analog wie man bei den Parameterwerten die formalen Parameter beim Aufruf durch aktuelle Parameter ersetzt. Als Strukturen kommen Methoden, aber auch Klassen und Interfaces in Frage, die man in diesem Fall **generische Methoden** bzw. **generische Klassen** oder **generische Interfaces** nennt, da man daraus verschiedene konkrete Methoden bzw. Klassen erzeugen kann. Zusammenfassend spricht man in Java von **Generics** oder von **parametrisierten Typen**.

Wie bei jeder Spracherweiterung handelt man sich mit Generics Vor- und Nachteile ein. Als Vorteil wird meist angeführt, dass große Programme in Zukunft noch robuster und sicherer geschrieben werden können, da die Generics dem Compiler ermöglichen, typische Programmierfehler im Zusammenhang mit falschen Datentypen und deren Umwandlung mit Casts bereits bei der Compilation zu entdecken. Da gewisse Klassen und Methoden bis auf wenige verwendete Datentypen praktisch identisch sind, lässt sich durch die Möglichkeit, auch Datentypen als Parameter aufzufassen, die verpönte und fehleranfällige Codeduplikation vermeiden. Als Nachteil handelt man sich aber durch die Einführung von Generics eine wesentliche Komplikation von Java ein. Da allerdings die Spracherweiterungen mit Generics in Java glücklicherweise rückwärtskompatibel ist, bleibt der bisherig geschriebene Code immer noch gültig und lässt sich, bis auf einige Compilerwarnungen, problemlos verwenden.

### 45.6.1 Verwendung von vordefinierten generischen Klassen

Es ist methodisch ratsam, im Zusammenhang mit der Einführung von Generics zwei Phasen strikte zu unterscheiden, nämlich die Erstellungsphase (Deklaration) und die Anwendungsphase (Aufruf). Um einen ersten motivierenden Eindruck von Generics zu geben, ist es durchaus sinnvoll, nicht systematisch von der Erzeugung zur Verwendung vorzugehen, sondern zuerst einige vordefinierte Generics anzuwenden. Auch begehen wir im Zusammenhang mit Generics wieder "Auslassungssünden", denn im Folgenden wird keine Vollständigkeit angestrebt.

Das erste Beispiel verwendet die Klasse `Vector` in ihrer nicht-generischen Form. Bekanntlich kann man irgendwelche Referenzen darin abspeichern, da die Komponenten den Datentyp `Object` besitzen. Beim Zurückholen müssen wir aber wissen, um welchen konkreten Datentyp es sich handelt und den Rückgabewert von `elementAt()` mit einer unschönen Schreibweise entsprechend casten.

```
// GenUseEx1.java

import ch.aplu.turtle.*;
import java.awt.Color;
import java.util.*;

public class GenUseEx1
{
    public GenUseEx1()
    {
        Vector v = new Vector();

        v.add(new Turtle(Color.red));
        v.add(new Turtle(Color.green));

        ((Turtle)(v.elementAt(0))).forward(100);
        ((Turtle)(v.elementAt(1))).back(100);
    }

    public static void main(String[] args)
    {
        new GenUseEx1();
    }
}
```

Wir nennen ein solches Programm aus folgendem Grund **nicht-typsicher**: Kopieren wir versehentlich an Stelle einer Turtle-Referenz eine Integer-Referenz in den Vektor, indem wir die Zeile

```
v.add(new Turtle(Color.green));
```

durch

```
v.add(new Integer(1));
```

ersetzen, so können wir das Programm ohne jede Fehlermeldung oder Warnung compilieren. Es bricht aber mit einer `java.lang.ClassCastException` ab, was viel schlimmer als ein Compilationsfehler ist.

Die generische Klasse `Vector<T>` (gesprochen "Vektor von T") ist Teil des neuen Collection Frameworks, das die einfache und typsichere Speicherung von beliebigen Referenzen erlaubt. (Es gibt dazu mehrere hervorragende Tutorials, die man mit einer Web-Suchmaschine und den Stichwörtern *collection framework tutorial 1.5* findet.) T in den Spitzklammern steht als Platzhalter für einen **Typ-Parameter**, mit dem wir bei der Verwendung angeben, welche Typen im Vector gespeichert werden dürfen. Neben dem neuen generischen `Vector<T>` ist nach wie vor die Klasse `Vector` ohne Typ-Parameter verfügbar (**raw-type**). Diese verhält sich gleich wie vor der Einführung der Generics, so dass

wir, abgesehen von einigen Compilerwarnungen, alte Programme problemlos compilieren und ausführen können. Wir schreiben das vorherige Programm nun typsicher.

```
// GenUseEx2.java

import ch.aplu.turtle.*;
import java.awt.Color;
import java.util.*;

public class GenUseEx2
{
    public GenUseEx2()
    {
        Vector<Turtle> v = new Vector<Turtle>();

        v.add(new Turtle(Color.red));
        v.add(new Turtle(Color.green));

        v.elementAt(0).forward(100);
        v.elementAt(1).back(100);
    }

    public static void main(String[] args)
    {
        new GenUseEx2();
    }
}
```

Begehen wir aus Unachtsamkeit denselben Fehler wie oben, indem wir eine Integer-Referenz statt eine Turtle-Referenz im Vektor abspeichern, so werden wir glücklicherweise nun bereits bei der Compilation darauf aufmerksam gemacht.

Die neue for-each-Struktur ist bestens für den Durchlauf durch Collections geeignet. Das Füllen und Durchlaufen eines Vektors lässt sich damit sehr elegant, insbesondere ohne jeden Cast, formulieren. Im folgenden Beispiel erzeugen wir zuerst einen Turtlevektor `Vector<Turtle>` mit 4 verschiedenfarbigen Turtles im gleichen Playground, die übereinander liegen (die blaue ist zuoberst). Nachher durchlaufen wir den Vektor und schieben die Turtles unterschiedlich weit nach oben.

```
// GenUseEx3.java

import ch.aplu.turtle.*;
import java.awt.*;
import java.util.*;

public class GenUseEx3
{
    private final Font font =
```

```

    new Font("Helvetica", Font.PLAIN, 12);

    public GenUseEx3()
    {
        Vector<Turtle> v = new Vector<Turtle>();

        v.add(new Turtle(Color.red));
        v.add(new Turtle(v.get(0), Color.green));
        v.add(new Turtle(v.get(0), Color.yellow));
        v.add(new Turtle(v.get(0), Color.blue));

        int s = 10;
        for (Turtle t : v)
        {
            t.forward(s = s + 40);
            t.setFont(font).label("    " + t.getColor());
        }
    }

    public static void main(String[] args)
    {
        new GenUseEx3();
    }
}

```

Wir verwenden im Aufruf von `forward()` einen Seiteneffekt für `s`. Ob dies als unschöner Trick oder elegante Programmiertechnik angesehen wird, ist umstritten. Das Resultat erfreut uns trotzdem (Fig. 45.2).

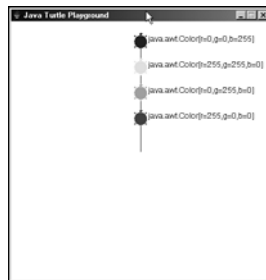


Fig. 45.2: Resultat von GenUseEx3

Den generischen Vektor können wir auch in einer Parameterliste wie einen gewöhnlichen Datentyp ansehen. Im nächsten Beispiel übergeben wir den generischen Vektor der Methode `showMe()`.

```
// GenUseEx4.java

import ch.aplu.turtle.*;
import java.awt.Color;
import java.util.*;

public class GenUseEx4
{
    public GenUseEx4()
    {
        Vector<Turtle> v = new Vector<Turtle>();

        v.add(new Turtle(Color.red));
        v.add(new Turtle(v.get(0), Color.green));
        v.add(new Turtle(v.get(0), Color.yellow));
        v.add(new Turtle(v.get(0), Color.blue));

        showMe(v);
    }

    private void showMe(Vector<Turtle> v)
    {
        int i = 0;
        for (Turtle t : v)
        {
            t.left(30 * i++);
            t.forward(50);
            i++;
        }
    }

    public static void main(String[] args)
    {
        new GenUseEx4();
    }
}
```

Selbstverständlich können wir auch in einem generischen Vektor immer noch Subtypen des Deklarationstyps speichern, denn diese stehen ja in einer is-a-Relation zum Deklarationstyp. Um dies zu zeigen, verwenden wir die Klassenhierarchie aus Kap. 13 mit der aus der Klasse `Turtle` abgeleiteten Klasse `TurtleKid` und den wiederum daraus abgeleiteten Klassen `TurtleGirl` und `TurtleBoy`. Alle drei abgeleiteten Klassen besitzen eine Methode `shape()`, die je nach Klassenzugehörigkeit eine andere Figur zeichnet.

Deklarieren wir einen Vektor mit `TurtleKids`, so können wir darin auch `Girls` und `Boys` abspeichern. Wegen der Polymorphie wird beim Aufruf von `shape()` mit den Vektorkomponenten das entsprechende Muster gezeichnet.

```
// GenUseEx5.java
// Verwendet das Package homeland

import ch.aplu.turtle.*;
import java.util.*;
import homeland.*;

public class GenUseEx5
{
    public GenUseEx5()
    {
        Vector<TurtleKid> v = new Vector<TurtleKid>();

        v.add(new TurtleKid());
        v.add(new TurtleBoy());
        v.add(new TurtleGirl());

        showMe(v);
    }

    private void showMe(Vector<TurtleKid> v)
    {
        for (TurtleKid t : v)
            t.shape();
    }

    public static void main(String[] args)
    {
        new GenUseEx5();
    }
}
```

Es wäre allerdings ein Irrtum zu glauben, dass ein generischer Vektor, dessen Komponenten einen Subtyp haben, selbst ein Subtyp des Vektors mit Komponenten des Basistyps ist. Beispielsweise ist `Vector<TurtleBoy>` kein Subtyp von `Vector<TurtleKid>`, obschon `TurtleBoy` ein Subtyp von `TurtleKid` ist. Wir zeigen dies in `GenUseEx6.java`, das zu einem Compilationsfehler führt.

```
// GenUseEx6.java

/* Compile time error:
showMe(java.util.Vector<homeland.TurtleKid>)
cannot be applied to (java.util.Vector<homeland.TurtleBoy>)
*/
```

```
import ch.aplu.turtle.*;
import java.util.*;
import homeland.*;

public class GenUseEx6
{
    public GenUseEx6()
    {
        Vector<TurtleBoy> v = new Vector<TurtleBoy>();

        v.add(new TurtleBoy());
        v.add(new TurtleBoy());

        showMe(v);
    }

    private void showMe(Vector<TurtleKid> v)
    {
        for (TurtleKid t : v)
            t.shape();
    }

    public static void main(String[] args)
    {
        new GenUseEx6();
    }
}
```

Wie GenUseEx6a.java zeigt, funktioniert das Analoge mit Arrays bestens, da ein Array mit TurtleBoys auch ein Array mit TurtleKids ist. Solch subtile Unterschiede gehören nicht zu den Vorteilen einer Programmiersprache.

```
// GenUseEx6a.java

import ch.aplu.turtle.*;
import java.util.*;
import homeland.*;

public class GenUseEx6a
{
    public GenUseEx6a()
    {
        TurtleBoy[] a = new TurtleBoy[2];

        a[0] = new TurtleBoy();
        a[1] = new TurtleBoy();
    }
}
```

```
    showMe(a);
}

private void showMe(TurtleKid[] k)
{
    for (TurtleKid t : k)
        t.shape();
}

public static void main(String[] args)
{
    new GenUseEx6a();
}
}
```

Bei der Verwendung eines generischen Typs kann aber das **?** als **Wildcard (Jokerzeichen)** gesetzt werden. Beispielsweise kann in `Vector<? extends TurtleKid>` ein `TurtleKid` und irgend ein Subtyp davon gespeichert werden. Das folgende Programm ist daher korrekt.

```
// GenUseEx7.java

import java.util.*;
import homeland.*;

public class GenUseEx7
{
    public GenUseEx7()
    {
        Vector<TurtleBoy> v = new Vector<TurtleBoy>();

        v.add(new TurtleBoy());
        v.add(new TurtleBoy());

        showMe(v);
    }

    private void showMe(Vector<? extends TurtleKid> v)
    {
        for (TurtleKid t : v)
            t.shape();
    }

    public static void main(String[] args)
    {
        new GenUseEx7();
    }
}
```



Eine Spezialität von Java ist es, dass zur Laufzeit die generisch erzeugte Typinformation verloren geht. Man spricht von **Typlöschung (type erasure)**. Dies ist darauf zurückzuführen, dass der Compiler bei der Verwendung einer generischen Klasse mit verschiedenen Datentypen nur eine einzige Klasse für den Typ `Object` erzeugt und für die nötigen Casts besorgt ist. Beispielsweise ist zur Laufzeit nicht mehr zwischen `Vector<TurtleGirl>` und `Vector<TurtleBoy>` zu unterscheiden, wie folgendes Beispiel eindrücklich zeigt. (Der Grund für diese etwas sonderbare Implementierung ist historisch erklärbar, da man die Java Virtual Machine (JVM), also die Java-Laufzeitumgebung unverändert lassen kann.)

```
// GenUseEx8.java
// Typloeschung

import static ch.aplu.util.Console.*;
import java.util.*;
import homeland.*;

public class GenUseEx8
{
    public GenUseEx8()
    {
        Vector<TurtleBoy> vBoy = new Vector<TurtleBoy>();
        Vector<TurtleGirl> vGirl = new Vector<TurtleGirl>();

        println(vBoy.getClass() == vGirl.getClass()); // true!
    }

    public static void main(String[] args)
    {
        new GenUseEx8();
    }
}
```

Es ist unumgänglich, dass bereits der Anfänger mit den Grundlagen von Generics vertraut ist, da viele Klassendokumentationen generische Datentypen enthalten, vor allem im Zusammenhang mit dem Collection Framework. Eine in der Praxis wichtige Anwendung ist das Sortieren von Objekten irgendwelchen Datentyps. Wir zeigen im nächsten Beispiel, wie man unter Anwendung einer Liste und der Klasse `Collections` vorgehen muss, um eine Turtlefamilie gemäß eines Farbvergleichs zu sortieren. Dazu hat man zuerst die Klasse `Turtle` so zu erweitern, dass das Interface `Comparable` implementiert wird. Dies bedeutet, dass man die Methode `compareTo()` implementieren muss. In der abgeleiteten Klasse `ColorTurtle` deklarieren wir zudem einige überladene Konstruktoren, die wir für die wichtigsten Instanzierungen benötigen. Für den Farbvergleich ziehen wir mit `getRGB()` den RGB-Index heran. Die Methode `compareTo()` muss gemäß der Dokumentation des Interfaces `Comparable<T>` eine negative Zahl, Null oder eine positive Zahl zurückgeben, je nachdem, ob das eine Objekt kleiner, gleich oder größer als das andere ist.

```
// ColorTurtle.java

import java.awt.Color;
import ch.aplu.turtle.*;

public class ColorTurtle extends Turtle
    implements Comparable<ColorTurtle>
{
    // Ctor1
    public ColorTurtle()
    {
        super();
    }

    // Ctor2
    public ColorTurtle(Color c)
    {
        super(c);
    }

    // Ctor3
    public ColorTurtle(Turtle t, Color c)
    {
        super(t, c);
    }

    public int compareTo(ColorTurtle ct)
    {
        if (getColor().getRGB() > ct.getColor().getRGB())
            return -1;
        if (getColor().getRGB() < ct.getColor().getRGB())
            return 1;
        return 0;
    }
}
```

Die Anwendung erstellt in einer Liste zuerst eine Familie mit ColorTurtles in der Farbreihenfolge schwarz, grün, rot und blau im gleichen Playground wie die Mutter. Die Methode `spreadOut()` lässt die Familie nach einer gewissen Anfangsdrehung ausschwärmen.

```
// GenUseEx9.java

import ch.aplu.turtle.*;
import java.awt.Color;
import java.util.*;

public class GenUseEx9
{
    public GenUseEx9()
    {
        ColorTurtle mother = new ColorTurtle(Color.black);
        List<ColorTurtle> family = new LinkedList<ColorTurtle>();
        family.add(mother);
        family.add(new ColorTurtle(mother, Color.green));
        family.add(new ColorTurtle(mother, Color.red));
        family.add(new ColorTurtle(mother, Color.blue));

        spreadOut(family);
        Collections.sort(family);
        spreadOut(family);
    }

    // May be called by all kinds of Turtles, so use wildcard
    private void spreadOut(List<? extends Turtle> fam)
    {
        double w = 0;
        for (Turtle t : fam)
        {
            t.left(w).forward(100);
            w += 30;
        }
    }

    public static void main(String[] args)
    {
        new GenUseEx9();
    }
}
```

Sortieren wir die Familie, so ist aus der Anfangsdrehung ersichtlich, dass die neue Reihenfolge rot, grün, blau, schwarz ist (Fig. 45.3).

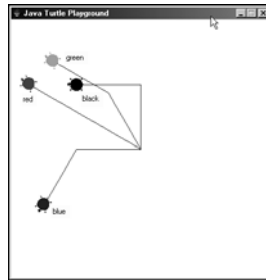


Fig. 45.3: Resultat von GenUseEx9

## 45.6.2 Erstellen von Generics

### 45.6.2.1 Deklaration von generischen Methoden

Es ist wichtig, zwischen generischen Klassen und Methoden zu unterscheiden. Beide haben gemeinsam, dass bei ihrer Verwendung eine Angabe über aktuelle Datentypen gemacht werden müssen. Generische Methoden können sich in einer generischen oder, wie im folgenden Beispiel, in einer nichtgenerischen Klasse befinden. Wir stellen uns die Aufgabe, eine Methode `randomSelect()` zu schreiben, die einen zufällig ausgewählten Parameterwert zurückgibt. In dieser allgemeinen Form ist die Aufgabe ohne Generics nicht zu lösen, da wir uns beim Schreiben der Methode auf einen einzigen Datentyp für die Parameter entscheiden müssen, welches auch der Rückgabotyp der Methode ist. Ohne Verwendung besonderer Techniken müssen wir auch vorgängig entscheiden, wie viele Parameter die Methode besitzen soll. Für zwei ints schreiben wir elegant unter Verwendung des `?`-Operators:

```
int randomSelect(int a, int b)
{
    return Math.random() < 0.5 ? a : b;
}
```

In einer nicht-OOP-Sprache müssten wir für jeden anderen Datentyp sogar auch einen anderen Methodennamen wählen, in Java können wir aber die Methode `randomSelect()` überladen, und für doubles

```
double randomSelect(double a, double b)
{
    return Math.random() < 0.5 ? a : b;
}
```

schreiben. Wollen wir die Methode auch für Strings zur Verfügung stellen, so müssen wir sie noch einmal überladen. Diese Codeduplikation für jeden verschiedenen Datentyp ist offensichtlich wenig elegant und macht uns klar, dass wir ein Sprachkonstrukt benötigen, bei dem auch der Datentyp als Variable (**Typ-Parameter**) aufgefasst wird. Da die Methode für verschiedene Datentypen eingesetzt werden kann, nennen wir sie eine **generische Methode**. Sie wird in Java mit einem Typ-Bezeichner in einer Spitzklammer unmittelbar vor dem Rückgabotyp gekennzeichnet. Üblich sind die Buchstaben T, U, V,..., es kann aber auch irgend ein anderer Bezeichner verwendet werden.

```
<T> T randomSelect(T a, T b)
{
    return Math.random() < 0.5 ? a : b;
}
```

In Java kann eine Typ-Parameter leider kein Basistyp sein; wir müssen an seiner Stelle die entsprechenden Hüllklassen verwenden. Dies wird uns aber durch Autoboxing/Unboxing wesentlich erleichtert, wie wir im nachfolgenden Beispiel erkennen, in dem wir `randomSelect()` für ints, doubles und Strings verwenden.

```
// GenCreateEx1.java

import static ch.aplu.util.Console.*;

public class GenCreateEx1
{
    public GenCreateEx1()
    {
        for (int i = 0; i < 5; i++)
        {
            print("0, 1: ");
            println(randomSelect(0, 1)); // Autoboxing

            print("0.0, PI: ");
            println(randomSelect(0.0, Math.PI));

            print("Samstag, Sonntag: ");
            println(randomSelect("Samstag", "Sonntag"));
            println();
        }
    }

    private <T> T randomSelect(T a, T b)
    {
        return Math.random() < 0.5 ? a : b;
    }

    public static void main(String[] args)
    {
```

```

    new GenCreateEx1();
  }
}

```

Eine typische Ausgabe ist:

```

0, 1:          1
0.0, PI:       0.0
Samstag, Sonntag: Samstag

0, 1:          1
0.0, PI:       3.141592653589793
Samstag, Sonntag: Sonntag

0, 1:          0
0.0, PI:       0.0
Samstag, Sonntag: Samstag

0, 1:          0
0.0, PI:       3.141592653589793
Samstag, Sonntag: Sonntag

0, 1:          1
0.0, PI:       3.141592653589793
Samstag, Sonntag: Sonntag

```

Es stellt sich die interessante Frage, welches der Rückgabotyp von `randomSelect()` beim Aufruf mit gemischten Parametertypen ist. Wir schreiben dazu in `GenCreateEx2.java` mit `getClass()` den Rückgabotyp aus.

```

// GenCreateEx2.java

import static ch.aplu.util.Console.*;

public class GenCreateEx2
{
    public GenCreateEx2()
    {
        Object o;
        for (int i = 0; i < 5; i++)
        {
            print("0, 1:          ");
            o = randomSelect(0, 1);
            println(o + "(" + o.getClass() + ")");

            print("0, 1.0:       ");
            o = randomSelect(0, 1.0);

```

```

        println(o + "(" + o.getClass() + ")");

        print("0, Sonntag: ");
        o = randomSelect(0, "Sonntag");
        println(o + "(" + o.getClass() + ")");
        println();
    }
}

private <T> T randomSelect(T a, T b)
{
    return Math.random() < 0.5 ? a : b;
}

public static void main(String[] args)
{
    new GenCreateEx2();
}
}

```

Das Resultat zeigt, dass der Rückgabotyp tatsächlich dem zufällig ausgewählten Parameter entspricht.

```

0, 1:      1(class java.lang.Integer)
0, 1.0:    1.0(class java.lang.Double)
0, Sonntag: Sonntag(class java.lang.String)

```

```

0, 1:      1(class java.lang.Integer)
0, 1.0:    0(class java.lang.Integer)
0, Sonntag: Sonntag(class java.lang.String)

```

```

0, 1:      1(class java.lang.Integer)
0, 1.0:    0(class java.lang.Integer)
0, Sonntag: 0(class java.lang.Integer)

```

```

0, 1:      0(class java.lang.Integer)
0, 1.0:    0(class java.lang.Integer)
0, Sonntag: 0(class java.lang.Integer)

```

```

0, 1:      1(class java.lang.Integer)
0, 1.0:    1.0(class java.lang.Double)
0, Sonntag: Sonntag(class java.lang.String)

```

Generische Methoden lassen sich problemlos überladen, sogar mit einer nicht-generischen Variante. In `GenCreate3.java` schreiben wir, zwar etwas wenig elegant, 4 überladene Versionen mit 0, 1, 2, und 3 Parametern.

```
// GenCreateEx3.java

import static ch.aplu.util.Console.*;

public class GenCreateEx3
{
    public GenCreateEx3()
    {
        print("5.0:          ");
        println(randomSelect(5.0));

        print("Samstag, Sonntag: ");
        println(randomSelect("Samstag", "Sonntag"));

        print("0, 1, 2:          ");
        println(randomSelect(0, 1, 2));

        print("kein Param:       ");
        println(randomSelect());
        println();
    }

    private String randomSelect()
    {
        return "Error using randomSelect()";
    }

    private <T> T randomSelect(T a)
    {
        return a;
    }

    private <T> T randomSelect(T a, T b)
    {
        return Math.random() < 0.5 ? a : b;
    }

    private <T> T randomSelect(T a, T b, T c)
    {
        double r = Math.random();
        if (r < 0.33)
            return a;
        if (r < 0.67)
            return b;
        return c;
    }

    public static void main(String[] args)
```



```
{
    new GenCreateEx3();
}
}
```

Schließlich zeigen wir noch die eleganteste Version ohne jede Codeduplikation unter Verwendung von varargs. Man beachte, dass man für den Fall, wo keine Parameter übergeben werden, den Rückgabewert auf T casten muss.

```
// GenCreateEx4.java

import static ch.aplu.util.Console.*;

public class GenCreateEx4
{
    public GenCreateEx4()
    {
        print("0, 1:          ");
        println(randomSelect(0, 1));

        print("Wochentag:    ");
        println(randomSelect("Montag", "Dienstag", "Mittwoch",
                             "Donnerstag", "Freitag", "Samstag", "Sonntag"));

        print("kein Param:   ");
        println(randomSelect());
        println();
    }

    private <T> T randomSelect(T... a)
    {
        if (a.length == 0)
            return (T)"Error using randomSelect()"; // Must cast

        int i = (int)(a.length * Math.random());
        return a[i];
    }

    public static void main(String[] args)
    {
        new GenCreateEx4();
    }
}
```

Selbstverständlich können generische Methoden mit Zugriffsbezeichner versehen und auch static deklariert werden.

#### 45.6.2.2 Deklaration von generischen Klassen und Interfaces

Die hauptsächliche Absicht bei der Entwicklung von Generics besteht darin, Klassenschablonen zur Verfügung zu stellen, die der Programmierer mit verschiedenen aktuellen Datentypen "ausprägen" kann. Anschaulich kann man sich vorstellen, dass der Compiler vor der Compilation alle formalen Typ-Parameter durch die aktuellen Typen ersetzt und erst nachher die eigentliche Compilation ausführt. Dass in Java nicht aus jeder verwendeten generischen Klasse auch tatsächlich eine eigene Klasse zur Laufzeit erzeugt wird, ist nebensächlich. Typ-Parameter werden bei der Deklaration generischer Klassen in Spitzklammern hinter dem Klassennamen angefügt (**Typdeklarations-Teil**). Diese Parameter sind lediglich Platzhalter (**formale Typ-Parameter**) und können irgendeine Bezeichnung haben, wobei allerdings wie bei Methoden auch hier T, U, V,... üblich ist. Die formalen Typ-Parameter sind im ganzen Deklarationsteil der Klasse sichtbar, sogar in inneren Klassen und überall im Klassen-Deklarationskopf, ausgenommen für statische Instanzvariablen und statische Initialisierungsblöcke, wo sie leider nicht verwendet werden können.

Im ersten Beispiel schreiben wir eine generische Klasse `RandomSelect`, die zwei Instanzvariablen `a` und `b` enthält, von denen eine mit der Methode `get()` zufällig zurückgegeben wird. Der Konstruktor initialisiert die beiden Instanzvariablen.

```
// GenCreateEx5.java

import static ch.aplu.util.Console.*;

class RandomSelect<T>
{
    private T a;
    private T b;

    public RandomSelect(T a, T b)
    {
        this.a = a;
        this.b = b;
    }

    public T get()
    {
        return Math.random() < 0.5 ? a : b;
    }
}

public class GenCreateEx5
{
    public GenCreateEx5()
    {
        println("zufaellig 0, 1: ");
        RandomSelect<Integer> rsInt =
```

```

        new RandomSelect<Integer>(0, 1);
        for (int i = 0; i < 5; i++)
            println(rsInt.get());

        println("\nzufaellig Samstag, Sonntag: ");
        RandomSelect<String> rsString =
            new RandomSelect<String>("Samstag", "Sonntag");
        for (int i = 0; i < 5; i++)
            println(rsString.get());
    }

    public static void main(String[] args)
    {
        new GenCreateEx5();
    }
}

```

Anschaulich kann man sich vorstellen, dass `RandomSelect<Integer>` bzw. `RandomSelect<String>` bestimmte feste Datentypen sind. Oft ist eine generische Klasse nur für einen bestimmten Teil einer Klassenhierarchie sinnvoll und fehlerfrei verwendbar. Es ist daher sinnvoll, dass man gewisse **Einschränkungen (constraints)** an die möglichen aktuellen Typen des Typ-Parameter formulieren kann. Wir spricht in Java davon, dass man den Typ-Parameter an eine Klasse oder ein Interface "**bindet**" (**bound**). Ersetzen wir im Programm `GenCreateEx5.java` den Deklarationskopf durch

```
class RandomSelect<T extends Number>
```

so ergibt sich beim Compilieren eine Fehlermeldung der Art

```
type parameter java.lang.String is not within its bound
```

Im Beispiel `GenCreateEx6.java` zeigen wir hingegen, dass man mit der Bindung

```
class RandomSelect<T extends Turtle>
```

eine aktuelle Klasse `RandomSelect<Kid>` erzeugen kann. (Wir verwenden wieder das Package `homeland`.) Wir speichern darin eine `TurtleBoy`- und ein `TurtleGirl`-Referenz und rufen als Übung zur Polymorphie von der zufällig mit `get()` zurückgegebenen `TurtleKid`-Referenz das entsprechende `shape()` auf.

```

// GenCreateEx6.java

import ch.aplu.turtle.*;
import homeland.*;

class RandomSelect<T extends Turtle> // bound
{
    private T a;
    private T b;
}

```

```
public RandomSelect(T a, T b)
{
    this.a = a;
    this.b = b;
}

public T get()
{
    return Math.random() < 0.5 ? a : b;
}
}

public class GenCreateEx6
{
    public GenCreateEx6()
    {
        RandomSelect<TurtleKid> rsKid =
            new RandomSelect<TurtleKid>(new TurtleBoy(),
                                         new TurtleGirl());

        for (int i = 0; i < 5; i++)
            rsKid.get().shape();
    }

    public static void main(String[] args)
    {
        new GenCreateEx6();
    }
}
```

Auch Interfaces können mit derselben Syntax generisch geschrieben werden. Beispielsweise lässt sich in `RandomSelectI.java` festlegen, dass eine Klasse `get()` implementieren muss.

```
// RandomSelectI.java

interface RandomSelectI<T>
{
    T get();
}
```

In `GenCreateEx6.java` könnte dann der Klassenkopf wie folgt aussehen:

```
class RandomSelect<T extends Turtle>
    implements RandomSelectI<T>
```

Interessanterweise lässt sich `GenCreateEx7.java` compilieren und ausführen, wenn wir die Klasse `RandomSelect` ohne Typ-Parameter verwenden.

```
// GenCreateEx7.java

import ch.aplu.turtle.*;
import homeland.*;

class RandomSelect<T extends Turtle> // bound
{
    private T a;
    private T b;

    public RandomSelect(T a, T b)
    {
        this.a = a;
        this.b = b;
    }

    public T get()
    {
        return Math.random() < 0.5 ? a : b;
    }
}

public class GenCreateEx7
{
    public GenCreateEx7()
    {
        RandomSelect rsBoy = new RandomSelect(new TurtleBoy(),
                                                new TurtleBoy());
        for (int i = 0; i < 5; i++)
            rsBoy.get().forward(100).left(90);

        RandomSelect rsGirl =
            new RandomSelect(new TurtleGirl(),
                            new TurtleGirl());
        for (int i = 0; i < 5; i++)
            rsGirl.get().forward(100).right(90);
    }

    public static void main(String[] args)
    {
        new GenCreateEx7();
    }
}
```

Dabei wird von der generischen Klasse der Raw-type nach folgender Regel verwendet: Der Compiler ersetzt den Typ-Parameter automatisch durch die (erste) beim Bound angegebene Klasse, hier also durch `RandomSelect<Turtle>`. Ist kein Bound angegeben, so wird für den Raw-type die Klasse `Object` eingesetzt. Dadurch erhält man eine weitgehende Kompatibilität des Raw-types zu den früheren nicht-generischen Klassen.

## 45.7 Kovarianz beim Überschreiben (covariant return)

Soll in einer Klassenhierarchie eine Methode überschrieben werden, so muss die Signatur (der Methodenname und die Parameterliste, d.h. die Anzahl und der Typ der Parameter) exakt übereinstimmen. In Java bis zu J2SE V5 muss zudem auch der Rückgabetyt übereinstimmen, ansonsten ein Fehler zu Compilationszeit angezeigt wird.

In J2SE V5 wird die Bedingung an den Rückgabetyt abgeschwächt. Dieser kann auch ein Subtyp des Rückgabetyts der zu überladenden Methode sein. (Man nennt dieses Verhalten **covariant return**.)

Folgendes Beispiel zeigt, dass sich diese Änderung positiv auswirken kann. Die meisten Methoden der Klasse `Turtle` geben eine `Turtle`-Referenz zurück, wodurch sich die Methoden kaskadieren lassen, beispielsweise mit der `Turtle`-Referenz `john`:

```
john.forward(100).left(90).forward(100);
```

Wenn wir eine aus der Klasse `Turtle` abgeleitete Klasse `LazyTurtle` deklarieren und die Methode `forward()` derart überschreiben möchten, dass die `LazyTurtle` sich statt auf einer Geraden auf einer Zickzack-Linie bewegt, so mussten wir ohne *covariant return* leider eine Referenz auf `Turtle` zurückgeben. Nun kann, wie in folgendem Beispiel gezeigt, `forward()` auch den Datentyp `LazyTurtle` zurückgeben.

```
// LazyTurtle.java

import ch.aplu.turtle.*;

public class LazyTurtle extends Turtle
{
    public LazyTurtle forward(double distance)
    {
        left(30);
        super.forward(0.577350 * distance);
        right(60);
        super.forward(0.577350 * distance);
        left(30);
        return this;
    }

    public LazyTurtle dance()
    {
```

```
        for (int i = 0; i < 10; i++)
            left(36);
        return this;
    }
}
```

Wie wir sehen, besitzt eine `LazyTurtle` zudem noch die Fähigkeit, mit der Methode `dance()` eine Pirouette zu tanzen. Die positiven Auswirkungen des *covariant return* zeigen sich am Beispielprogramm `OverEx1.java`, wo wir die Methoden von `LazyTurtle` problemlos kaskadieren können, selbst wenn wir die Methode `dance()` verwenden, die nur in `LazyTurtle`, aber nicht in `Turtle` deklariert ist.

```
// OverEx1.java

import ch.aplu.turtle.*;
import java.awt.Color;

public class OverEx1
{
    public OverEx1()
    {
        LazyTurtle lt = new LazyTurtle();
        lt.forward(100).dance().left(90).forward(100);

        // Zum Vergleich
        Turtle t = new Turtle(lt, Color.red);
        t.forward(100).left(90).forward(100);
    }

    public static void main(String[] args)
    {
        new OverEx1();
    }
}
```

Hätten wir in `forward()` eine `Turtle`-Referenz zurückgeben müssen, so würde die Zeile `lt.forward(100).dance().left(90).forward(100);`

unweigerlich zu einer Fehlermeldung führen, da `dance()` für `Turtles` nicht deklariert ist. Wegen der Polymorphie verwendet auch der zweite Aufruf von `forward()` in dieser Zeile die überschriebene Methode aus der Klasse `LazyTurtle`, obschon `left(90)` eine `Turtle`-Referenz zurückgibt.