Java – Eine Übersicht

Roland Schätzle[1]

Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB)

Universität Karlsruhe (TH)

1 Einführung

Kaum eine Programmiersprache hat so kurz nach ihrer Einführung soviel Aufmerksamkeit auf sich gezogen und eine so massive Unterstützung von Seiten unterschiedlicher Softwarehersteller bekommen wie Java [Flan96, GJS96, Java97].

Etwas verständlicher wird dies, wenn man betrachtet, daß hinter dem Begriff Java eigentlich mehr steckt, als nur eine Programmiersprache: Zum einen eine Idee; nämlich die einer Sprache, die in einen plattformneutralen Code übersetzt, in HTML-Seiten eingebettet über das Internet übertragen werden kann, um dann in einem Web-Browser mit Hilfe eines Interpreters ausgeführt werden zu können. Die Firma Sun bringt dies mit dem Werbeslogan „write once, run anywhere“ auf eine prägnante Formel. Zum anderen gehört zu Java eine inzwischen recht umfangreiche Sammlung von quasi standardisierten Klassenbibliotheken, Programmierschnittstellen (APIs) und Anwendungsarchitekturen, die von der Programmierung von graphischen Benutzungsschnittstellen über Electronic-Commerce, Embedded-Systems bis hin zum Betriebssystem ein weites Feld abdecken.

Darum soll es in diesem Artikel aber nicht gehen. Hier soll eine kurze Einführung in die wesentlichen Eigenschaften von Java als Programmiersprache vermittelt werden. Beim Leser werden Kenntnisse in einer anderen Programmiersprache, vornehmlich C oder C++, vorausgesetzt. Am Schluß findet noch ein Vergleich mit anderen objektorientierten Programmiersprachen (Eiffel, C++, Smalltalk) statt, der die Stärken und Schwächen von Java näher beleuchtet.

2 Eine einfache, sichere und portable Sprache

Die Syntax von Java ist stark an C bzw. C++ angelehnt. Gegenüber diesen Sprachen ist Java aber wesentlich *einfacher* und *schlanker*. Dies ist zum einen darauf zurückzuführen, daß Java eine rein objektorientierte Sprache ist, während C++ einen hybriden Ansatz verfolgt, um mit C kompatibel zu sein. D.h. Elemente wie Records (`struct`), variante Records (`union`), und benutzerdefinierte Typen (`typedef`) sind in Java nicht vorhanden, da sie durch Klassen ausgedrückt werden können.

Zum anderen sollte die Programmierung in Java auch *weniger fehleranfällig* sein, weshalb es keine Zeiger bzw. Zeigerarithmetik, keine manuelle Speicherverwaltung und keinen Makroprozessor gibt.

[1] Dipl.-Inform. Roland Schätzle, Institut AIFB, Universität Karlsruhe (TH), 76128 Karlsruhe, e-Mail: schaetzle@aifb.uni-karlsruhe.de

Die dritte Designleitlinie beim Entwurf von Java war *Portabilität*. Deshalb sind die Wortbreite und der Wertebereich der eingebauten Datentypen fest vorgegeben und nicht plattformabhängig. Auch die Auswertungsreihenfolge von Ausdrücken, die in C in einigen Bereichen dem Compilerbauer überlassen bleibt, ist durchgängig festgelegt.

2.1 Kontrollstrukturen

Die Kontrollstrukturen für Schleifen und bedingte Verzweigungen wurden fast identisch aus C übernommen. Der einzige Unterschied liegt bei Ausdrücken, die einen Boole'schen-Wert zum Ergebnis haben. In C/C++ gibt es keinen Datentyp zur Darstellung von Wahrheitswerten, weshalb der ganzzahlige Wert 0 als logisch falsch und jeder andere ganzzahlige Wert als logisch wahr interpretiert wird. Diese mangelnde Trennung von eigentlich unterschiedlichen Datentypen wurde von vielen Programmierern zur Formulierung mehr oder weniger obskurer Ausdrücke ausgenutzt. Dies ist in Java nicht mehr möglich, da der Datentyp `boolean` die explizite Darstellung der Werte `true` und `false` erlaubt und an entsprechenden Stellen, wie z.B. bei Abbruchbedingungen von Schleifen, auch erforderlich ist.

2.1.1 Schleifen

Die folgenden Programmbeispiele zeigen jeden, der in Java vorhandenen drei Schleifentypen; es werden jeweils die Zahlen von 1 bis 10 aufaddiert.

```
int i = 1;
int sum = 0;

while (i <= 10) do {
    sum = sum + i;
    i = i + 1;
}
```

```
int i = 1;
int sum = 0;

do {
    sum = sum + i;
    i = i + 1;
} while (i <= 10);
```

```
int i;
int sum = 0;

for (i = 1; i <= 10; i = i + 1) {
    sum = sum + i;
}
```

2.1.2 Bedingte Verzweigungen

Nachstehender Programmcode zeigt ein Beispiel für die bedingte Verzweigung mit dem `if/else`-Befehl bzw. dem `switch`-Statement. Beim `switch`-Statement ist nach der Abarbeitung eines Falles, wie in C, ein `break`-Befehl notwendig, damit der Block verlassen wird und nicht auch noch die anderen Fälle ausgeführt werden.

```
int a, b, max;
…
if (a > b)
    max = a
else
    max = b;
```

```
char c;
int z;
…
switch (c) {
  case ‘a’: z = 1;
      break;
  case ‘b’: z = 2;
      break;
  case ‘c’: z = 3;
      beak;
}
```

2.1.3 Unbedingte Verzweigungen

Der `goto`-Befehl zur Durchführung unbedingter Sprünge existiert in Java nicht. Allerdings kann mit `continue` innerhalb einer Schleife der nächste Schleifendurchgang vorzeitig veranlaßt werden und mit `break` kann der aktuelle Block verlassen werden. Beide Befehle können um eine Sprungmarke ergänzt werden.

2.2 Einfache Datentypen

Folgende Tabelle listet die in Java vorkommenden einfachen Datentypen auf. Im Unterschied zu C/C++ gibt es einen Boole'schen Datentyp (`boolean`) und alle Integerwerte sind vorzeichenbehaftet. Die Wortbreite und der Wertebereich aller einfachen Datentypen sind aus Gründen der Portabilität für alle Plattformen verbindlich vorgegeben. Der Datentyp `char` kann alle Zeichen des Unicode-Zeichensatzes darstellen und hat deswegen eine Wortbreite von 16 Bit. Die Darstellung der Gleitkommazahlen entspricht der Norm IEEE 754.

Typ	Inhalt	Größe	Min	Max
boolean	true, false			
char	Unicode	16 Bit	Hex 0000	Hex FFFF
byte	signed integer	8 Bit	-128	127
short	signed integer	16 Bit	-32768	32767
int	signed integer	32 Bit	-2.147.483.648	2.147.483.647
long	signed integer	64 Bit	-2^{63}	$2^{63} - 1$
float	floating point	32 Bit	$\pm m\ 2^{-149}$ ($m < 2^{24}$)	$\pm m\ 2^{104}$
double	floating point	64 Bit	$\pm m\ 2^{-1054}$ ($m < 2^{53}$)	$\pm m\ 2^{1000}$

2.3 Operatoren

Die Operatoren stimmen im Wesentlichen mit denen von C überein. Neu hinzugekommen ist der vorzeichenlose Right-Shift (**>>>**); dieser Operator betrachtet die eigentlich vorzeichenbehafteten Integerwerte von Java bei der Schiebeoperation als vorzeichenlos. Bei den logischen Operatoren &, ^ und | ist zu beachten, daß es sich bei ganzzahligen Argumenten um Bitoperationen handelt und bei Boole'schen Argumenten um logische Operationen. Untenstehende Tabelle gibt eine Übersicht über sämtliche Operatoren geordnet nach deren Bindungsstärke.

Bindung	Operator	Beschreibung
1	++, ––	Inkrement-, Dekrement-Operator
	+, –	Vorzeichen
	~	Bitweise Negation
	!	Logische Negation
	(type) expr	Cast (Typumwandlung)
2	*, /	Multiplikation, Division
	%	Restbildung
3	+, –	Addition, Subtraktion
	+	Verkettung von Strings
4	<<, >>	Left-/Right-Shift
	>>>	Vorzeichenloser Right-Shift
5	<, >, <=, >=	Vergleichsoperatoren
	==, !=	Gleichheit/Ungleichheit
6	&	UND
7	^	XOR
8	|	ODER
9	&&	bedingtes UND (logisch)
10	||	bedingtes ODER (logisch)
11	?:	bed. Operator
12	=	einfache Zuweisung
	*=, /=, +=, –=, <<=, >>=, >>>=, &=, ^=, !=	Zuweisung mit Operation

3 Eine objektorientierte Sprache

In Java ist das hauptsächliche Strukturierungskonzept die Klasse. Ein Java-Programm wird durch eine Ansammlung von Klassen gebildet. Klassen enthalten Attribute (Daten) und Methoden (Operationen). Dies bedeutet insbesondere, daß es in Java, im Gegensatz zu C++, keine alleinstehenden Funktionen gibt (eine Funktion tritt nur innerhalb einer Klasse auf) und auch keine globalen Variablen. Zur Laufzeit können zu jeder Klasse beliebig viele Instanzen (Objekte) gebildet werden.

3.1 Klassen

Das folgende Beispiel zeigt die Klasse `Point`, die Punkte im zweidimensionalen Raum repräsentiert. Sie besteht aus zwei Attributen, nämlich den `x`- und `y`-Koordinaten und Operationen zum Verschieben und zur Berechnung der Distanz zu einem anderen Punkt.

```
class Point {
    int x;
    int y;

    void move (int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
    double dist (int otherX, int otherY) {
        int dx = otherX - x;
        int dy = otherY - y;
        return (Math.sqrt ((dx * dx) + (dy * dy)));
    }
}
```

3.2 Objekte

Klassen sind benutzerdefinierte Datentypen und können wie eingebaute Datentypen verwendet werden. Variablen, die Objekte enthalten, werden auf die selbe Weise deklariert wie andere Variablen, d.h. genauso wie durch

```
int x;
```

eine Integer-Variable deklariert wird, kann mit

```
Point p;
```

eine Variable vereinbart werden, die ein Objekt vom Typ `Point` enthält. Allerdings erzeugt diese Vereinbarung noch kein Objekt. Dies geschieht mit Hilfe des `new`-Operators:

```
p = new Point();
```

Durch diese Operation wird ein Objekt vom Typ `Point` erzeugt und für dieses Speicherplatz zur Verfügung gestellt.

3.2.1 Konstruktor-Methoden

Bei der Erzeugung des `Point`-Objekts im obigen Beispiel wird den Attributen `x` und `y` der Vorgabewert 0 zugewiesen. Möchte man bei der Objekterzeugung eine anwendungsspezifische Initialisierung durchführen, dann muß eine sogenannte Konstruktormethode für die betreffende Klasse formuliert werden. Für die Klasse `Point` könnte dies eine Methode sein, die Koordinaten des zu erzeugenden Punktes mit benutzerdefinierten Anfangswerten belegt. Der folgende Programmcode zeigt die entsprechende Erweiterung von `Point`:

```
class Point {
    Point (int x0, int y0) {
        x = x0;
        y = y0;
    }
    // Rest wie oben ...
}
```

Somit kann ein Punkt mit den Koordinaten (4, 5) auf folgende Art erzeugt werden:

```
p = new Point(4, 5);
```

Konstruktormethoden haben immer denselben Namen wie die zugehörige Klasse. Zu einer Klasse kann es mehrere Konstruktormethoden geben, die sich aber in den Argumenten (im Typ oder der Anzahl der Argumente) unterscheiden müssen.

3.2.2 Wertesemantik und Referenzsemantik

Bei der Verwendung von Variablen eines Klassentyps muß beachtet werden, daß diese im Gegensatz zu einfachen Datentypen (`int`, `char`, usw.) Referenzsemantik besitzen, d.h. eine Variable vom Typ einer Klasse enthält nicht ein Objekt, sondern stellt nur eine Referenz auf dieses dar. Folgendes Beispiel soll diesen Sachverhalt verdeutlichen:

```
int a, b;

a = 4;
b = a;

a = a + 1;
```

```
Point p1, p2;

p1 = new Point(10, 3);
p2 = p1;

p1.move (5, 5);
```

Im linken Beispiel hat `b` nach der Zuweisung `b = a` denselben Wert wie `a`, nämlich 4. Durch die Addition in der nachfolgenden Zeile wird `a` um Eins erhöht, `b` bleibt durch die Operation aber unverändert, da es eine *Kopie* des Wertes aus `a` enthält.

Ganz anders im rechten Beispiel: Dort enthält `p2` nach der Zuweisung `p2 = p1` auch den Punkt (10, 3). Da es sich aber um eine Referenz handelt, d.h. da sowohl `p1` als auch `p2` eigentlich nur Zeiger auf dasselbe Objekt sind, betrifft die Verschiebeoperation in der letzten Zeile beide Variablen. Danach ist `p1.x` = `p2.x` = 15 und `p1.y` = `p2.y` = 8 (= wird hier im Sinne von Gleichheit benutzt).

3.2.3 Garbage Collecting

Wenn ein Objekt im Laufe der Programmausführung nicht mehr gebraucht wird, d.h. wenn keine Referenz mehr auf dieses Objekt existiert, dann sollte der benutzte Speicherplatz wieder freigegeben werden.

In C++ bleibt diese Aufgabe dem Programmierer überlassen. Er muß den Zeitpunkt kennen, zu dem das Objekt freigegeben werden kann und muß dies durch einen entsprechenden Funktionsaufruf explizit veranlassen. Diese Aufgabe ist sehr fehleranfällig. Oft werden Objekte zu früh freigegeben, was zu schwer aufzufindenden Speicherfehlern führt, oder die Objekte werden gar nicht oder zu spät freigegeben, was zu einem nicht unerheblichen Speicherplatzverbrauch führt.

Java entlastet den Programmierer durch den Garbage Collector von dieser Aufgabe. Der Garbage Collector ist ein Mechanismus des Laufzeitsystems, der selbständig erkennt, wann ein Objekt nicht mehr benötigt ist und dieses dann freigibt.

3.2.4 Arrays

Im Gegensatz zu C/C++ sind Arrays in Java nicht einfach nur Speicherbereiche, auf die mit Hilfe von Zeigern zugegriffen werden kann, sondern es handelt sich um vollwertige Objekte. Ein Array ist in Java also eine dynamische Datenstruktur, die zur Laufzeit erzeugt wird. Zum Erzeugungszeitpunkt wird dann die Größe ei-

nes Arrays angegeben und nicht schon bei seiner Deklaration. Die Größe kann jederzeit über das Attribut `length` abgefragt werden. Da ein Array ein Objekt ist, können Zugriffe darauf kontrolliert werden, was bedeutet, daß in Java die Indexwerte auf Einhaltung der Bereichsgrenzen geprüft werden. Arrays sind in Java eindimensional; analog zu C/C++ werden mehrdimensionale Arrays durch Verschachtelung (d.h. Arrays von Arrays) dargestellt.

Es gibt zwei verschiedene Schreibweisen zur Deklaration eines Array. Folgendes Beispiel zeigt die Vereinbarung eines Array von Integerwerten auf beide Arten:

```
int[] ai;
int ai[];
```

Das nächste Beispiel zeigt eine Methode, die ein zweidimensionales Array von Bytes erzeugt und mit einem Anfangswert füllt:

```
byte[][] makeByteArray (int n, int m, byte initVal)
{
    byte[][] newArray = new byte[n][m] // das Array wird erzeugt
    for (int i = 0; i < newArray.length; i++)
        for (int j = 0; j < newArray[i].length; j++)
            newArray[i][j] = initVal;
    return (newArray);
}
```

3.2.5 Strings

Auch bei den Strings hat sich Java von seiner C/C++-Herkunft gelöst. Strings sind in Java keine Arrays, die Zeichen enthalten, sondern es handelt sich um eine vollwertige Klasse, von der Objekte erzeugt werden können. Ein String wird beispielsweise mit

```
String s;
```

deklariert und kann mit dem new-Operator erzeugt und initalisiert werden:

```
s = new String("abc")
```

Zur Initialisierung wird auch eine vereinfachende Schreibweise angeboten:

```
s = "abc"
```

Mit der Methode `length()` kann die Länge eines Strings abgefragt werden und mit `concat()` bwz. `+` lassen sich zwei Strings verketten. Außerdem bietet die Klasse `String` eine Vielzahl von Methoden zur Suche von Teilstrings, zum Vergleich und für andere Stringbearbeitungsoperationen zur Verfügung [Stri97].

3.3 Vererbung

Java unterstützt als objektorientierte Sprache natürlich auch das Konzept der Vererbung, d.h. aus bestehenden Klassen können neue Klassen abgeleitet werden, welche sämtliche Attribute und Methoden der Oberklasse erben. In der abgeleiteten Klasse können weitere Attribute und Methoden hinzugefügt werden und die Implementierung geerbter Klassen kann durch eine andere ersetzt werden (overriding).

3.3.1 Einfaches Erben von Klassen

Es könnte beispielsweise eine Klasse zur Darstellung von farbigen Punkten basierend auf der oben gezeigten Klasse `Point` formuliert werden. Ein farbiger Punkt ist ein Punkt (`Point`) mit einem zusätzlichen Attribut, nämlich der Farbe.

```
class ColoredPoint extends Point {
    Color aColor;   // die Farbe des Punktes
}
```

3.3.2 Interfaces

Java unterstützt nur einfache Vererbung, d.h. eine Klasse kann nur die Attribute und Methoden von genau einer Oberklasse erben. Echte Mehrfachvererbung, also das erben von mehreren Oberklassen ist nur in einer vereinfachten Form möglich.

Zu diesem Zweck werden sogenannte Interfaces bereitgestellt. Dies sind im Prinzip Klassen, die nur die Signaturen von Methoden (also Name und Parameterliste) definieren, aber keine Implementierung besitzen. Man kann damit festlegen, daß jede Klasse, die von einem solchen Interface erbt, alle dort vorkommenden Operationen anbietet. Bei Interfaces ist auch Mehrfachvererbung erlaubt, d.h. eine Klasse kann die Eigenschaften mehrerer Interfaces erben.

Das folgende Beispiel für ein Interface stammt aus der Objectspace-Bibliothek (www.objectspace.com). Dies ist eine frei verfügbare Java-Klassenbibliothek für Datenstrukturen aller Art (z.B. Listen, Stacks, Mengen). Jede der dort vorkommenden Datenstrukturen bietet einen Grundstock von Operationen an, die im Interface `Container` festgelegt sind. Aus diesem Grund erbt jede dieser Klassen von Container und verpflichtet sich dadurch, die dort aufgelisteten Operationen zu implementieren. Das Schlüsselwort, das die Vererbung von Interfaces anzeigt heißt deswegen auch `implements` und nicht `extends`, wie bei der Klassenvererbung.

```
public interface Container extends Cloneable, Serializable {
  // Add an object to myself.
  public Object add( Object object );

  // Remove all of my objects.
  public void clear();

  // Return true if I contain no objects.
  public boolean isEmpty();

  // Return an Enumeration of the components in this container
  public Enumeration elements();

  // Return an iterator positioned at my first item.
  public ForwardIterator start();

  // Return an iterator positioned after my last item.
  public ForwardIterator finish();

  // Return true if I'm equal to a specified object.
  public boolean equals( Object object );

  // Return the number of objects that I contain.
  public int size();

  // Return the maximum number of objects that I can contain.
  public int maxSize();

  // Return a shallow copy of myself.
```

```
    public Object clone();

    // Return a string that describes me.
    public String toString();
}
```

3.4 Abstrakte Klassen

Für manche Vererbungshierarchien möchte man eine Oberklasse definieren, die – ähnlich wie bei einem Interface – festlegt, welche Operationen (und auch Attribute) die Unterklassen anbieten sollen. Einige dieser Operationen können schon in der Oberklasse definiert werden, andere benötigen in jeder Unterklasse eine spezielle Implementierung und können deshalb in der Oberklasse noch nicht ausformuliert werden.

Zu diesem Zweck bietet Java das Konzept der abstrakten Klasse an. Dies sind Klassen, die Methoden ohne Implementierung, eben abstrakte Methoden, enthalten. Die betreffenden Klassen und Methoden werden mit dem Schlüsselwort `abstract` gekennzeichnet.

3.5 Einkapselung und Packages

Einkapselung in objektorientierten Sprachen bedeutet, daß Klassen einen Teil ihrer Attribute und Methoden vor dem Zugriff anderer Klassen schützen können. Auf diese Weise bleiben Implementierungsdetails nach außen verborgen und man erreicht eine Entkopplung von Implementierung und Schnittstelle.

3.5.1 Packages

Java stellt mit den Packages neben den Klassen noch ein weiteres Strukturierungselement zur Verfügung, so daß die Einkapselung über mehrere Stufen realisiert werden kann. Ein Package ist eine Sammlung von (thematisch zusammengehörenden) Klassen. Durch

```
package AIFB.graphics;
class Point { …
```

wird z.B. die Zugehörigkeit der Klasse `Point` zum Package `AIFB.graphics` deutlich gemacht. Mit

```
import java.lang.*;
class Point { …
```

wird angegeben, daß die Klasse `Point` alle Klassen (`*`) aus dem Package `java.lang` benutzen möchte.

3.5.2 Sichtbarkeitsbereiche

Welche Klasse, wo sichtbar ist und wer auf welche Elemente einer Klasse zugreifen darf, wird durch die verschiedenen Sichtbarkeitsbereiche in Java geregelt.

Wenn eine Klasse oder ein Interface global sichtbar sein soll, also auch außerhalb des Package, in dem sie/es definiert wurde, dann muß dem Schlüsselwort `class` bzw. `interface` der Modifier `public` vorangestellt werden. Ohne die-

sen Zusatz ist eine Klasse bzw. ein Interface nur innerhalb des eigenen Package bekannt (Sichtbarkeitsbereich „package“).

Der Sichtbarkeitsbereich der Attribute und Methoden kann noch differenzierter angegeben werden. Auch hier bedeutet das Schlüsselwort `public` globale Sichtbarkeit (was natürlich nur für Elemente einer globalen Klasse Sinn macht). Etwas mehr Einschränkung bietet `protected`; mit diesem Modifier gekennzeichnete Elemente sind nur innerhalb des eigenen Package oder in Klassen, die von der betreffenden Klasse erben (und außerhalb dieses Package sein können) sichtbar.

Wie für Klassen gibt es auch für deren Elemente den Sichtbarkeitsbereich „package“, d.h. solche Elemente sind nur im eigenen Package sichtbar. Dies betrifft alle Elemente ohne speziellen Modifier. Die stärkste Einschränkung schließlich bietet das Schlüsselwort `private`. Damit gekennzeichnete Elemente sind nur innerhalb ihrer Klasse bekannt, nicht einmal erbende Klassen könne auf solche Attribute oder Methoden zugreifen.

3.6 Generische Klassen

Das Array ist in Java der einzige generische Datentyp, d.h. man kann Arrays von beliebigen Datentypen (einfache Typen wie z. B. `int`, `float`, `char` oder auch benutzerdefinierte Klassen wie `Point`) bilden. Die Sprache bietet dem Programmierer aber nicht die Möglichkeit, eigene generische Datenstrukturen zu erstellen.

Wenn man in Java trotzdem eine ähnliche Funktionalität realisieren möchte, wenn man also z.B. einen Stack implementieren möchte, der beliebige Elemente aufnehmen kann, dann wird dies wie folgt realisiert:

```
public class Stack {
    protected Object[] inhalt;
    protected int topIndex = 0;    // der nächste freie Eintrag

    /** Initialisiert einen neuen Stack, indem
        ein Array der Größe 'groesse' erzeugt wird */
    public Stack (int groesse) {   // der Konstruktor
        inhalt = new Object[groesse];
    }

    /** Gibt das oberste Element des Stacks zurück */
    public Object top() {
        return (inhalt[topIndex -1]);
    }

    /** Entfernt das oberste Element vom Stack */
    public void pop() {
        --topIndex;
    }

    /** Legt 'newElement' auf den Stack */
    public void push(Object newElement) {
        inhalt[topIndex] = newElement;
        ++topIndex;
    }
```

```
    /** Ist der Stack leer? */
    public boolean empty() {
        return (topIndex == 0);
    }

    /** Ist der Stack voll? */
    public boolean full() {
        return(topIndex == inhalt.length);
    }
}
```

In obigem Beispiel wurde ein Stack so implementiert, daß er Elemente vom Typ `Object` aufnehmen kann. `Object` ist eine Java-Klasse, von der alle anderen Klassen erben. Das bedeutet unter anderem, daß jede Variable vom Typ `Object` Objekte jeden anderen (Klassen-)Typs aufnehmen kann. Auf diese Weise können Datenstrukturen zur Aufnahme beliebiger Objekte realisiert werden.

Für den Programmierer ist es allerdings nicht möglich zu deklarieren, daß ein Stack z.B. nur Objekte vom Typ `Point` aufnehmen soll; das können nur echte generische Klassen leisten. Ein und derselbe Stack kann hier Objekte unterschiedlichen Typs enthalten.

Das folgende Codefragment zeigt, wie der `Stack` benutzt werden kann:

```
Stack s = new Stack(10);
Point p;

s.push(new Point(5, 8));
p = (Point) s.pop();
```

Es wird zunächst ein Stack der Größe 10 erzeugt. Auf diesem wird dann ein Punkt mit den Koordinaten (5, 8) abgelegt und in der folgenden Zeile wieder vom Stack entfernt und der Variablen `p` zugewiesen. Hier wird eine weitere Unzulänglichkeit der gezeigten Implementierung deutlich: Da `pop()` als Ergebnis Objekte vom Typ `Object` liefert, müssen diese mit Hilfe eines Cast-Operators wieder in ihren ursprünglichen Typ (hier: `Point`) umgewandelt werden.

Eine weitere Einschränkung besteht darin, daß die einfachen Datentypen, wie z.B. `int`, `char` und `float`, in Java keine Objekte sind! Das bedeutet, daß obiger Stack Werte mit einfachem Datentyp gar nicht aufnehmen kann.

Als „Lösung“ bietet Java zu jedem einfachen Datentyp eine korrespondierende Klasse an (z.B. zum einfachen Typ `int` die Klasse `Integer`), so daß auch solche Werte als Objekte einer Klasse behandelt werden können. Folgendes Listing zeigt, wie auf diese Weise ein Stack Integerzahlen aufnehmen kann.

```
Stack si;
Integer io;
int i;

si = new Stack(10);  // ein Stack der Kapazität 10 wird erzeugt

// die Zahlen 3, 8 und 23 werden auf den Stack gelegt

si.push(new Integer(3));
si.push(new Integer(8));
si.push(new Integer(23));
```

```
// 23 wird vom Stack geholt, in ein Objekt vom Typ 'Integer'
// zurückgewandelt und schließlich wird der einfache Wert
// vom Typ 'int' extrahiert

io = (Integer) si.top();
i = io.intValue();
```

4 Weitere Eigenschaften

Die folgenden Abschnitte beschreiben weitere Eigenschaften und Elemente, die in Java vorhanden sind. Im Einzelnen handelt es sich um die Fehlerbehandlung, Multithreading, die Dokumentationsmöglichkeiten von Kommentaren und schließlich wird noch der Unterschied zwischen Applets und Applications erläutert.

4.1 Exception-Handling

Java besitzt Konzepte und Mechanismen zur Behandlung von Fehlersituationen. Ein Fehler, der in einem Java-Programm auftritt, kann durch eine Fehlerbehandlungsroutine abgefangen werden. Dadurch wird vermieden, daß das Programm in Ausnahmesituationen einfach abstürzt.

Der Code, bei dessen Abarbeitung ein Fehler erwartet wird, muß zum Zwecke der Fehlerbehandlung in einen sogen. `try`-Block eingeschlossen werden. Eine nachfolgende `catch`-Klausel enthält dann die Fehlerbehandlungsroutine. In nachstehendem Beispiel kann im Ausdruck `r = i/j` eine Division durch Null auftreten. Für diesen Fall möchte man den Benutzer durch eine entsprechende Meldung informieren, was im `catch`-Block geschieht.

```
int i,j,r;

try {
    r = i / j;
}
catch (ArithmeticException e) {
    System.out.println (e.toString());
}
```

In realistischen Programmen wird die Fehlerbehandlung natürlich etwas umfangreicher ausfallen als im obigen Beispiel.

Immer, wenn ein Fehler auftritt, erzeugt das Laufzeitsystem von Java ein Objekt vom Typ `Throwable`, welches eine genauere Beschreibung des Fehlers enthält. Für viele Fehlersituationen gibt es Unterklassen von `Throwable`, die die speziellen Eigenschaften bestimmter Fehler detaillierter beschreiben. Es gibt zwei wichtige Unterklassen von `Throwable`: Zum einen die Klasse `Error`, deren Erben bei schwerweigenden Fehlern erzeugt werden. Dies sind Fehler, die von der Anwendung nicht behandelt werden können bzw. nicht behandelt werden sollten. Die andere Unterklasse von `Throwable` ist `Exception`. Solche Fehler können Abgefangen werden. `Exception` hat wiederum eine Unterklasse, nämlich `RuntimeException`. Bei dieser Fehlerart verlangt Java sogar, daß eine Fehlerbehandlung durchgeführt wird.

Zu einer `try`-Klausel kann es mehrere `catch`-Blöcke geben, von denen jeder auf eine ganz bestimmte Fehlerart, die im `try`-Block auftritt, reagiert. Im obigen Beispiel reagiert die `catch`-Klausel nur auf Fehler vom Typ `ArithmeticException`.

4.2 Multithreading

Java bietet ein in die Sprache integriertes Konzept zur Programmierung von parallelen leichtgewichtigen Prozessen (Threads) mit entsprechenden Synchronisationsmechanismen. Da die Behandlung dieses Themas den Rahmen des Artikels sprengen würde, wird auf [Sure97] verwiesen, wo auf ca. 25 Seiten eine verständliche Einführung und Übersicht zu dieser Thematik geboten wird.

4.3 Kommentare

Es gibt drei verschiedene Arten von Kommentaren in Java-Programmen. Zum einen die aus C bekannten Blockkommentare, die über mehrere Zeilen reichen können:

```
/* ein mehrzeiliger Java-Kommentar,
   der aus C bekannt sein dürfte */
```

Als nächstes sind die aus C++ übernommenen Zeilenkommentare zu erwähnen, die mit `//` eingeleitet werden und jeweils bis zum Zeilenende reichen.

Die dritte und neue Möglichkeit sind sogen. Dokumentationskommentare. Sie sind syntaktisch den C-Kommentaren ähnlich, werden aber durch `/**` statt `/*` eingeleitet. Solche Kommentare werden von einem speziellen Werkzeug (javadoc), das aus dem Java-Quellcode eine Schnittstellendokumentation in HTML- oder FrameMaker-Format extrahieren kann, erkannt.

Zur Benutzung dieser Kommentare gibt es bestimmte Konventionen. So soll zu Beginn jeder Klasse ein solcher Kommentar die Klasse kurz charakterisieren und vor jedem Attribut und vor jeder Methode soll eine kurze Beschreibung des betreffenden Elements erscheinen.

Innerhalb der Dokumentationskommentare können noch verschiedene Schlüsselworte, die mit `@` eingeleitet werden, benutzt werden. Im folgenden Beispiel wurden diese Schlüsselworte verwendet.

```
/** Ein Dokumentationskommentar

    @author Herbert Hillgruber
    @version 1.1

    mit Querverweis zu anderen Programmteilen
    @see String

    oder sogar mit HTML-Links
    @see <a href="spec.html"> Java Spec</a>

    Funktionsparameter
    @param file – the file to be searched

    Rückgabewerte
    @return Das Maximum der Eingabewerte

    Klassen/Methoden oder Attribute, die in einer
    neuen Bibliotheksversion nicht mehr benutzt werden
    sollen, können als veraltet deklariert werden

    @deprecated Benutzen Sie isEmpty() statt empty()
*/
```

Dokumentationswerkzeuge, wie das oben genannte „javadoc“, können diese Schlüsselworte zur besseren Formatierung oder zur Erzeugung von Querverweisen verwenden. Das `@deprecated`-Schlüsselwort wird direkt vom Compiler verwen-

det, um den Benutzer vor der Verwendung von veralteten Bibliothekselementen zu warnen.

4.4 Applets/Applications

Java-Programme lassen sich in zwei Kategorien einteilen. Zum einen kann man mit Java ganz „gewöhnliche“ Anwendungen schreiben, die mit einem Compiler übersetzt werden und dieselben Möglichkeiten haben, wie jedes andere in einer beliebigen Sprache implementierte Programm. In diesem Fall spricht man von einer Java-Application.

Der andere und bei Java vielleicht bekanntere Fall sind Programme, die in den plattformunabhängigen Java-Bytecode übersetzt und auf einem Web-Server abgelegt werden. Von dort können sie – eingebettet in eine HTML-Seite – von einem beliebigen (Java-fähigen) Web-Browser aufgerufen werden. Der betreffende Browser hat einen Interpreter für den Java-Bytecode eingebaut (deshalb ist er Java-fähig) und kann das via Internet übertragene Programm lokal ausführen. Diese Java-Programme nennt man Applets. Sie unterscheiden sich von Applications hauptsächlich durch ihre aus Sicherheitsgründen eingeschränkten Möglichkeiten. Sie dürfen auf dem lokalen Rechner z.B. nicht auf das Dateisystem zugreifen und auch nicht drucken.

5 Vergleich

Um die Eigenschaften von Java besser einschätzen zu können, wird die Sprache im folgenden mit verschiedenen anderen objektorientierten Programmiersprachen verglichen. Dies sind im einzelnen Eiffel, C++ und Smalltalk. Die beiden letztgenannten aus Gründen der Verbreitung; es handelt sich wohl um die im Moment meistbenutzten objektorientierten Sprachen.

Der Vergleich mit Eiffel [Meye92] hat einen anderen Grund. B. Meyer hat in [Meye88] bzw. [Meye97] eine Liste von fundierten und begründeten Anforderungen an eine moderne objektorientierte Programmiersprache, die den Erfordernissen des Software-Engineering genügen soll, aufgestellt und in Eiffel realisiert. Deshalb kann Eiffel am ehesten als Maßstab aufgefaßt werden, an dem eine objektorientierte Sprache gemessen werden kann.

5.1 Datentypen, generische Klassen und Vererbung

5.1.1 Eiffel

Alle Objekte sind statisch typisiert; es wird zwischen einfachen Werten mit Wertesemantik und komplexen Objekten mit Referenzsemantik unterschieden. Im Gegensatz zu Java können die einfachen Werte genauso in generischen Klassen verwendet werden, wie Objekte, da die Typen dieser Werte in die allgemeine Typhierarchie eingebunden sind.

Eiffel bietet echte generische Klassen, die typsicher verwendet werden können.

Mehrfachvererbung ist möglich; es werden differzenzierte Mechanismen zur Konfliktbehandlung zur Verfügung gestellt.

5.1.2 C++

Alle Objekte sind statisch typisiert. Es wird aber wie in Java zwischen einfachen Werten mit Wertesemantik, die nicht in das Klassenmodell integriert sind, und komplexen Objekten mit Referenzsemantik unterschieden.

C++ bietet generische Klassen an („templates“). Dieses Konzept wurde aber erst nachträglich in die Sprache aufgenommen und ist dadurch etwas umständlich zu handhaben. Außerdem ist die Umsetzung durch die Compiler oft sehr ineffizient realisiert. Aus diesen Gründen werden „templates“ in C++-Programmen oft nicht verwendet, obwohl sie konzeptuell angebracht wären.

Mehrfachvererbung ist möglich. Dieses Konzept wurde aber genauso wie die generischen Klassen erst nachträglich in die Sprache aufgenommen.

5.1.3 Smalltalk

Objekte sind dynamisch typisiert, d.h. es findet keine Deklaration von Variablen mit einem Datentyp statt. Einfache Werte mit Wertesemantik und komplexe Objekte mit Referenzsemantik sind vollständig in die Klassenhierarchie eingebunden.

Aufgrund der dynamischen Typisierung stellt sich das Problem der generischen Klassen als Sprachkonzept gar nicht. Da jeder Variablen Objekte beliebigen Typs zugewiesen werden können, werden Gruppierungsdatentypen ähnlich wie in Java implementiert. Das bedeutet, daß die Verwendung solcher Strukturen nicht typsicher ist.

Smalltalk erlaubt nur einfache Vererbung.

5.2 Einkapselung/Sichtbarkeitsbereiche

5.2.1 Eiffel

Klassen sind in Eiffel immer global sichtbar. Wenn zwei gleich benannte Klassen in einem Programm verwendet werden sollen, dann kann eine der beiden Klassen mit einer Art Alias-Namen versehen werden.

Methoden und Attribute können global sichtbar sein, oder nur für bestimmte Klassen sichtbar gemacht werden oder für andere Klassen unsichtbar sein. Die Elemente einer Klasse sind für erbende Klassen immer sichtbar. Dadurch wird das sogen. „open-closed“-Prinzip realisiert. Nach [Meye88] sollte die Einkapselung nur zwischen Klassen angewendet werden, die in einer „Client-Supplier“-Beziehung stehen (eine Klasse ist „Client“ einer anderen, wenn sie deren Dienste benutzt). Ganz anders bei der Vererbung. Dort besteht eine sehr enge Beziehung zwischen Oberklasse und erbender Klasse, da die erbende Klasse eine Erweiterung der Oberklasse ist. Gegenüber der erbenden Klasse müssen alle Elemente offengelegt werden, da der Programmierer, der die Oberklasse implementiert, nicht voraussehen kann, welche Erweiterungen in der Zukunft durch Vererbung realisiert werden.

Auf Attribute kann von „Client“-Klassen immer nur lesend zugegriffen werden (sofern sie sichtbar sind). Diese Maßnahme ist Teil des „uniform access“-Prinzips [Meye88]. Demnach sollte es für eine „Client“-Klasse keinen Unterschied machen, ob ein Wert aus einem Attribut ausgelesen oder über eine parameterlose Funktion berechnet wird. D.h. die Implementierungsentscheidung, ob ein Wert gespeichert oder berechnet wird, darf keine Auswirkung auf die Schnittstelle einer Klasse ha-

ben, so daß diese Entscheidung jederzeit revidiert werden kann, ohne den Rest des Systems ändern zu müssen. Wenn nun eine „Client"-Klasse die Möglichkeit bekäme einem Attribut direkt einen Wert zuzuweisen, dann wäre für diese Klasse die Tatsache, daß es sich um ein Attribut handelt, offensichtlich und das Attribut könnte nicht ohne weiteres durch eine Funktion ersetzt werden (an die keine Zuweisung möglich ist).

5.2.2 C++

Methoden und Attribute können global sichtbar sein (public). Die erste Stufe der Einschränkung ist der Sichtbarkeitsbereich „protected"; so gekennzeichnete Elemente sind nur für die Erben, aber nicht für „Client"-Klassen sichtbar. Die stärkste Einschränkung bietet, wie in Java, der Bereich „private". Solche Elemente sind weder für „Client"-Klassen, noch für erbende Klassen sichtbar.

Durch den Sichtbarkeitsbereich „private" wird das oben genannte „open-closed"-Prinzip verletzt, da den erbenden Klassen nicht alle Elemente offengelegt werden. B. Meyer sieht darin eine Einschränkung der Erweiterbarkeit von objektorientierten Programmsystemen.

In C++ können Klassen zu „Freunden" anderer Klassen erklärt werden und bekommen so Zugriff auf Elemente, die nicht als „public" deklariert sind.

An öffentliche (public) Attribute können auch Werte zugewiesen werden, so daß der erwähnte „uniform access" in C++ nicht möglich ist. Abgesehen davon muß bei einer parameterlosen Funktion immer ein leeres Klammerpaar angegeben werden, wodurch sich der Aufruf einer solchen Funktion immer von einem lesenden Zugriff auf ein Attribut unterscheidet (`obj.value` bzw. `obj.value()`). Dasselbe gilt auch für Java.

5.2.3 Smalltalk

Attribute sind für „Client"-Klassen nie sichtbar, während alle Methoden immer global verfügbar sind.

5.3 Polymorphe Methoden und Overloading

Polymorphe Methoden sind Methoden in *unterschiedlichen* Klassen, die dieselbe Signatur aber verschiedene Implementierungen besitzen. Bei einem polymorphen Methodenaufruf sorgt das Laufzeitsystem automatisch dafür, daß je nach Objekttyp die richtige Implementierung ausgeführt wird. Dies ist in allen hier zum Vergleich anstehenden Sprachen möglich. In C++ müssen diese Methoden aber als „virtual" speziell gekennzeichnet werden.

Als überladene Methoden bezeichnet man Methoden mit gleichem Namen, aber unterschiedlichen Argumenttypen und Implementierungen in der *gleichen* Klasse. Wenn es sich hierbei nicht um Methoden, sondern Operatoren handelt, dann spricht man von Operator-Overloading.

Eiffel bietet für bestimmte Operatoren das Operator-Overloading an. C++ kennt beide Konzepte, während Java nur für das Überladen der Konstruktormethoden erlaubt. In Smalltalk sind diese Konzepte auf Grund der fehlenden statischen Typisierung nicht vorhanden und z.T. auch nicht notwendig.

In [Joyn96] wird das Konzept des Methoden-Overloading allerdings als fehlerträchtig kritisiert.

5.4 Speicherverwaltung

Eiffel und Smalltalk bieten einen Garbage Collector an, der nicht mehr benutzte Objekte automatisch beseitigt (wie in Java).

In C++ muß sich der Programmierer um die Speicherverwaltung kümmern. Es gibt dort zwar, analog zu den Konstruktoren, sogen. Destruktoren, die beim Löschen eines Objekts Aufräumarbeiten übernehmen und in bestimmten Situationen auch automatisch aufgerufen werden. Die Hauptarbeit bleibt aber beim Programmierer und ist eine Quelle schwer zu beseitigender Fehler. C++ ist auch die einzige, der hier aufgeführten Sprachen, die es mit Hilfe von Zeigern erlaubt, direkt Speicherinhalte zu manipulieren.

5.5 Exceptionhandling

Das Exceptionhandling von C++ entspricht im wesentlichen dem von Java. Auch Smalltalk kennt Mechanismen, die es erlauben, Fehler zur Laufzeit abzufangen.

Das umfangreichste Konzept im Bereich der Programmsicherheit bietet Eiffel an. Auf Basis des „Programming by Contract“ [Meye88] können in Eiffel zu jeder Methode Vor- und Nachbedingungen formuliert werden. Zu jeder Klasse können Klasseninvarianten angegeben werden, die Zustände und Randbedingungen beschreiben, die für jedes Objekt der Klasse während seiner Lebensdauer eingehalten werden müssen. Außerdem können Schleifenvarianten und sogen. Check-Klauseln angegeben werden, die weitere Fehlersituationen kontrollieren. Basierend auf diesen Integritätsbedingungen arbeitet in Eiffel der eigentliche Fehlerbehandlungsmechanismus.

6 Ausblick

Java ist gegenüber anderen populären objektorientierten Sprachen wie C++ oder Smalltalk sicher ein Fortschritt. Besonders im Vergleich zu C++ fehlen in Java viele Elemente, die häufig kritisiert wurden [Joyn96, Wien95] und C++ zu einer sehr komplexen Sprache gemacht haben. Allerdings haben viele Konzepte und Ideen, die in Eiffel schon seit einiger Zeit verwirklicht sind, in Java leider keinen Eingang gefunden.

Der oft angeführte Kritikpunkt, Java sei in der Ausführung zu langsam, ist sicher nur ein temporäres Problem. Die Verwendung von verbesserter Interpretertechnologie, von JIT-Compilern (JIT = Just in time) und echten Native-Code-Compilern dürfte hier Abhilfe schaffen.

Was im Moment noch große Schwierigkeiten bereitet, ist die schnelle Entwicklung der Sprache und insbesondere der zugehörigen Bibliotheken. Zu Beginn des Jahres erschien JDK 1.1 (Java Development Kit). Nach Angaben der Firma Sun wird im Abstand von jeweils 6-8 Monaten zur aktuellen Version eine neue Nachfolgeversion erscheinen. Kaum ein Softwarehersteller kann dieses Tempo mithalten. So gibt es im Moment keine kommerziell verfügbare Entwicklungsumgebung, die JDK 1.1 unterstützt und keiner der verbreiteten Web-Browser kann Applets entsprechend der Version 1.1 ausführen.

Auch den zu Java gehörenden Bibliotheken ist anzumerken, daß sie in kurzer Zeit entwickelt wurden. Positiv hervorzuheben ist hier allerdings, daß es diese Bibliotheken überhaupt gibt und daß sie einen Standard bilden. Objektorientierte

Sprachen sind meist sehr schlank und haben einen recht kompakten Sprachumfang, so daß die Ausdrucksmächtigkeit erst durch die umfangreichen Klassenbibliotheken entsteht. Dies hat zur Folge, daß Compiler oder Entwicklungsumgebungen verschiedener Hersteller nur dann zueinander kompatibel sind, wenn sie auch dieselben Bibliotheken anbieten. Ein Umstand, dem bisher kaum Rechnung getragen wurde und der z.B. Smalltalk-Entwicklern den Umstieg auf eine andere Entwicklungsumgebung fast unmöglich machte.

Sobald Java einen gewissen Grad der Reife und Stabilität erreicht hat, wird sich die Sprache sicher auch für umfangreichere Projekte einsetzen lassen. Die Verfügbarkeit einer großen Zahl von Entwicklungswerkzeugen, Klassenbibliotheken und Schnittstellen ist jetzt schon abzusehen.

7 Literatur

[Flan96] David Flanagan: *Java in a nutshell*, O'Reilly, Bonn, Cambridge/Mass., 1996

[GJS96] James Gosling, Bill Joy, Guy Steele: *The Java language specification*, Addison-Wesley, Reading, Mass, 1996 oder:
http://java.sun.com:80/docs/books/jls/html/index.html

[Java97] http://java.sun.com

[Joyn96] Ian Joyner: A Critique of C++ and Programming and Language Trends of the 1990s, 3rd edition, c/- Unisys - ACUS, 115 Wicks Rd, North Ryde, Australia 2113, ian@syacus.acus.oz.au,
http://www.progsoc.uts.edu.au/~geldridg/cpp/cppcv3.html

[Meye88] Betrand Meyer: *Object oriented software construction*, Prentice-Hall, New York, 1988

[Meye92] Betrand Meyer: *Eiffel: The Language, Prentice-Hall*, New York, 1992

[Meye97] Bertrand Meyer: *Object oriented software construction*, 2nd edition, Prentice-Hall, New York, 1997

[Stri97] java.lang.String – API-Dokumentation zur Java-String-Klasse
http://java.sun.com:80/products/jdk/1.1/docs/api/java.lang.String.html

[Sure97] York Sure: *Programmieren mit Threads*, Vortrag im Rahmen des Seminars: Objektorientierte Programmierung, Java und das Internet, Institut AIFB, 1997,
http://www.aifb.uni-karlsruhe.de/Lehrangebot/Sommer1997/java-seminar/

[Wien95] Richard Wiener: *Software development using Eiffel – There can be life other than C++*, Prentice-Hall, Englewood Cliffs, NJ, 1995