Überblick über die Programmiersprache Java

Geschrieben von Oliver Stecklina

Letzte Änderung 10.Juni 1998

Inhaltsverzeichnis

1 Einleitung

Entstanden ist diese Arbeit im Rahmen eines Seminars an der BTU Cottbus im Sommersemester 98. Diese Arbeit soll einen Einblick in die objektorientierte Programmierung anhand der Sprache Java geben und baut auf die Ausarbeitung von Alek Opitz zum Thema UML auf. Da die Erläuterung einer Programmiersprache ein sehr umfangreiches Thema ist, beschränkt sich diese Arbeit auf nur einige ausgewählte Gebiete von Java. Als Grundlage für diesen Bericht diente das Buch [Lem97]. Allerdings beschäftigt sich, die von mir verwendet Auflage noch mit Java 1.0. Da ich aber in meinem Bericht nur die Grundlagen von Java behandle, spielt dies hier keine Rolle.

Anschließend wird in diesem Kapitel ein Beispielprogramm in Form eines UML Klassendiagramms vorgestellt und kurz erläutert. Aus diesem Beispiel sind bis auf wenige Ausnahmen fast alle Listings dieses Berichtes entnommen.

Im zweiten Kapitel folgt eine kurze Einleitung in die Programmiersprache Java. Themen sind die Geschichte und die Plattformunabhängigkeit von Java.

Das dritte Kapitel gibt eine Einführung in die Grundlagen der Programmiersprache. Da Java sehr an die Sprache C angelehnt ist, wird größtenteils auf eine umfangreiche Beschreibung verzichtet.

Im darauffolgenden vierten Kapitel, welches den überwiegenden Teil dieser Arbeit ausmacht, wird die Umsetzung einiger objektorientierter Konzepte in Java behandelt.

1.1 Das Programmbeispiel

Um die Verständlichkeit der einzelnen Konstrukte zu vereinfachen, werden alle an einem gemeinsamem Beispiel erläutert. Der gesamte Sourcecode des Beispiels ist im Anhang zu finden.

Abbildung 1.1 zeigt das Klassendiagramm des Beispielprogrammes. Das Diagramm beinhaltet sämtliche Klassen sowie die Schnittstelle `SortInterface`. Alle Klassen, deren Eigenschaften und Attribute werden in den folgenden Abschnitten noch ausführlich erläutert.

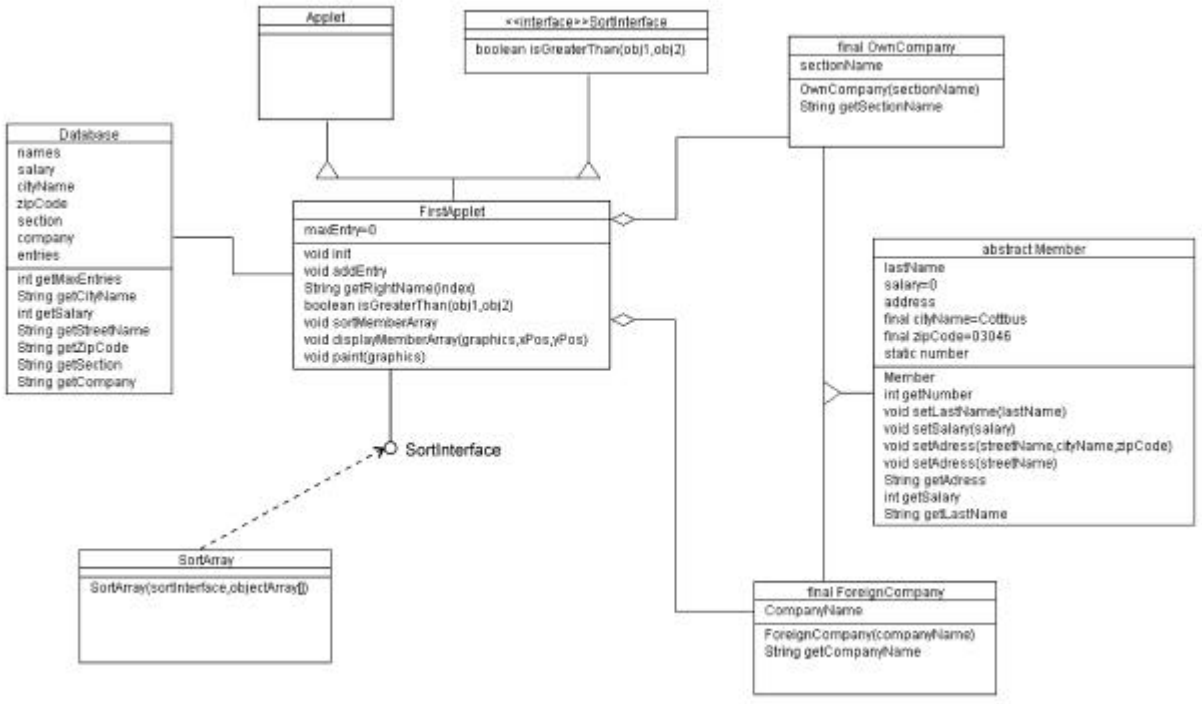

Abb. 1.1 UML Klassendiagramm

Als Beispiel wird ein kleines Applet implementiert. Applets sind dynamische, interaktive Programme, die in einer Internet – Seite ausgeführt werden können. Als Grundlage für das Applet dient eine Datenbank mit dem Mitarbeiterstamm einer kleinen fiktiven Firma. Wir gehen davon aus, daß die Firma eigene Angestellte und Mitarbeiter fremder Firmen beschäftigt. Um einen kleinen Überblick über alle Mitarbeiter zu bekommen, sollen die Mitarbeiter mit den jeweils dazugehörigen Daten dargestellt werden. Zu jedem Mitarbeiter ist der Name, die Adresse, das monatliche Gehalt und der Beschäftigungsort in einer kleinen Datenbank gespeichert. Um Schreibaufwand zu vermeiden, wurden nur bei den Angestellten der Ort und die Postleitzahl zusätzlich gespeichert, wo sich diese Werte vom Firmenstandort unterscheiden. In dem Applet werden alle Mitarbeiter nach ihrem Einkommen sortiert, mit den Namen, der kompletten Adresse, dem Beschäftigungsort und ihrem Gehalt angezeigt.

In diesem Bericht werden nur einzelne Stücke des Programmes dargestellt. So wird zum Beispiel der verwendet Sortieralgorithmus nur im Anhang komplett vorgestellt. Es wird nur soweit auf die Implementierung eingegangen, daß die Grundlagen und die objektorientierten Konzepte deutlich werden. Obwohl der Hauptverwendungszweck des Applets die Präsentation von Daten ist, ist das Layout auf das Nötigste beschränkt. Auf die Layoutgestaltung wird in diesem Bericht nicht eingegangen, da dies ein eigenes Thema in Java ist.

2 Was ist Java ?

2.1 Die Geschichte

Die Idee einer solchen Programmiersprache bestand schon seit längerer Zeit, die eigentliche Arbeit begann ungefähr 1990 bei SUN. Die Grundidee war, eine Programmiersprache für kleine spezielle Anwendungen zu entwickeln. Dabei sollte sich Java an C++ anlehnen, jedoch bedeutend einfacher und schneller zu erlernen sein.

Eine Gruppe von Programmierern um James Gosling und Bill Joy entwickelten aus diesen Grundüberlegungen Java. Das Projekt beschäftigte sich zunächst wenig erfolgreich mit Consumer Electronics, Interactive TV und Settop Boxes, bevor es zu seinem heutigen Einsatzgebiet im World Wide Web kam.

Ab 1995 wurde Java in diese Richtung weiterentwickelt. Die erste größere Anwendung von Java für das World Wide Web war der Hot Java Browser. Dieser Browser hatte die Fähigkeit, zur Laufzeit neue Protokolle und Dateiformate einzubinden und in Java geschriebene, über das Netz geladene Programme auszuführen.

Durch den Erfolg dieses Browsers wurde das Interesse an Java auch bei anderen Anbieter von WWW-Browser geweckt. Sie statteten ihre Browser ebenfalls mit einem Java-Byte-Code-Interpreter aus, so konnte wenig später der Netscape-Navigator oder der von Microsoft entwickelte Internet Explorer ebenfalls Java-Applets abspielen. IBM, Borland, Microsoft und andere Firmen haben Java lizenziert, um eigene Anwendungen zu entwickeln.

2.2 Java ist plattformunabhängig

Die Plattformunabhängigkeit ist eine der wichtigsten Vorteile von Java. Java ist sowohl auf der Quell- als auch auf der Binärebene plattformunabhängig.

Auf allen Entwicklungsplattformen hat auf der Quellebene jeder primitive Datentyp die gleiche Größe. Da Java in den meisten Fällen auf die Nutzung umfangreicher Klassenbibliothek zurückgreift, wird das Schreiben einer Anwendung, die von einer Plattform auf die andere verlagert werden soll, vereinfacht. Der Code muß an keine plattformspezifischen Eigenschaften angepaßt werden.

Beim Kompilieren eines C – oder PASCAL – Programms wird mittels eines für jede Plattform entwickelten Compilers ein spezifischer Maschinencode erzeugt. Soll das Programm auf einem anderen System eingesetzt werden, muß der Sourcecode an die plattformspezifischen Eigenschaften angepaßt und neu kompiliert werden. In einzelnen Fällen kann es durchaus auch nötig sein, daß der Quellcode ziemlich umfangreich überarbeitet werden muß. (Siehe Abb. 2.1).

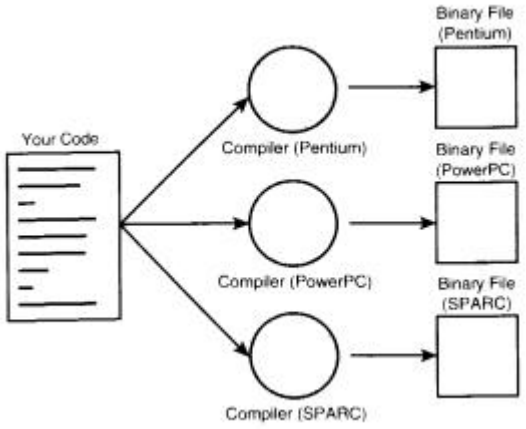

Abb. 2.1 Herkömmlich kompilierte Programme (entnommen [Lem97])

Bei Java wird die Plattformunabhängigkeit auf der Binärebene durch die Interpretation des sogenannten Bytecodes erreicht. In diesem Sinn ist Java eine Interpretersprache. Wohingegen bei herkömmlichen Interpretersprachen z.B. Basic, Tcl oder Perl der Sourcecode interpretiert wird, wird bei Java ein vorkompilierter Byte-Code interpretiert.

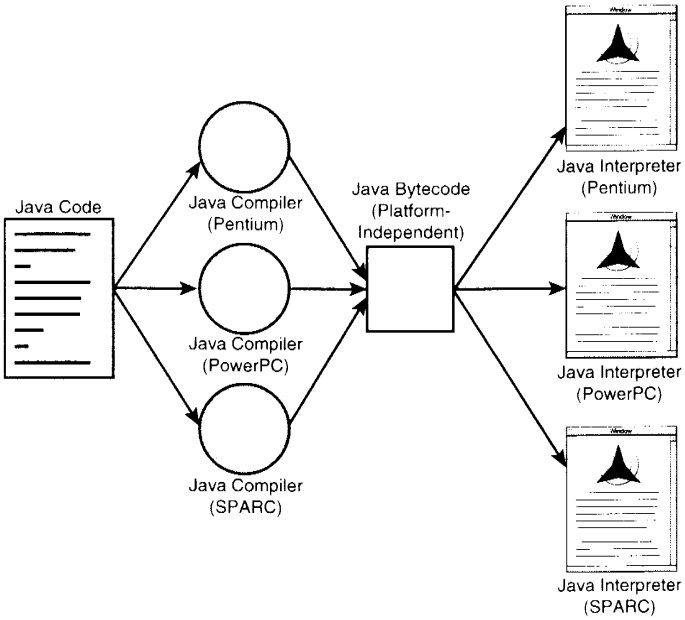

Abb. 2.2 Java- Programme entnommen [Lem97]

Der Bytecode ist vollkommen plattformunabhängig. Erst der Interpreter, oft in einem WWW-Browser integriert, ist für die jeweilige Plattform kompiliert (siehe Abb. 2.2).

Der Nachteil eines Interpreters zeigt sich in der Geschwindigkeit, mit welcher die Programme ausgeführt werden. Der Bytecode wird erst zur Laufzeit in Maschinenbefehle übersetzt und dann

ausgeführt. Aus diesem Grund wird bei Java auch der Bytecode und nicht der reine Quellcode interpretiert. Dieser ist schon in eine maschinennahe Form gebracht und kann bedeutend schneller als der Sourcecode interpretiert werden.

Bei vielen Java – Programmen ist die Geschwindigkeit jedoch nicht der entscheidende Faktor. Soll ein Programm geschrieben werden, bei dem die Geschwindigkeit eine wichtige Rolle spielt, kann der Byte - Code durch eine Java - Compiler in eine Maschinencode konvertiert werden. Die Programme, die in Maschinencode umgewandelt wurden, haben dann keine großen Geschwindigkeitsnachteile zu anderen Programmiersprachen. Bei solchen Lösung sollte man aber immer beachten, daß man damit die Portabilität des Java-Programmes einbüßt.

3 Einführung in die Java – Programmierung

3.1 Java Grundlagen

Dieser Abschnitt behandelt die Grundlagen für die Erstellung eines Programmes in Java. Besprochen werden Variablen, die Datentypen, Zeichenketten (Strings), Ausdrücke sowie die verschiedenen Operatoren.

3.1.1 Variablen

Java kennt drei Arten von Variablen: *Instanzvariablen*, *Klassenvariablen* und *lokale Variablen.*

Instanzvariablen werden zum Definieren von Attributen oder des Zustandes eines bestimmten Objektes benutzt.

Klassenvariablen sind mit Instanzvariablen vergleichbar. Der Unterschied besteht darin, daß sie nicht auf eine Instanz beschränkt sind.

Auf die Benutzung und die Unterschiede zwischen Klassen- und Instanzvariablen wird im Abschnitt 4.1.2 noch tiefgründiger eingegangen.

Lokale Variablen werden innerhalb von Blöcken deklariert und benutzt. Blöcke sind z.B. Prozedurkörper oder Körper von Schleifen. Eine lokale Variable gilt nur in dem Block, in dem sie angelegt wurde. Außerhalb dieses Blockes hört sie auf zu existieren. Lokale Variablen werden zum Speichern von Informationen, die nur in den jeweiligen Block von Bedeutung sind (z.B. Indexzähler in Schleifen), genutzt.

Im Gegensatz zu anderen Sprachen gibt es in Java keine globalen Variablen. Um globale Informationen auszutauschen, nutzt man in Java Instanz- und Klassenvariablen. Da Java eine objektorientierte Sprache ist, denkt man an Objekte sowie deren Interaktionen und weniger an Programme im herkömmlichen Sinne.

Deklaration von Variablen

```
1:     int currentAge;
2:     String lastName;
3:     boolean isActiv;
4:
5:     int x,y,z;
6:
7:     String cityName = "Cottbus"
8:     int hisAge, hisSize = 180;
```

Listing 3.1 Deklarieren von Variablen

Die Deklaration einer Variable erfolgt wie in C++. Zuerst steht der Datentyp gefolgt von dem Variablennamen (siehe Listing 3.1). Es ist ebenfalls möglich mehrere Variablen, durch ein Komma getrennt, zu verketten. Alle Variablen werden dann vom selben Typ deklariert.

Außerdem kann jeder Variable während der Deklaration ein Anfangswert zugewiesen werden. Wird nur eine Variable initialisiert, gilt der Wert auch nur für diese Variable auch

dann, wenn mehrere Variablen gleichzeitig deklariert werden. Sollen mehrere Variablen, die verkettet deklariert wurden, eine Initialwert bekommen, muß die Initialisierung für jede Variable explizit angegeben werden.

Bei der Wahl der Variablennamen ist zu beachten, daß als erstes Zeichen keine Zahlen und kein Sonderzeichen wie %, *, @ usw. verwendet werden. Erlaubt ist die Nutzung von „_“ sowie „$“. Außerdem unterscheidet Java zwischen der Groß- und Kleinschreibung, wodurch die Variable `name` verschieden von der Variablen `Name` ist.

3.1.2 Datentypen

Wie schon oben erwähnt, steht bei der Deklaration einer Variablen zuerst der Datentyp. In Java ist ein Datentyp:

- einer der acht primitiven Datentypen,
- der Name einer Klasse oder
- ein Array.

Arrays und Klassen werden zu einem späterem Zeitpunkt behandelt. An dieser Stelle folgt eine Erläuterung der acht primitiven Datentypen.

`boolean`: Im Gegensatz zu C ist in Java ein Datentyp für Wahrheitswerte vorgesehen. Er kann die Werte `true` und `false` annehmen.

`char`: Der Zeichentyp umfaßt die 16-Bit-Unicode-Zeichen.

Ganzzahltypen: In Java gibt es vier Ganzzahltypen (siehe Tabelle 3.1) mit je einem anderen Wertebereich. Wird ein Wert zu groß für den Variablentyp, wird er abgeschnitten.

Typ	Größe	Bereich
`byte`	1 Byte	-128 bis 127
`short`	2 Byte	-32.768 bis 32.767
`int`	4 Byte	-2.147.483.648 bis 2.147.483.647
`long`	8 Byte	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807

Tabelle 3.1 Ganzzahltypen

Gleitkommazahlen: Es gibt zwei verschiedene Gleitkommazahlentypen: `double` und `float`.

`float`: Bei `float` handelt es sich um eine 32-Bit- IEEE754-Gleitkommazahl (eine internationale Norm zur Definition von Gleitpunktzahlen und Arithmetik).

`double`: Hierbei handelt es sich um eine 64-Bit- IEEE754-Gleitkommazahl. .

3.1.3 Zeichenketten

Eine Folge von Zeichen des Typs `char` wird als Zeichenkette bzw. `String` bezeichnet. In Java ist eine Zeichenkette eine Instanz des Klasse `String`. Wie schon oben erwähnt, sind Klassen ebenfalls Datentypen. Mittels des Klassennamens und eines Namens für die Variable wird eine Variable vom Typ dieser Klasse deklariert (siehe Listing 3.1). Die Bezeichnung Variable ist an dieser Stelle eigentlich nicht richtig, man spricht von einer Instanz der Klasse `String`. In Java sind Zeichenkette nicht Arrays von Zeichen, wie in der Programmiersprache C, obwohl es viele Ähnlichkeiten gibt.

Die Klasse `String` beinhaltet verschiedene Methoden, um `String`s zu bearbeiten. Wird in einem Programm eine Zeichenkette verwendet, erstellt Java automatisch eine Instanz der Klasse `String`. Bei allen anderen Klassen ist es normale rweise nötig, explizit eine Instanz der Klasse anzulegen. Bei allen anderen primitiven Datentypen handelt es sich um keine Objekte.

Zeichenketten können Konstanten wie neue – Zeile -, Tab - und Unicode - Zeichen enthalten. Für `String`s existieren außerdem besondere Operationen, welche in dem Punkt „Zeichenkettenarithmetik“ genauer beschrieben werden.

3.1.4 Ausdrücke und Operationen

Auch an dieser Stelle wird die Nähe von Java zu C deutlich. Der Umgang mit Ausdrücken und Operationen erfolgt genauso, wie man es aus diesen Sprache kennt.

Arithmetik

In Java sind die in Tabelle 3.2 dargestellten arithmetischen Operationen bekannt.

Operator	Bedeutung	Beispiel
+	Addition	`3 + 4`
-	Subtraktion	`5 – 7`
*	Multiplikation	`5 * 5`
/	Division	`14 / 7`
%	Modulus	`20 % 7`

Tabelle 3.2 Operationen in Java

Die Division einer Ganzzahl ergibt wieder eine Ganzzahl. Ist einer der Operanden jedoch eine Gleitkommazahl, so ist das Ergebnis der Operation ebenfalls eine Gleitkommazahl.

Zuweisungen

Eine Zuweisung erfolgt immer über das Gleichheitszeichen, dabei steht auf der linken Seite immer eine Variable oder ein Arrayelement, wohingegen auf der rechten Seite auch ein arithmetischer oder logischer Ausdruck stehen kann. Zuweisungen können beliebig verkettet werden. Bei einem Zuweisungsausdruck wird immer erst die rechte Seite ausgewertet und dann das Ergebnis der linken Seite zugewiesen. Damit wir bei einer Zuweisung, wie in Tabelle 3.3 in der 2.Spalte während der Auswertung des Ausdrucks der ursprüngliche x-Wert verwendet.

Ausdruck	Bedeutung
`x += y`	`x = x + y`
`x -= y`	`x = x – y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`

Tabelle 3.3 Abkürzungen

In Java können die gleichen Abkürzungen wie in C verwendet werden. Die Kurzformen sind in der Tabelle 3.3 beschrieben.

Inkrement- und Dekrementausdrücke

Wie in C und C++ sind auch in Java die Operatoren “++“ und “--“ zum Inkrementieren bzw. Dekrementieren von Variablen vorhanden. Die Inkrement- und Dekrementoperatoren können vorangestellt oder hinten angefügt werden.

```
1:     int y1, y2, x = 0;
2:     y1 = x++;
3:     x =0;
4:     y2 = ++x;
```

Listing 3.2 Inkrement- und Dekrementausdrücke

Grundsätzlich wird sowohl bei dem Ausdruck `x++` als auch bei `++x` der Wert von `x` um `1` erhöht. Der Unterschied besteht nur darin, wann der Wert erhöht wird. Man betrachte das Listing 3.2. Am Ende der Ausführung dieses Codestückes steht in `y1` eine `0` und in `y2` eine `1`. Bei `y1` wurde erst der Wert von `x` der Variable `y1` zugewiesen und dann `x` inkrementiert. Bei `y2` erfolgt dies genau umgekehrt, zuerst wird `x` inkrementiert und dann wird der neue Wert `y2` zugewiesen.

Vergleiche

Bei den Vergleichsoperationen gibt es keine Besonderheiten. Tabelle 3.4. enthält die Vergleichsoperationen von Java. Das Ergebnis ist immer vom Typ `boolean` .

Operator	Bedeutung	Beispiel
==	Gleich	X == 3
!=	Ungleich	X != 3
<	Kleiner als	X < 3
>	Größer als	X > 3
<=	Kleiner als oder gleich	X <= 3
>=	Größer als oder gleich	X >= 3

Tabelle 3.4 Vergleiche

Bitweise Operatoren

Die bitweisen Operatoren stammen alle von C und C++. Da sie in keinem weiterem Abschnitt verwendet werden, wird hier auch nicht genauer auf sie eingegangen. Ein kurzer allerdings nicht vollständiger Überblick wird in der Tabelle 3.5 gegeben. Eine genauere Beschreibung findet man in jedem C und C++ Handbuch.

Operatorpräzedenzen

Durch die Operatorpräzedenzen wird die Reihenfolge vorgegeben, in der Ausdrücke ausgewertet werden. Die Tabelle 3.5 zeigt die Präzedenzen, wie sie in Java verwendet werden. Die Operatoren

Operator	Anmerkung
`. [] ()`	Mit Klammern werden Ausdrücke gruppiert. Der Punkt dient zum Zugriff auf Methoden und Variablen in Objekten und Klassen. [ ] wird für Arrays verwendet.
`++ - ! ~ instanceof`	Die ersten 4 Operation sollten bekannt sein. `Instanzof` liefert den Klassenamen einer Instanz .
`new (Typ) Ausdruck`	Der `new` – Operator wird zum Erstellen neuer Instanzen von Klassen benutzt. `()` dient in diesem Fall zum Abbilden eines Wertes auf einen anderen Typ.
`* / %`	Multiplikation, Division und Modulus
`+ -`	Addition und Subtraktion
`<< >>`	Bitweises Verschieben nach links oder nach rechts
`< >`	Relationale Vergleichstests
`== !=`	Gleichheit und Ungleichheit
`& ^ |`	AND XOR und OR
`&& ||`	Logisches AND und logisches OR
`? :`	Kurzform für `if ... then ... else`
`+= -= *= /= &= ^= |= <<= >>= >>>=`	Verschiedene Zuweisungen

Tabelle 3.5 Operatorpräzendenzen

sind von oben nach unten einzuordnen, d.h. je weiter unten ein Operator in der Tabelle steht, um so später wird er ausgewertet.

Zeichenkettenarithmetik

An dieser Stelle spielt der Additionsoperator „+“ eine wichtige Rolle. Mittels dieses Operators ist es möglich zwei Strings zu einem String zu verbinden.

```
1:     String outStr, myName = " Tody ";
2:     int myAge = 21;
3:
4:     outStr = myName + " is " + myAge;
5:     outStr += " old ";
```

Listing 3.3 Zeichenketenarithmetik

Listing 3.3 zeigt ein Beispiel für die Zeichenkettenarithmetik in Java. In Zeile 4 wird mittels des Additionsoperators ein neuer String zusammengebaut. Die Variable `myAge` von Typ `int` wird hierbei automatisch zu einem `String` konvertiert. Der in Zeile 5 verwendete Operator „+=“, funktioniert ebenfalls bei Zeichenketten.

3.2 Arrays, Bedingungen und Schleifen

3.2.1 Arrays

Im Gegensatz zu anderen Sprachen sind Arrays in Java Objekte. Dadurch ist der Umgang mit Arrays etwas anders als in C oder C++. Arrays können jeden Datentyp enthalten, es ist jedoch nicht möglich, daß ein Array Elemente verschiedene Datentypen enthält.

Das Erstellen eines Arrays erfolgt in zwei Schritten:

?? Zunächst wird ein Variable zur Aufnahme eines Arrays deklariert,

?? anschließend wird ein neues Array – Objekt erstellt.

```
1:     String names[];
2:     names = new String[10];
3:
4:     int[] age = new int[10];
5:
6:     int[] temp = {10, 11, 12}
7:
8:     age[0] = 21;
9:     int result = age[1];
```

Listing 3.4 Arbeiten mit Arrays

Listing 3.4 zeigt, wie Array – Variablen deklariert und ein Array – Objekt erstellt wird. In Zeile 1 und 2 wird ein Arrays von Typ `String` mit maximal 10 Elementen erstellt. Wird ein Array – Objekt mit `new` angelegt, muß in den eckigen Klammern ([]) die Größe des Arrays angeben werden. Anstelle mit `new` zuarbeiten, kann wie in Zeile 6 ein Array auch manuell erstellen und initialisieren.

Der Zugriff auf ein Element eines Arrays erfolgt wie in Zeile 8 und 9 von Listing 3.4. Mittels des Arraynamens und des Elementindex, welches in den eckigen Klammer angegeben wird, kann der Wert des Elementes ausgelesen werden. Beim Erstellen eines Objektes mit `new` werden alle Elemente mit `0` bei Zahlen, `false` bei Wahrheitswerten und `'\0'` bei Strings initialisiert. Wie in den meisten anderen Programmiersprachen hat das erste Element eines Arrays den Index 0 und das n-te Element den Index n-1.

In Java ist es nicht möglich, ein mehrdimensionales Array anzulegen. Dies ist auch nicht nötig, da Arrays aus allen Datentypen aufgebaut werden können. Es ist leicht möglich, eine Array von Arrays anzulegen. Damit kann ein Array mit beliebig vielen Dimensionen angelegt werden. Listing 3.5 zeigt die praktische Umsetzung.

```
1:     int coords[][] = new int [10][10];
2:     coords[0] [0] = 1;
3:     coords[0] [1] = 2;
```

Listing 3.5 Mehrdimensionale Arrays

3.2.2 Blockanweisungen

Blockanweisungen fassen mehrere Anweisungen zu einer Anweisung zusammen. Man schreibt die Anweisungen in geschweifte Klammern ({}). Blockanweisungen sollten aus anderen Programmiersprachen bekannt sein. Wichtig bei der Verwendung einer Blockanweisung in Java ist, daß eine Variable, die in einem Block deklariert wird, außerhalb nicht existiert.

In den folgenden Abschnitten wird die Verwendung von Blöcken an einigen Beispielen verdeutlicht.

3.2.3 If- Bedingung

Eine `if` – Bedingung veranlaßt die Ausführung eines Programmteils anhand eines Testes. Eine `if` – Bedingung beginnt immer mit dem Schlüsselwort `if,` gefolgt von einem boolschen Test. Hier liegt auch der einzige Unterschied zu C oder C++. Während bei C oder C++ der Test eine Ganzzahl als Ergebnis haben kann, muß bei Java das Ergebnis immer ein boolscher Wert sein.

```
1:     if (number == 1) {
2:           return Database.zipCode[0];
3:     }
4:     else if (number == 5) {
5:           return Database.zipCode[1];
6:     }
7:     return "home";
```

Listing 3.6 Verwendung einer if - Anweisung

Wird der Test mit wahr bewertet, wird die Anweisung nach der Bedingung ausgeführt. An dieser Stelle setzt man Blöcke ein, da meist mehrere Anweisungen ausgeführt werden sollen. Mittels des Schlüsselwortes `else` kann die Ausführung einer Anweisung bzw. eines Blockes veranlaßt werden, wenn das Ergebnis des Testes falsch war.

Eine Alternative zur Verwendung der Schlüsselwörter `if` und `else` ist der sogenannte Bedingungsoperator. Der Bedingungsoperator wurde ebenfalls aus der Programmiersprache C übernommen. Der Bedingungsoperator hat eine sehr niedrige Präzedenz, weshalb er meist erst ganz zum Schluß ausgewertet wird.

3.2.4 Switch- Anweisung

Es kommt oft vor, daß man den Inhalt einer Variablen mit verschiedenen möglichen Werten vergleichen will. Eine Möglichkeit wäre, ein Reihe von `if` – Anweisungen zu schreiben. Um sich unnötige Schreibarbeit zu ersparen, hat man in Java die `switch` - Anweisung übernommen.

```
1:     string result;
2:     switch (anyValue) {
3:       case 2:
4:       case 3: { result = „<= 3“;
5:                 break;}
6:       case 4: { result = „== 4“;
7:                 break;}
8:       default: {
9:                 result = „>= 5“;}
10:     }
```

Listing 3.7 Verwendung einer switch - Anweisung

Bei einer `switch` – Anweisung muß die Bedingung (siehe in Listing 3.7 `anyValue`) vom Datentyp `char`, `byte`, `short` oder `int` sein. Es dürfen keine Objekte, wozu ja auch Strings und Arrays zählen und keine größeren primitive Datentypen (`long`, `float`) verwendet werden.

Mittels des Schlüsselwortes `default` kann eine Anweisung ausgeführt werden, falls das Ergebnis mit keinem der angegeben Werte übereinstimmt.

Soll eine Anweisung für ein Reihe von möglichen Ergebnissen ausgeführt werden, wird wie in den Zeilen 3 und 4 von Listing 3.7 das Ergebnis ausgelassen. Es wird dann, die nächst mögliche Anweisung ausgeführt. Mittels des Schlüsselwortes `break` wird die `switch` – Anweisung beendet bzw. abgebrochen. Wird `break` nicht explizit angegeben, so werden alle Möglichkeiten ausgewertet bis ein `break` oder das Ende der `switch` – Anweisung erreicht wird.

3.2.5 Schleifen

In Java gibt es drei Typen von Schleifen, das ist die `for` – Schleife, die `while` – Schleife und die `do` – Schleife.

Die **for – Schleife** ist die wohl am häufigsten genutzte Schleife. Der Schleifenkopf besteht aus drei Teilen. Der erste Teil beinhaltet die Initialisierung der Laufvariable und der zweite Teil die Abbruchbedingung. Im dritte Teil wird die Schrittweite der Laufvariable angegeben. Alle drei Teile stehen in einer Klammer () und werden mittels eines Semikolons von einander getrennt. Es ist durchaus möglich, einen der drei Teile oder gar alle Teile leer zulassen. Für die Ausführung der Operation, die sonst an diesen Stellen ausgeführt wird, muß dann an anderen Stellen im Schleifenkörper gesorgt werden. Nach dem Schleifenkopf folgt die auszuführende Anweisung. Üblicherweise wird an dieser Stelle meist ein Block mit Anweisungen stehen. Die Anweisung wird dann so oft ausgeführt, bis die Abbruchbedingung das Ergebnis `false` liefert oder im Anweisungsblock ein Schleifenabbruch erfolgt.

```
for (Initialisierung; Bedingung; Schrittweite) {
   Schleifenkörper;
}
```

Bei der **while – Schleife** handelt es sich um den allgemeineren Fall einer `for` - Schleife. Während bei der `for` - Schleife der Kopf aus drei Teilen besteht, wird bei der `while` – Schleife nur die Abbruchbedingung angegeben. Eine `while` – Schleife entspricht exakt einer `for` – Schleife, wenn man bei einer `for` – Schleife den Teil für die Initialisierung der Laufvariable und den Teil mit der Schrittweite leer läßt. Genauso wie bei einer `for` - Schleife wird der Anweisungsblock solange ausgeführt, bis die Auswertung der Bedingung das Ergebnis `false` liefert.

```
while (Bedingung) {
   Schleifenkörper;
}
```

Die **do – Schleife** ist eigentlich eine umgedrehte `while` – Schleife. Bei einer `while` – Schleife wird der Anweisungsblock nur ausgeführt, wenn die Bedingung einmal `true` ergibt. Bei der `do` – Schleife wird die Anweisung mindestens einmal ausgeführt und anschließend die Bedingung geprüft.

```
do {
   Schleifenkörper;
} while (Bedingung);
```

Alle drei Schleifen enden, wenn die Bedingung falsch ist. Um einen Anweisungsblock vorzeitig zu verlassen, gibt es in Java die Schlüsselwörter `break` und `continue`. Beide sorgen dafür, daß der Anweisungsblock nicht weiter ausgeführt wird. Der Unterschied ist, daß bei `break` die gesamte Schleife abgebrochen wird und bei `continue` nur der aktuelle Schleifendurchlauf. Wird eine `while` – oder eine `do` – Schleife mit `continue` abgebrochen, startet die Ausführung des Blocks wieder von vor, so die Bedingung noch wahr ist (`while` – Schleife). Bei einer `for` – Schleife wird die Laufvariable um die Schrittweite verändert und dann die Bedingung geprüft, bevor der Anweisungsteil neu ausgeführt wird.

```
1:     foo:
2:      for (int i = 1; i <= 5; i++)
3:       for (int j = 1; j <= 3; j++) {
4:            if ((i + j) > 4)
5:                 break foo;
6:            result = i + j;
7:           }
```

Listing 3.8 for – Schleife mit benannter break - Anweisung

Sowohl die `break` – als auch die `continue` – Anweisung kann benannt werden. Damit ist es möglich, bei verschachtelten Schleifen aus einer inneren Schleife mit nur einer Anweisung, alle Schleifen abzubrechen. Listing 3.8 zeigt wie Schleifen benannt werden. Beide Schleifen werden abgebrochen, wenn die Summe von `i` und `j` größer 4 ist.

4 Objektorientiert Programmierung in Java

Dieser Abschnitt behandelt das eigentliche Thema des Berichtes die objektorientierte Programmierung in Java. Diese Art der Programmierung eignet sich besonders gut, um größere Konzepte praxisnah in ein Programm umzusetzen. Eine bedeutende Eigenschaft von Java ist die Möglichkeit gute Internetapplikationen leicht und schnell zu erstellen.

4.1 Objekte und Klassen

4.1.1 Erstellen und Instanziieren von Klassen

Um eine Klasse instanziieren zu können, muß sie zuerst erstellt bzw. definiert werden. Das Definieren einer Klasse erfolgt in Java durch das Schlüsselwort `class`. In Listing 4.1 werden die Klassen `FirstApplet`, `Member`, `OwnCompany` und `ForeignCompany` aus Abbildung 1.1 definiert.

```
1:  class FirstApplet {
2:
3:     void addEntry() {
4:        Member newEntry;
5:        if (/* Es soll eine Instanz von OwnCompany
6:                  erstellt werden */) {
7:           newEntry = new OwnCompany(/*arg1, arg2, ... */);
8:        }
9:        else {
10:          newEntry = new ForeignCompany(/*arg1, arg2, ... */);
11:       }
12:    }
13:    ...
14: }
15:
16: class Member {
17:    ...
18: }
19:
20: class OwnCompany {
21:    ...
22: }
23:
24: class ForeignCompany {
25:    ...
26: }
```

Listing 4.1 Definition und Instanziierung von Klassen

Im Prinzip ist Zeile 16 und 18 eine vollständige Klassendefinition. Sie reicht aus, um im folgenden mit der Klasse `Member` arbeiten zu können. Ist eine Klasse erst einmal definiert, so kann sie wie ein Datentyp verwendet.

Um eine Instanz einer Klasse zu erstellen, verwendet man das Schlüsselwort `new` gefolgt von dem Klassennamen. In Zeile 7 wird eine Instanz der Klasse `OwnCompany` erstellt. In der Instanz `OwnCompany` werden später jeweils die Informationen eines Mitarbeiters gespeichert. Das Instanziieren muß immer in einer Methoden erfolgen. Im Listing 4.1 ist dies die Methode `addEntry`. Beim Instanziieren können wie in Zeile 7 Argumente an das Objekt übergeben werden, diese werden dann von einer sogenannten Konstruktor – Methode verarbeitet. Auf das Thema Methoden gehe ich zu einem späteren Zeitpunkt noch genauer ein.

4.1.2 Instanz - und Klassenvariablen

Die Klasse `Member` aus Listing 4.1 ist zwar bereits eine Klasse, jedoch hat sie keinerlei Attribute oder Eigenschaften. Der Sinn einer Klasse besteht aber darin, Eigenschaften und Attribute eines bestimmten Objektes zu sammeln. In diesem Abschnitt wenden wir uns den Attributen zu. Attribute werden mit Hilfe von Variablen in einer Klasse definiert. Man spricht dann von Instanz- bzw. Klassenvariablen. Die Deklaration einer solchen Variable erfolgt wie in Abschnitt 3.1.1 beschrieben. Um dies zu verdeutlichen, erweitern wir die Klasse `Member` um einige Attribute.

```
1:     class Member {
2:           String lastName;
3:           int salary = 0;
4:           String adress;
5:
6:           static int number;
7:
8:           final String cityName = "Cottbus";
9:
10:          final String zipCode = "03046";
11:
12:          // Klasse ist noch unvollständig
13:    }
```

Listing 4.2 : Klasse mit Attributen

Wird jetzt eine Instanz der Klasse `Member` erstellt, stehen jedem Objekt die Attribute `lastName`, `salary`, `adress`, `number`, `cityName` und `zipCode` zur Verfügung. Es ist durchaus möglich, die Variablen mit einem Grundwert zu initialisieren. In jeder Instanz, die von der Klasse `Member` erstellt wird, ist der Wert der Variablen `salary` mit 0 initialisiert.

Man betrachte die Zeile 6 aus Listing 4.2. An dieser Stelle wurde die Variable `number` initialisiert. Sie dient als Zähler für die Anzahl der Arbeiter. Da wir für jeden Arbeiter eine Instanz der Klasse `Member` erstellen, brauchen wir einen globalen Zähler, der die Arbeiter zählt. Wie bereits erwähnt gibt es in Java keine globalen Variablen. Instanzvariablen sind nur in der jeweiligen Instanz global. Mit Instanzvariablen, wie wir sie in den Zeilen 2 bis 4 erstellt haben, ist es somit unmöglich die Anzahl der Arbeiter zu zählen. Wird eine neue Instanz der Klasse `Member` erstellt, so sind die Variablen `adress`, `lastName` und `salary` in allen Instanzen verschieden. Für diese Attribute soll das auch so sein, jeder Arbeiter wird wahrscheinlich eine andere Adresse, einen anderen Name und unter Umständen ein anderes Einkommen haben. Um aber trotzdem eine Art globalen Zähler zu erzeugen, verwendet man Klassenvariablen. `number` ist eine Klassenvariable.

Das Schlüsselwort `static` gibt an, daß es sich bei der nachfolgenden Variablen um eine Klassenvariable handelt. Klassenvariablen sind für jede Instanz einer Klasse gleich. Es ist nicht nötig, eine Instanz der Klasse zu erstellen, um den Wert einer Klassenvariable zu ermitteln. Nehmen wir an, wir erstellen von der Klasse `Member` die Instanzen `Member1` und `Member2`.

Nun setzen wir in der Instanz `Member1` die Variable `number` auf `5`, so ist der Wert der Variablen `number` in dem Objekt `Member2` ebenfalls `5`. Klassenvariablen sind in der Klasse und somit auch in jeder ihrer Instanzen global. Es existiert immer nur eine Variable für alle Instanzen. Damit läßt sich also mit der Variablen `number` ein Zähler für die Anzahl der Instanzen der Klasse `Member` implementieren.

Wie greift man nun auf eine Klassen- bzw. Instanzvariablen zu. In Listing 4.3 wird in Zeile 2 die Instanz `m1` der Klasse `Member` erstellt. Die Klasse `Member` hat als Attribut `salary`, somit ist `salary` jetzt eine Instanzvariable von `m1`. Der Zugriff auf `m1` erfolgt mittels `m1.salary` wie in Zeile 4. Vor dem Punkt steht der Objektname und nach dem Punkt der Name des Attributes.

```
1:    public boolean isGreaterThan (Object obj1, Object obj2) {
2:                Member m1 = (Member) obj1;
3:                Member m2 = (Member) obj2;
4:                if (m1.salary < m2.salary) {
5:                      return true;
6:                }
7:                return false;
8:          }
```

Listing 4.3 Zugriff auf Klassen- und Instanzvariablen

Es kann durchaus vorkommen, daß die Variable, auf die sie zugreifen wollen, Instanzvariable eines Objektes ist, welches ebenfalls Objekt eines Objektes ist. In diesem Fall schreibt man die Hierarchiestufen, durch einen Punkt getrennt hintereinander.

```
result = Superklasse.Klasse.Subklasse.ObjektDerSubklasse;
```

Ist die Variable, auf die sie zugreifen wollen eine Klassenvariable, so können wir an Stelle des Objektnames auch gleich den Klassennamen angeben. In diesem Fall ist es auch aus Gründen der Übersichtlichkeit üblich, den Namen der Klasse zu verwenden.

4.1.3 Konstanten

Der Begriff der Konstanten sollte aus anderen Programmiersprachen wie C oder PASCAL bekannt sein. Um in Java eine Konstante zu deklarieren, deklariert man eine Variable und kennzeichnet sie mittels des Schlüsselwortes `final` als Konstante. Man beachte, daß man in Java Konstanten nur für Instanz- und Klassenvariablen nicht aber für lokale Variablen erstellen kann.

In Listing 4.2 sind die Variablen `cityName` und `zipCode` als Konstanten gekennzeichnet. In Java gibt es kein `#define`, wie es aus der Sprache C bekannt ist. Mittels `final` kann man aber die gleiche Funktionalität erreichen.

Der Zugriff auf Konstanten erfolgt genauso wie der Zugriff auf eine Variable.

4.1.4 Das Schlüsselwort this

Das Schlüsselwort `this` wird verwendet, um auf das aktuelle Objekt Bezug zu nehmen. Die Methode `setLastName` ist eine Eigenschaft der Klasse `Member`. Im Listing 4.2 wurde das Attribut `lastName` deklariert. Um nun auf dieses Attribute zu verweisen, verwendet man das Schlüsselwort `this`. An dieser Stelle ist es egal, ob in der Methode eine lokale Variable mit dem gleichen Namen

```
1:    void setLastName (String lastName) {
2:          this.lastName = lastName;
3:    }
```

Listing 4.4 Schlüsselwort this

existiert. Mittels dem Schlüsselwort `this` wird Java mitgeteilt, daß nicht die lokale Variable sondern die globale Instanzvariabel verwendet werden soll. Ist in der Methode keine gleichnamige Variable vorhanden, reicht es aus, nur den Attributnamen anzugeben, um mit der Variable arbeiten zu können. Aus Gründen der Übersichtlichkeit sollte voralllem bei längeren Prozeduren das Schlüsselwort `this` verwendet werden.

4.1.5 Instanz- und Klassenmethoden

In Java können Methoden nur in Klassen angelegt werden. Eine Methode darf sich nie außerhalb eine Klasse befinden. Listing 4.4 zeigt die Definition einer Methode. Eine Methode besteht immer aus:

?? dem Namen der Methode,

?? dem Objekt oder Grundtyp, den die Methode zurückgibt,

?? den Parametern und

?? dem Methodenkörper.

Die Kombination der ersten drei Punkte bezeichnet man als Signatur einer Methode. Der Name der Methode muß nicht eindeutig sein, d.h. in einer Klasse dürfen mehrere Methoden mit dem gleichen Namen existieren. Erst wenn sie die gleiche Signatur haben, gibt der Compiler einen Fehler aus. Dieses Thema wird noch einmal im Abschnitt 4.3.1 genauer betrachtet.

In Listing 4.4 ist `setLastName` der Name der Methode, `lastName` ein Parameter und `void` der Rückgabetyp. Der Methodenkörper wird immer in einem Block zusammengefaßt. Sollen einer Methode mehrere Parameter übergeben werden, so müssen diese durch eine Komma voneinander getrennt werden.

Der Aufruf einer Methode erfolgt ähnlich dem Zugriff auf eine Variable. Wird die Methode aus einer Methode der gleichen Klasse aufgerufen, so reicht die Angabe des Methodennamens. Um unnötiges Suchen im Sourcecode zu vermeiden, ist es auch hier angebracht, das Schlüsselwort `this` zu verwenden. Es hat hier die gleiche Bedeutung wie bei Variablen. Die Methode eines Objekts wird mittels `Objektname.Methodenname(Parameter)` aufgerufen.

Ebenso wie es in Java Klassenvariablen gibt, gibt es Klassenmethoden. Klassenmethoden sind wie Klassenvariablen global und können ohne eine Instanz der Klasse verwendet werden. Um eine Klassenmethode zu definieren, verwendet man das Schlüsselwort `static`. In den Klassenbibliotheken von Java gibt es viel Klassenmethoden. Zum Beispiel die Klasse `Math` definiert zahlreiche mathematische Operationen, die in jedem Programm mit verschiedenen Zahlentypen genutzt werden können:

```
float root = Math.sqrt(453.0);
```

In dem Anwendungsbeispiel aus dem Bericht sind alle Methoden der Klasse `Database` Klassenmethoden. Die Klasse `FirstApplet` verwendet alle Methoden der Klasse `Database`, hat aber zu keiner Zeit eine Instanz der Klasse erstellt.

4.2 Konvertieren von Objekten und Primitivtypen

Um einen Wert von einem Typ in einen anderen zu konvertieren, verwendet man einen Mechanismus namens Casting. In Java ist es möglich, primitive Typen (`int, float, boolean`) als auch Objekte (`String`, `Point`, `Window` usw.) zu konvertieren. Daraus ergeben sich drei Formen von Casting.

?? Konvertieren zwischen primitiven Typen.

?? Konvertieren zwischen Objekttypen, d.h. eine Klasseninstanz wird auf eine Instanz einer anderen Klasse abgebildet.

?? Konvertieren von primitiven Typen in Objekte und Herausziehen von primitiven Werten, um sie dem Objekten zurückzugeben.

4.2.1 Konvertieren von Primitivtypen

Bei dieser Form des Casting kann ein Wert eines primitiven Typs in einen anderen primitiven Typ umgewandelt werden. Numerische Typen werden häufig untereinander umgewandelt z.B. eine Ganzzahl in eine Gleitkommazahl. Bei Variablen vom Typ `boolean`, wird `true` bzw. `false` in 0 bzw. 1 umgeformt. Die Umwandlung einer Variable eines numerischen Typs in eine Variable vom Typ `boolean` ist nicht möglich.

Ist der angestrebte Typ „größer“ als der zu konvertierende Typ, muß in der Regel kein explizites Casting angewendet werden. Es reicht den Wert der ursprünglichen Variable der Variable des neuen Typs zuzuweisen. Da der neue Typ mehr Informationen speichern kann, gehen beim Konvertieren auch keine Informationen verloren.

Um einen großen Wert in einen kleineren Typ zu konvertieren, muß das Casting explizit angewendet werden. Erfolgt die Anwendung nicht, besteht die Gefahr, daß relevante Informationen verloren gehen. Explizites Casting hat die folgende Form:

```
ValueInNewTyp = (typname) oldValue
```

`typname` ist der Datentyp, in den der Wert `oldValue` konvertiert werden soll. `oldValue` kann auch ein Ausdruck sein, wobei an dieser Stelle zu beachten ist, daß das Casting eine ziemlich hohe Präzedenz hat. Es ist dann unter Umständen nötig, den Ausdruck in zu klammern.

4.2.2 Konvertieren von Objekten

Das Konvertieren von Objekten unterliegt der Einschränkung, daß die beiden Objekte durch Vererbung miteinander verbunden sein müssen. Das Gebiet der Vererbung in Java wurde zwar bis hierher noch nicht behandelt, folgt aber im anschließenden Abschnitt. Die Grundlagen zur Vererbung sollten aus dem vorangegangen Bericht von Alek Opitz bekannt sein.

Wird eine Instanz einer Superklasse in eine Instanz einer ihrer Subklassen konvertiert, ist es nicht nötig, explizit das Casting anzugeben. Da die Subklassen normalerweise bedeutend mehr Informationen enthält als ihre Superklasse, somit gehen an dieser Stelle keine Informationen verloren.

```
1: String getRightName (int i) {
2:     String name = "";
3:
4:     if (memberArray[i] instanceof OwnCompany) {
5:           OwnCompany ofHelp = (OwnCompany) memberArray[i];
6:           name = ofHelp.getSectionName();
7:     }
8:     else if (memberArray[i] instanceof ForeignCompany) {
9:           ForeignCompany ffHelp = (ForeignCompany) memberArray[i];
10:          name = ffHelp.getCompanyName();
11:    }
12:    return name;
13:}
```

Listing 4.5 Casting von Objekten

Ein Beispiel für das Casten von Objekten ist, im Listing 4.5 Zeile 5 zu sehen. Die Informationen der Mitarbeiter werden in der Klasse `FirstApplet` in einem Array namens `memberArray`, dessen Elemente vom Typ `Member` sind, abgelegt. In der Methode `addEntry` werden Objekte vom Typ `ForeignCompany` sowie vom Typ `OwnCompany` als Elemente in das Array `memberArray` eingefügt. In der Funktion `getRightName` aus Listing 4.5 bekommen wir nur den Index für das Array übergeben. Wir wissen also nicht, von welcher Klasse das aktuelle Element instanziiert wurde. Prinzipiell haben wir ein Array von `Member` – Elementen erstellt. Mittels des Schlüsselwortes `instanzof` können wir ermitteln, von welcher Klasse ein bestimmte Instanz ist (siehe Zeile 4). Um z.B die Funktion `getSectionName` nutzen zu können, müssen wir den Eintrag, so er eine Instanz von `OwnCompany` ist, konvertieren. An dieser Stelle (Zeile 5) wenden wir das Casten von Objekten an.

Durch das Konvertieren einer Subklasse in eine Instanz ihrer Superklasse gehen Informationen verloren, weshalb es nötig ist, das explizite Casting anzuwenden. Die Syntax für das Konvertieren von Objekten ist genauso wie bei Konvertieren von primitiven Datentypen.

Abgesehen vom Konvertieren der Objekte einer Klassen ist es außerdem möglich, Objekte von Schnittstellen zu konvertieren, jedoch unter der Einschränkung, daß die Klasse oder eine ihrer Superklassen die Schnittstelle implementiert. Dadurch kann erreicht werden, daß man eine Methoden dieser Schnittstelle verwenden kann, ohne diese direkt zu implementieren.

4.2.3 Konvertieren von Primitivtypen in Objekte und umgekehrt

Eigentlich ist es in Java unmöglich, eine dieser Konvertierungen vorzunehmen, allerdings ist im Paket `java.lang` für jeden primitiven Datentyp eine entsprechende Klasse definiert. Mit diesen Klassen und dem Operator `new` ist es dann möglich, ein Objekt von Typ `Integer` zu erstellen.

```
Integer intObjekt = new Integer(45);
```

Die obige Zeile erzeugt ein solches Objekt mit dem Wert `45`. Die Klassen der primitiven Datentypen enthalten außerdem Methoden, mit den es möglich ist, das Objekt in einen primitiven Wert zu konvertieren.

```
int theInt = intObject.intValue();
```

4.3 Objektorientiertes Design mit Java

4.3.1 Vererbung

In Java dient das Schlüsselwort `extends` dazu, die Vererbung zu implementieren. Eine Vererbung stellt im allgemeinen eine Erweiterung (engl. Extention) der Vaterklasse dar. Die Klassen `ForeignCompany` und `OwnCompany` sollen die Klasse `Member` um eine kleine Anzahl von Attributen und Eigenschaften erweitern. Somit werden `OwnCompany` als auch `ForeignCompany` von der Klasse `Member` alle Eigenschaften und Attribute erben.

```
1:    class OwnCompany extends Member {
2:          String sectionName;
3:
4:          OwnCompany (String sectionName) {
5:                super();
6:                this.sectionName = sectionName;
7:          }
8:
9:          String getSectionName () {
10:         return sectionName;
11:         }
12:   }
```

Listing 4.6 Vererbung und das Schlüsselwort super

Listing 4.6 zeigt die Klassendefinition von `OwnCompany`. Gleich nach dem Klassennamen wird mittels `extends` angeben, von welcher Klasse die neu Klasse erben soll. Wird nichts angeben, so erbt die neu Klasse von der Klasse `Object`. In Java ist die Klasse `Object` die Mutter aller Klassen. Sie besitzt die Methoden `equals`, `getclass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `clone` und `finalize` und ist im Paket `java.lang` enthalten. Ich werde in diesem Bericht nicht auf die Bedeutung dieser Methoden eingehen. Es sei nur soviel erwähnt, daß jede Klasse Subklasse der Klasse `Object` ist und somit jeder Klasse diese Methoden zur Verfügung stehen.

Da die Klasse `OwnCompany` eine Subklasse von `Member` ist, stehen `OwnCompany` die selben Element wie `Member` zur Verfügung. Wir könne jetzt zum Beispiel eine Instanz `ownCompany1` von `OwnCompany` erstellen und der Variablen `ownCompany1.lastName` einen Wert zuweisen, obwohl sie in der Klasse `Member` deklariert wurde. `lastName` ist jetzt auch ein Attribut der Klasse `OwnCompany`.

Man betrachte die Zeile 5 im Listing 4.6. An dieser Stelle wird das Schlüsselwort `super` verwendet. Das Schlüsselwort `super` ist ähnlich dem Schlüsselwort `this`. Mit `this` wird Java mitgeteilt, daß an dieser Stelle ein Attribute oder eine Methode der eigenen Klasse verwendet wird. Mittels `super` gibt man an, daß man auf ein Element der Superklasse zugreifen will.

Hinter dem Schlüsselwort `extends` kann immer nur eine Klasse stehen. In Java ist es nicht möglich, daß eine Klasse von zwei oder mehreren Superklassen erbt. Java unterstützt das Konzept der Mehrfachvererbung in dieser Weise nicht. In dem Anwendungsbeispiel erbt die Klasse `FirstApplet` zwar von `Applet` und von `SortInterface`, jedoch handelt es sich bei `SortInterface` um eine Schnittstelle. Will man die Attribute und die Eigenschaften von mehreren Klassen, die nicht selbst in einer Vererbungsbeziehung stehen erben, so muß man auf die Nutzung von Schnittstellen zurückgreifen. Das Konzept der Schnittstellen wird im Abschnitt 4.5 behandelt.

4.3.2 Reichweite und Accessor - Methoden

Ein Klasse soll seine Elemente von anderen Klassen verbergen können, man spricht hier von der Kontrolle der Sichtbarkeit. Bis jetzt haben wir alle Elemente in unseren Klassen gleich behandelt. Prinzipiell konnten bis jetzt alle Klassen auf alle Elemente jeder Klasse zugreifen.

Um dies zu vermeiden, gibt es in Java vier verschiedene Schutzebenen: `public`, `package`, `protected` und `privat`. Mittels diesen vier Schutzebenen kann man die grundlegenden Beziehungen, die eine Methode oder Variable einer Klasse mit einer anderen Klasse haben kann, definieren.

Eine Variable wird als `protected` definiert, in dem man vor die Typenbezeichnung das Schlüsselwort `protected` schreibt. Zum Beispiel wird durch folgende Zeile

```
protected String lastName;
```

die Variable `lastName` (in unserem Beispiel aus der Klasse `Member`) als `protected` definiert.

`public`

Mittels `public` können Methoden und Attribute für alle Klassen sichtbar gemacht werden. Wird zum Beispiel eine Variable `v1` als `public` definiert, können alle Klassen auf die Variable `v1` zugreifen, wenn sie in irgendeiner Beziehung zu der Klasse, in der `v1` deklariert wurde, stehen.

`package`

Eine ähnliche Notation ist in C++ vorhanden, mit dieser können Funktionen innerhalb einer bestimmten Quelldatei aufeinander zugreifen. In Java werden Funktionen nicht mittels ihrer Angehörigkeit zu Quelldateien zusammengefaßt sondern durch Pakete. Pakete sind explizite Notation zum Gruppieren von Klassen. Mehr über Pakete steht im Abschnitt 4.5.2.

Wird einer Variablen keine Schutzebene zugewiesen, ist sie automatisch als `package` definiert. Für Klassen und deren Inhalte, die in einem Paket liegen und als `package` definiert sind, gelten die gleichen Zugriffsrechte als wären sie `public`. Klassen fremder Pakete haben auf diese Elemente jedoch keine Zugriffserlaubnis. Das Schlüsselwort `package` (getestet in Microsoft J++ Java 1.0) darf nicht explizit zum Definieren des Zugriffsrechtes einer Variablen oder Methode verwendet werden. Die Definition erfolgt durch das Weglassen der Schutzdefinition.

`privat`

In Java ist `privat` die stärkste Zugriffsbeschränkung für Elemente. Ist eine Variable als `privat` definiert, kann sie nur noch von der Klasse selbst verwendet werden. Es besteht keine Möglichkeit eine Variable oder eine Methode, die als `privat` gekennzeichnet ist, außerhalb der Klasse zu verwenden. Wird eine Klasse vererbt, werden alle Element mit dieser Zugriffsbeschränkung nicht mit vererbt. Es besteht somit auch nicht die Möglichkeit, von einer Subklasse auf `privat` – Element ihrer Superklassen zuzugreifen.

`protected`

Die Zugriffsebene `protected` ist ähnlich `privat`. Der einzige Unterschied besteht darin, daß `protected` – Elemente an Subklassen mit vererbt werden. Ansonsten gelten für diese Elemente die gleichen Zugriffsbeschränkungen.

Accessor – Methoden

Wird ein Programm konsequent objektorientiert geschrieben, so werden alle Attribute, die die Eigenschaften einer Klasse beschreiben, `protected` oder `private` gekennzeichnet. Um aber trotzdem den Zugriff auf eine solche Variable zu gewährleisten, bedient man sich sogenannter Accessor – Methoden.

```
1:     final class ForeignCompany extends Member {
2:           private String CompanyName;
3:
4:           public  ForeignCompany (String CompanyName) {
5:                 super();
6:                 this.CompanyName = CompanyName;
7:           }
8:
9:           public String getCompanyName() {
10:                return CompanyName;
11:          }
12:    }
```

Listing 4.7 Accessor Methoden

Listing 4.7 zeigt die vollständige Implementierung der Klasse `ForeignCompany`. `ForeignCompany` ist genauso wie `OwnCompany` eine Subklasse von `Member`. Das Attribut `CompanyName` ist als `privat` definiert. Um `CompanyName` zu verwenden, sind die Methoden `ForeignCompany` und `getCompanyName` vorhanden. Beide Funktionen sind als `public` gekennzeichnet und können somit von außerhalb verwendet werden. Auf dem Umweg über diese beiden Funktionen kann die Variable `CompanyName` gesetzt bzw. ausgelesen werden.

4.3.3 Der final – Modifier

Der `final` – Modifier kann auf Klassen, Variablen und Methoden angewendet werden.

?? Wird der `final` – Modifier auf eine Klasse angewendet, bedeutet das, daß von dieser Klasse keine Subklassen erstellt werden können.

?? Nach der Anwendung auf eine Methode, kann die Methode von einer Subklasse nicht mehr überschrieben werden.

?? Bei Variablen bedeutet die Anwendung des `final` – Modifiers, daß die Variable konstant ist.

`final` – Klassen

Im Listing 4.7 ist die Klasse `ForeignCompany` als `final` gekennzeichnet. Im kompletten Programm ist auch die Klasse `OwnCompany` als `final` definiert. Beide Klassen erben von der Klasse `Member` und haben spezifische Eigenschaften, die nur für unser Applet von Bedeutung sind. An dieser Stelle können die Klassen beliebig optimiert werden. Es besteht nicht die Gefahr, daß spätere Subklassen mit diesen Optimierungen Schwierigkeiten bekommen. In beiden Klassen wurde der Konstruktor (die Funktion `ForeignCompany`) auch zum Setzen der Variable `lastName` verwendet. Zum Konstruktor sein an dieser Stelle nur soviel gesagt, daß es hierbei sich um eine spezielle Methode in einer Klasse handelt. Da wir aber genau wissen, daß die Klasse `ForeignCompany` nicht weiter vererbt werden kann, könnten wir an dieser Stelle den Konstruktor auch mit spezielleren Operationen versehen.

`final` – Methoden

Die Besonderheit von `final` – Methoden hängt mit der Polymorphie zusammen und wird deswegen im Abschnitt 4.4.1 beschrieben.

`final` – Variablen

Wird eine Variable als `final` definiert, handelt es sich um eine Konstante. Das Thema der Konstanten wurde bereits im Abschnitt 4.1.3 behandelt.

4.3.4 abstract - Klassen und Methoden

In der objektorientierten Programmierung geht man davon aus, daß in einer Vererbungshierarchie die höheren Klassen abstrakter und allgemeiner sind. Durch mehrfaches Vererben, in dem Sinn das eine Klasse von einer anderen erbt und diese Klassen wieder weiter vererbt wird, werden die Subklassen immer spezifischer und konkreter. Die Subklasse enthält dann die selben Design- und Implementierungsmerkmale, solange sie nicht als `privat` definiert wurden wie ihre Superklasse. Ist der einzige Grund, warum eine Superklasse erstellt wurde, die Optimierung von Code bzw. Speicherplatz, d.h. die Klasse beinhaltet nur Attribute und Eigenschaften, die in mehreren anderen Klassen identisch benötigt werden, so nennt man eine solche Superklasse eine `abstract` – Klasse.

Von `abstract` – Klassen können keine Instanzen erstellt werden, was eigentlich auch nicht nötig ist, da man davon ausgeht, daß Subklassen existieren, die die Methoden der `abstract` – Klassen gemeinsam nutzen. Eine `abstract` – Klasse kann alles enthalten, was in einer normalen Klasse auch stehen kann. Außerdem können Methoden mit dem Präfix `abstract` definiert werden. Solche `abstract` – Methoden brauchen keine Implementierung. Sie stellen nur eine Maske der

```
1:    abstract class Member {
2:          protected String lastName;
3:
4:          public abstract void setLastName (String lastName);
5:    }
6:
7:    final class OwnCompany extends Member {
8:          public abstract void setLastName (String lastName) {
9:                this.lastName = lastName;
10:   }
```

Listing 4.8 Abstract - Klassen und Methoden

Methode dar. In den nicht abstrakten Subklassen müssen diese Methoden dann implementiert werden.

Die Klasse `Member` ist eine `abstract` – Klasse. Alle Funktionen aus der Klasse `Member` würden wir in den Klassen `OwnCompany` und `ForeignCompany` identisch implementieren. Beide Klassen haben aber noch verschiedene Attribute und Eigenschaften, die nur für ihre Klasse von Bedeutung sind. In der Klasse `Member` können wir diese Eigenschaften nicht zusammenfassen, somit ist nur eine Verbindung von `Member` und `OwnCompany` bzw. `ForeignCompany` für unser Programm sinnvoll. Von der Klasse `Member` würden wir nie eine Instanz erstellen, da sie nicht alle nötigen Attribute beinhaltet. Aus diesem Grund ist in unserem Beispiel `Member` eine abstrakte Klasse.

Die Funktion `setLastName` ist im Listing 4.8 eine `abstract` – Methode. In der Klasse `Member` (Zeile 4) ist nur der Prozedurkopf eingefügt, die eigentliche Implementierung erfolgt erst in der nicht abstrakten Subklasse `OwnCompany`. Da wir `setLastName` in beiden Subklassen mit der identisch Funktionalität benötigen, ist im Orginalprogramm die Funktion `setLastName` nicht abstrakt und in der Klasse `Member` implementiert.

4.4 Mehr über Methoden

4.4.1 Konstruktor - Methode

In dem Bericht ist der Konstruktor in den Listings und im Text schon oft verwendet worden. Die Erklärung erfolgt an dieser Stelle.

Eine Konstruktor – Methode ist eine spezielle Methode, die beim Erstellen der Klasse bestimmt wie ein Objekt instanziiert wird. Sie wird immer dann aufgerufen, wenn eine Instanz der Klasse erstellt wird. Es gibt keine Möglichkeit eine Konstruktor – Methode direkt aufzurufen. Wird mittels `new` eine Instanz einer Klasse erstellt, wird dem Objekt zunächst Speicher zugewiesen. Danach werden die Instanzvariablen des Objektes auf ihre Anfangswerte oder auf einen Standardwert instanziiert. Anschließend wird der Konstruktor der Klasse aufgerufen.

```
1:     abstract class Member {
2:           protected static int number;
3:
4:           public Member () {
5:                 Member.number++;
6:     }
7:
8:     final class OwnCompany extends Member {
9:           private String sectionName;
10:
11:          public OwnCompany (String sectionName) {
12:                super();
13:                this.sectionName = sectionName;
14:          }
15:    }
```

Listing 4.9 Konstruktor – Methoden (Auszug aus den Klassen Member und OwnCompany)

Es ist nicht nötig für jede Klasse einen Konstruktor anzulegen. Man kann auch auf in verzichten und erhält trotzdem ein Objekt. Dann wird es im allgemeinen aber nötig, andere Methoden „von Hand“ aufzurufen, um die Klasse zu initialisieren.

Konstruktor – Methoden sehen normalen Methoden sehr ähnlich. Sie habe nur zwei Besonderheiten.

?? Eine Konstruktor – Methode besitzt nie einen Rückgabewert und somit auch keine `Return` – Anweisung

?? Konstruktoren tragen den selben Namen wie ihre Klasse.

Im Listing 4.9 haben die Klasse `Member` und die Klasse `OwnCompany` je eine Konstruktor. Das „je“ ist wichtig, da eine Klasse auch mehrere Konstruktoren besitzen kann. Dazu mehr im nächsten Abschnitt. Der Konstruktor aus der Klasse `Member` dient dazu, die Klassenvariable `number` zu inkrementieren. Wir erinnern uns, die Variable `number` haben wir eingeführt, um die Anzahl unserer Mitglieder zu zählen. Jedesmal wenn eine neue Instanz der Klasse `Member` oder eine Instanz seiner Subklassen erstellt wird, wird die Klassenvariable `number` inkrementiert. Das ist auch schon die einzige Funktion, die der Konstruktor hat. In der Subklasse `OwnCompany` ist ein neuer Konstruktor definiert. Würden wir an dieser Stelle keinen neuen Konstruktor definieren, wäre der Konstruktor aus der Superklasse übernommen worden. In dem Moment, wo wir eine neue Konstruktor – Methode definiert haben, haben wir den alte Konstruktor überschrieben. Das Überschreiben von Methoden wird ebenfalls im nächsten Abschnitt genauer erklärt. An dieser Stelle ist nur wichtig, daß der neue Konstruktor an die Stelle von dem Konstruktor aus der Superklasse `Member` getreten ist.

Da wir aber nur Instanzen von den Subklassen von `Member` erstellen, brauchten wir an dieser Stelle die Funktionalität des alten Konstruktors, um die Anzahl der Mitglieder zu zählen. Es wäre also ganz schön, wenn wir ihn zusätzlich in der Superklasse `Member` aufrufen könnten. Obwohl wie oben beschrieben, es eigentlich keine Möglichkeit gibt, Konstruktoren direkt aufzurufen, können wir mittels `super()` wie in Zeile 12 (Listing 4.9) den Konstruktor der Superklasse aufrufen. Auf diese Weise können wir in einer Subklasse die Funktionalität der Konstruktor – Methode aus der Superklasse erweitern. In der objektorientierten Programmierung ist dies eine häufige angewendete Methode, um den Konstruktor einer Superklasse zu erweitern, ohne Code kopieren zu müssen.

Oben wurde bereits erwähnt, daß eine Klasse mehrere Konstruktor – Methoden besitzen kann. Soll aus einem Konstruktor ein anderer Konstruktor der selben Klasse aufgerufen werden, so kann dies mittels `this(arg1,...,argn)` realisiert werden.

4.4.2 Finalize – Methoden

`finalize` – Methoden sind in gewissem Sinn die Gegenstücke zu den Konstruktor – Methoden. In einer Konstruktor – Methode wird das Objekt instanziiert und in einer `finalize` – Methode können noch einige „Reinigungsarbeiten“ getan werden, bevor das Objekt zerstört wird.

```
void finalize() {
      // irgendwelche Aufräumarbeiten
}
```

Bei der `finalize` – Methode handelt es sich um eine gewöhnliche Methode, die jederzeit auch von Hand aufgerufen werden kann. Mit dem Aufruf einer `finalize` – Methode von Hand wird allerdings nicht erreicht, daß das Objekt zerstört wird.

Eine `finalize` – Methode unterliegt gewissen Einschränkungen. Es ist z.B. nicht immer gewährleistet, daß die Methode aufgerufen wird, bevor der Speicher des Objektes tatsächlich zurückgefordert werden kann, was einige Zeit dauern kann, nachdem alle Referenzen auf das Objekt entfemt wurden.

4.4.3 Polymorphismus

Polymorphismus ist die Fähigkeit einer Subklasse, das Verhalten der geerbten Methoden basierend auf den Bedürfnissen der Klasse und den übergebenen Parametern anzupassen. Polymorphismus kann in Java dadurch erreicht werden, daß Attribute und Methoden überschrieben oder Methoden überladen werden.

Attribute und Methoden überschreiben

Klassenattribute werden vom Namen überschrieben. Dies wird eigentlich nur getan, um den Datentyp, die Reichweite, den Schutz oder den Wert einer Konstanten in einer Subklasse zu verändern.

Man betrachte folgenden Code:

```
protected float salery; // in der Klasse Member

privat int salery;      //in der Klasse ForeignCompany
```

Da unsere Mitarbeiter immer ganzzahlige Gehälter bekommen, erstellen wir in der Klasse `Member` ein Attribut `salary` vom Typ `int`. Die Mitarbeiter fremder Firmen bekommen aber Gehälter mit Pfennigbeträgen, also überschreiben wir in der Klasse `ForeignCompany` das Attribut `salary`, um eine Attribut `salary` von Typ `float` zu erhalten. Der obige Codeauszug zeigt wie dies gemacht werden kann.

Um eine Eigenschaft zu überschreiben, definiert man in der Subklasse die Methode mit der selben Signatur neu. Danach kann man den Methodenkörper beliebig verändert. Wird im Programm die Methode aufgerufen, nutzt Java die neue Variante. Eigentlich wird dies nur genutzt um den Code einer Klasseneigenschaft zu optimieren. Es ist gefährlich eine Methode zu überschreiben. Wird zum Beispiel von unserer Klasse eine weitere Subklasse erstellt und der Programmierer der Subklasse weiß nicht, daß wir die Funktion überschrieben haben, so kann dies zu schwer erkennbaren Problem führen. Wird eine Eigenschaft überschrieben sollte man sicher stellen, daß das Ergebnisse der ursprünglichen Methode mit dem Ergebnis der neuen Methode übereinstimmt. Ein weitere Möglichkeit um solche Probleme zu vermeiden ist, die Klasse als `final` zu definieren und damit ein Vererben unmöglich zu machen.

Überladen von Funktionen

In Java erfolgt der Aufruf einer Methode nicht allein über den Namen sondern auch über den Rückgabewert und die Parameter. Listing 6.10 enthält die Methode `setAdress` zweimal.

```
1:  abstract class Member {
2:
3:     private final String cityName = "Cottbus";
4:     private final String zipCode = "03046";
5:
6:     public final void setAdress(String streetName, String cityName,
7:           String zipCode) {
8:           this.adress = streetName + " " + zipCode + " " + cityName;
9:     }
10:
11:    public void setAdress(String streetName) {
12:          this.adress = streetName + " " + this.zipCode + " "
13:                              + this.cityName;
14:    }
15: }
```

Listing 6.10 Überladen von Funktionen

In unserem Beispiel waren nur für die Mitarbeiter die Heimatstadt und die dazugehörige Postleitzahl gespeichert, wo sich der Inhalt von der Adresse der Firma unterscheidet. Die Konstanten `cityName` und `zipCode` sind in der Klasse `Member` als Standardwerte enthalten und beinhalten die Adresse unserer Firma. Wird die Methode `setAdress` nur mit einem Argument aufgerufen, wird die zweite Methode verwendet, weil sie die passende Signatur zum Methodenaufruf hat.

Die Überladung (Overloading) von Methoden kann ziemlich nützlich sein, um Standardelementen die Vereinfachung von Methodenaufrufen in Klassen zu erleichtern. Außerdem kann die Überladung von Methoden dazu eingesetzt werden, verschiedene Datentypen mit einem Methodenaufruf zu behandeln. Java unterscheidet Overloading – Methoden mit gleichem Namen

anhand der Zahl und dem Typ der Parameter für die jeweilige Methode, nicht anhand der Rückgabewert. So kann Java zwei Methoden mit gleichem Namen und gleicher Parameterliste, jedoch unterschiedlicher Rückgabewert nicht unterscheiden.

Damit wird auch klar, warum eine Klasse mehrere Konstruktoren besitzen kann. Genauso wie jede andere Methode kann auch der Konstruktor einer Klasse überladen werden. Beim Instanziieren der Klasse wird dann anhand der übergebenen Parameter entschieden, welcher Konstruktor verwendet werden soll.

4.4.4 final – Methoden

Wird eine Klasse als `final` deklariert, so kann diese Klasse nicht weiter vererbt werden. Bei `final` – Methoden ist das ähnlich. Eine `final` – Methode kann nicht mehr überschrieben werden. Normalerweise werden Methoden als `final` deklariert, wenn die Superklasse versucht, ihre eigenen Variablen zu schützen. Im der Klasse `Member` dient eine der Funktionen `setAdress` zum Zusammenbauen des `adress` – Attributes. Da wir nicht sicher sein können, daß in allen Subklassen, die von `Member` erstellt werden, genau diese Formatierung wiederverwendet werden soll und die nötigen Konstanten vorhanden sind, definieren wir die nicht optimierte `setAdress` – Methoden als `final`.

4.5 Schnittstellen und Pakete

4.5.1 Das Konzept der Schnittstellen

Wie bereits im Abschnitt 4.3.1 zum Thema Vererbung erwähnt, gibt es in Java nicht die Mehrfachvererbung von Klassen. Es ist somit unmöglich, daß eine Klasse die Attribute und Eigenschaften von zwei Klassen erbt, die nicht selbst in einer Vererbungsbeziehung stehen.

Um aber nicht ganz auf das Konzept der Mehrfachvererbung verzichten zu müssen, bietet Java die Möglichkeit mittels Schnittstellen, eine ähnliche Beziehung zwischen Klassen zu realisieren.

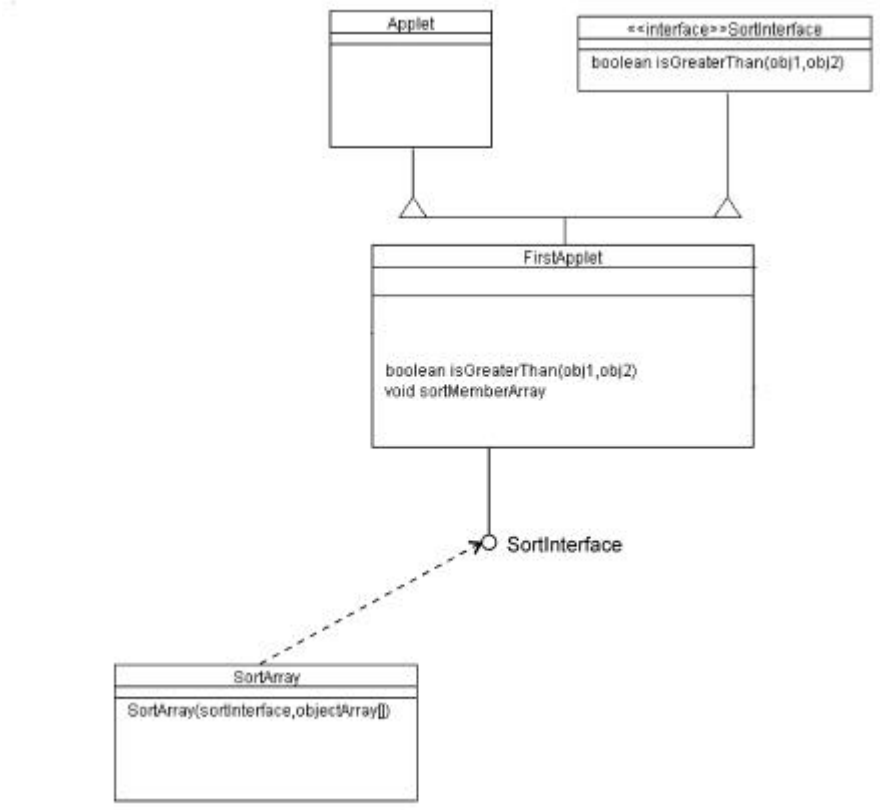

Abb. 6.1 Darstellung der Schnittstelle SortInterface

Eine Schnittstelle (`interface`) ist eine Sammlung von Methodennamen ohne Definitionen, die andeuten, daß eine Klasse außer dem von der Superklasse geerbten Verhalten zusätzlich Informationen hat.

Schnittstellen sind leere abstrakte Module und werden durch das Schlüsselwort `interface` definiert. Beim Arbeiten mit Schnittstellen sind einige grundlegende Punkte zu beachten.

?? Obwohl der Modifier `abstract` nicht auf den Klassennamen einer Schnittstelle angewendet wird, sind Schnittstellen immer abstrakt und können somit auch nicht instanziiert werden

?? Die Klassenattribute einer Schnittstelle müssen immer als `static` deklariert werden. Schnittstellen wurden dafür entwickelt Funktionalität zu enthalten und nicht um Klassen zu ersetzen.

?? Anders als bei Klassen darf eine Subklasse von mehreren Schnittstellen erben. Die einzelnen Schnittstellen werden mit einem Komma von einander getrennt.

```
public class ClassName extends SuperClass
        implements Interface1, Interface2, ... {
```

?? Schnittstellen können genau wie Klassen über folgende Syntax vererbt werden.

```
public interface Interface1 extends Interface2
        implements Interface3, Interface4, ... {
```

?? Schnittstellen müssen immer öffentlich zugänglich sein und dürfen nicht `final` sein, da sie von Natur aus abstrakt sind und ihr Sinn darin besteht, daß sie vererbt und implementiert werden.

?? Methoden von Schnittstellen können nicht als `native`, `static`, `synchronized` oder `final` deklariert werden. Sie sind außerdem abstrakt und dürfen somit keinen Programmcode enthalten.

Auf dem ersten Blick scheint der letzte Punkt die Funktionalität der Schnittstelle ein wenig in Frage zu stellen. Jedoch liegt der Grund eine Schnittstelle zu definieren darin, daß die abstrakte Definition dieser Schnittstelle in einer anderen Klasse als Datentyp verwendet werden kann. In Abbildung 6.1 erbt die Klasse `FirstApplet` von der Klasse `Applet` und von der Schnittstelle `SortInterface`. `SortInterface` definiert nur die Funktion `isGreaterThan`. Diese Funktion wird in `FirstApplet` implementiert. Eine weitere Klasse hier `SortArray` (die wir hätten beerben wollen, aber nicht konnten, weil Mehrfachvererbung in Java nicht möglich ist) bekommt die Schnittstelle `SortInterface` als Datentyp für ein Argument in einer (auch mehrere sind möglich) Methode übergeben.

```
1:    public interface SortInterface {
2:          public abstract boolean
3:                isGreaterThan(Object obj1, Object obj2);
4:    }
5:
6:    public class SortArray {
7:          public static void sort
8:                 (SortInterface si, Object objArray[]) {
9:                /*
10:               Implementierung eines Sortieralgorithmus der
11:               als Vergleichsmethode
12:                     si.isGreaterThan(arrayEntry1, arrayEntry2)
13:               verwendet
14:               */
15:         }
16:   }
17:   public class FirstApplet extends java.applet.Applet
18:            implements SortInterface {
19:
20:         private Member memberArray[];
21:
22:         public void sortMemberArray () {
23:               SortArray.sort(this, (Object []) memberArray);
24:         }
25:
26:         public boolean isGreaterThan (Object obj1, Object obj2) {
27:               Member m1 = (Member) obj1;
28:               Member m2 = (Member) obj2;
29:               if (m1.salary > m2.salary) {
30:                     return true;
31:               }
32:               return false;
33:         }
34:         ...
35:   }
```

Listing 4.11 Implementierung der Schnittstellle SortInterface

In Listing 4.11 ist die Implementierung von Abbildung 6.1 zu sehen. Zuerst wurde in der ersten Zeile mittels des Schlüsselwortes `interface` die Schnittstellenklasse `SortInterface` angelegt. In der Schnittstelle wurde dann die einzige Funktion `isGreaterThan` definiert. Die Klasse `SortArray` enthält genau den Algorithmus zum Sortieren, den wir in unserem Applet verwenden wollen. Da wir aber ein Applet erstellen, muß die Klasse `FirstApplet` von `Applet` erben, so daß wir den Algorithmus aus `SortArray` nicht einfach erben können. In der Methode `sort` (Zeile 7) von `SortArray` bekommt die Klasse `SortArray` die Schnittstelle `SortInterface` übergeben. In dieser Klasse ist die Methode `isGreaterThan` enthalten, die wir zum Sortieren des Arrays, welches als zweiter Parameter an die Methode `sort` übergeben wird, benötigen. Somit bleibt es der Klasse `FirstApplet`, welche die Methode `isGreaterThan` implementiert (Zeile 26-33), überlassen, die Sortierung des Arrays festzulegen. In unserem Beispiel werden die Mitarbeiter nach ihrem Gehalt in absteigender Reihenfolge sortiert. Zum Schluß ruft die Methode `sortMemberArray()` die Methode `sort` auf, die ein Argument vom Typ `SortInterface` und das Daten – Array erwartet.

Wenn eine Schnittstelle implementiert wird, sollte immer eine ähnliche Struktur entstehen. Eine Schnittstelle ist keine Superklasse, sie ist eher eine Brücke zwischen zwei Klassen, so daß die Funktionalität der einen Klasse von einer anderen Klasse implementiert werden kann.

4.5.2 Arbeiten mit Paketen

In C++ werden größere Programme fast immer in einzelne Dateien unterteilt, die dann mittels einer `include` – Anweisung in das Programm eingefügt werden. In Java werden die Klassen in

einzelne Pakete zusammengefaßt. Ein Paket ist nicht auf eine Datei beschränkt. Es kann durch aus vorkommen, daß zwei Klassen in einem Paket liegen, aber aus völlig verschieden Dateien stammen.

Bei der Arbeit mit Paketen sind ein paar Besonderheiten zu beachten. Alle Dateien die Klassen eines Paketes enthalten müssen in einem Verzeichnis liegen. Dieses Verzeichnis muß den selben Namen wie das Paket haben.

Ein Paket wird mittels des Schlüsselwortes `package` definiert. Die `package` – Anweisung muß ganz am Anfang einer `*.java` – Datei stehen. Die einzige Ausnahme bilden Kommentare, sie dürfen als einziges über `package` – Anweisung stehen. Alle Klassen die in dieser Datei definiert werden, sind dann in diesem Paket zusammengefaßt.

Warum sollte man Pakete auf verschiedene Dateien verteilen? Die Antwort ist, daß in einer Quelldatei immer nur eine Klasse oder eine Schnittstelle `public` sein darf. Demzufolge muß das Listing 6.11, in welchem beide Klassen und die Schnittstelle `SortInterface` `public` sind, auf drei Dateien aufgeteilt werden.

```
1:     package Example;
2:
3:     import java.awt.*;
4:     import sort.*;
5:     import sortInter.*;
6:     import database.*;
7:
8:     public class FirstApplet extends java.applet.Applet
9:           implements SortInterface {
10:       // ...
```

Listing 4.12 Pakete und die import - Anweisung

Zum Einen kann mittels der `import` – Anweisung ein Paket in eine Quellcode – Datei eingefügt werden. Die gesamte Klassenbibliothek von Java ist in Pakete unterteilt. In unserem Beispiel haben wird verschieden Pakete verwendet. Listing 6.12 zeigt alle mittels der `import` – Anweisung importierten Pakete. Die `import` – Anweisung steht gleich unter der `package` Anweisung, wenn eine vorhanden ist. Die Anweisung

```
import java.awt.*;
```

importiert alle Klassen die im `awt` – Paket zusammengefaßt sind. Aus diesem Paket verwenden wird die Klasse `graphics`. Man kann aber auch nur die Klasse `graphics` mit folgender Zeile importieren.

```
import java.awt.graphics;
```

In Listing 6.12 Zeile 8 wurde obige Möglichkeit Paket zu importieren angewendet. Man kann genauso gut die Klasse mit „dem gesamten Pfad“ angeben. Die Bezeichnung Pfad ist dem Sinn sogar richtig, da die Pakete in einer gleichen Hierarchie aufgebaut sind, wie die Verzeichnisse in denen sie liegen. Die Klasse `graphics` würde man unter `java/awt/“anyfile“.java` finden. Verwendet man eine Klasse nur einmal, ist es aus Gründen der Übersichtlichkeit besser, den gesamten Pfad anzugeben. Verwendet man eine Klasse öfter, müßte man an jeder Stelle den gesamten Pfad angeben. Aus Gründen der Effektivität wird man dann mittels der `import` – Anweisung die Klasse nur einmal einfügen.

Wenn man ein Programm mit eigenen Paketen schreibt, sollte man beachten, daß die Pfade so gesetzt sind, daß Java die Pakete auch finden kann. Normalerweise sind nur die Pfade für die java – eigenen Pakete gesetzt. Um selbst angelegte Pakete importieren zu können, muß man in der Entwicklungsumgebung die nötigen Pfade für das eigenen Projekt manuell setzen. Tut man das nicht, ist jedes Importieren eines eigenen Paketes zum Scheitern verurteilt.

5 Quellenverzeichnis und Literaturverweis

[Küh97] Ralf Kühnel. *Die Java-1.1-Fibel*. Addison-Wesley, Bonn [u.a.], 1997

[Lem96] Laura Lemay, Charles L. Perkins. *Java in 21 Tagen*. SAMS, Haar bei München, 1997

[Woo97] Charles Wood. Programmieren in Visual J++. Franzis, Feldkirchen, 1997

Zusätzliche Informationen zu Java findet man im World Wide Web unter:

http://www.javasoft.com

http://www.java.de

news:de.comp.lang.java

news:comp.lang.java.*

6 Anhang

FirstApplet.java

```
/*
Programmautor: Oliver Stecklina
Programmtitel: FirstApplet
Programmgeschichte: Geschrieben als bei Beispielprogramm für
       einen Bericht zu einem Java Vortrag.
       Der Bericht wurde im Rahmen eines Javaseminars erstellt
Dateiname: FirstApplet.java
*/

import java.awt.*;
import sort.*;
import sortInter.*;
import database.*;

public class FirstApplet extends java.applet.Applet
implements SortInterface {

       private Member memberArray[];

       private int maxEntry = 0;

       public void init () {
              maxEntry = Database.getMaxEntries();
              memberArray = new Member [maxEntry];
              this.addEntry();
       }

       public void addEntry() {
              for (int i = 0; i < maxEntry; i++) {
                     Member newEntry;

                     if (!(Database.getCompany(i).equals("unknown"))) {
                            newEntry = new OwnCompany(Database.getSection(i));
                     }
                     else {
                            newEntry = new ForeignCompany(Database.getSection(i));
                     }

                     newEntry.setLastName(Database.getName(i));
                     newEntry.setSalary(Database.getSalary(i));
                     if (Database.getCityName(i).equals("home")) {
                            newEntry.setAdress(Database.getStreetName(i));
                     }
                     else {
                            newEntry.setAdress(Database.getStreetName(i),
                                                Database.getCityName(i),
                                                       Database.getZipCode(i));
                     }

                     this.memberArray[i] = newEntry;
              }
       }

       public String getRightName (int i) {
              String name = "";

              if (memberArray[i] instanceof OwnCompany) {
                     OwnCompany ofHelp = (OwnCompany) memberArray[i];
                     name = ofHelp.getSectionName();
              }
              else if (memberArray[i] instanceof ForeignCompany) {
```

```
                        ForeignCompany ffHelp = (ForeignCompany) memberArray[i];
                        name = ffHelp.getCompanyName();
                }
                return name;
        }

        public boolean isGreaterThan (Object obj1, Object obj2) {
                Member m1 = (Member) obj1;
                Member m2 = (Member) obj2;
                if (m1.salary < m2.salary) {
                        return true;
                }
                return false;
        }

        public void sortMemberArray () {
                SortArray sa = new SortArray(this, (Object []) memberArray);
        }

        public void displayMemberArray (Graphics g,int x, int startPos) {
                String sectionName = "";
                for (int i = 0; i < maxEntry; i++) {
                        sectionName = this.getRightName(i);
                        g.drawString(memberArray[i].getLastName() + " form "
                                + sectionName, x, startPos + 30*i);
                        g.drawString("live at "
                                + memberArray[i].getAdress(), x, startPos + 30*i+10);
                        g.drawString(" get " + memberArray[i].getSalary()
                                + " DM", x, startPos + 30*i + 20);
                }
        }

        public void paint (Graphics g) {
                g.drawString (Member.getNumber() + " Members"
                        + " <= Sort Members by Salary", 5, 10);
                this.sortMemberArray();
                this.displayMemberArray(g,5,30);
        }
}

abstract class Member {
        protected String lastName;
        protected int salary = 0;
        protected String adress;

        protected static int number;

        private final String cityName = "Cottbus";
        private final String zipCode = "03046";

        public Member () {
                Member.number++;
        }

        public static int getNumber() {
                return Member.number;
        }
        public void setLastName (String lastName) {
                this.lastName = lastName;
        }

        public void setSalary (int salary) {
                this.salary = salary;
        }

        public final void setAdress(String streetName, String cityName,
                String zipCode) {
                this.adress = streetName + " " + zipCode + " " + cityName;
        }
```

```
	public void setAdress(String streetName) {
		this.adress = streetName + " " + this.zipCode + " " + this.cityName;
	}

	public String getAdress() {
		return this.adress;
	}

	public String getLastName() {
		return this.lastName;
	}

	public int getSalary() {
		return this.salary;
	}
}

final class OwnCompany extends Member {
	private String sectionName;

	public OwnCompany (String sectionName) {
		super();
		this.sectionName = sectionName;
	}

	public String getSectionName () {
		return sectionName;
	}
}

final class ForeignCompany extends Member {
	private String CompanyName;

	public  ForeignCompany (String CompanyName) {
		super();
		this.CompanyName = CompanyName;
	}

	public String getCompanyName() {
		return CompanyName;
	}
}
```

SortArray.java

```
/*
Programmautor: Oliver Stecklina
Dateiname: SortArray.java
*/


package sort;

import sortInter.*;

public class SortArray {
	public SortArray (SortInterface si, Object objectArray[]) {
		for (int loop1 = 0;
			loop1 < objectArray.length;
			loop1++) {
			for (int loop2 = 0;
				loop2 < objectArray.length-loop1-1;
				loop2++) {
				if (si.isGreaterThan(objectArray[loop2],
					objectArray[loop2+1])) {
					Object help;
					help = objectArray[loop2];
					objectArray[loop2] = objectArray[loop2 + 1];
```

```
                                    objectArray[loop2+1] = help;
                              }
                        }
                  }
            }
}
```

SortInterface.java

```
/*
Programmautor: Oliver Stecklina
Dateiname: SortInterface.java
*/

package sortInter;

public interface SortInterface {
      public abstract boolean isGreaterThan(Object obj1, Object obj2);
}
```

Database.java

```
/*
Programmautor: Oliver Stecklina
Dateiname: Database.java
*/

package database;

public class Database {

      private static final int entries = 6;

      private static String[] names = {
            "Wolfgang", "Helmut", "Dieter", "Thomas", "Christian", "Werner"};
      private static int[] salary = {
            1300, 2100, 1600, 1600, 1800, 2500};
      private static String[] cityName = {
            "Dresden", "Berlin"};
      private static String[] streetName = {
            "Ernst Barlach Str. 4", "Salzgasse 5", "Clara Zetkin Str. 21",
                  "Berlinerstr. 14", "Mittelstr. 21", "Spremberger Str. 5"};
      private static String[] zipCode = {"01067", "12629"};
      private static String[] section = {
            "Reception", "unknown", "Sale", "Sale", "Purchase", "unknown"};
      private static String[] company = {
            "own", "GEA", "own", "own", "own", "Snemeis"};

      public static int getMaxEntries() {
            return Database.entries;
      }

      public static String getName(int number) {
            if (number < Database.entries) {
                  return Database.names[number];
            }
            return "error";
      }

      public static String getCityName(int number) {
            String result;
            switch (number) {
            case 1: {
                  result = Database.cityName[0];
                  break;
                        }
            case 5: {
                  result = Database.cityName[1];
                  break;
```

```
                        }
                default: {
                        result = "home";
                                }
                }
                return result;
        }

        public static int getSalary(int number) {
                if (number < Database.entries) {
                        return Database.salary[number];
                }
                return -1;
        }

        public static String getStreetName(int number) {
                if (number < Database.entries) {
                        return Database.streetName[number];
                }
                return "error";
        }

        public static String getZipCode(int number) {
                if (number == 1) {
                        return Database.zipCode[0];
                }
                else if (number == 5) {
                        return Database.zipCode[1];
                }
                return "home";
        }

        public static String getSection(int number) {
                if (number < Database.entries) {
                        return Database.section[number];
                }
                return "error";
        }


        public static String getCompany(int number) {
                if (number < Database.entries) {
                        return Database.company[number];
                }
                return "error";
        }
}
```