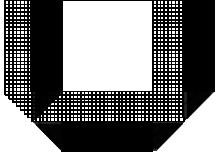


UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK
D-76128 Karlsruhe

Java
Seminarbeiträge

Michael Philippsen, Editor

Technical Report 24/96 – July 1996

Seminarbeitrag von **Ulrich Springer**

Keine neue Programmiersprache hat in den vergangenen 20 Jahren so viel positive Unruhe in die Computerwelt gebracht wie Java. Java verspricht sichere, robuste, portable und objektorientierte Programme, die sogar über das Netz in Java-tauglichen Browsern laufen können. Von Sun wird das komplette Entwicklungpaket, das sogenannte Java Developers Kit (JDK) kostenlos via Internet zur Verfügung gestellt. Schon jetzt haben viele namhafte Firmen Java lizensiert, so daß davon ausgegangen werden kann, daß Java die Entwicklungsplattform für Internet-Projekte in der Zukunft sein kann.

Vorhaben wie Datenbankanbindungen und Einbettungen in gängigen Betriebssysteme sind zwar noch Zukunftsmusik, zeigen aber, daß Java keine Eintagsfliege unter den objektorientierten Sprachen sein wird. Diese Vorhaben machen Java auch für Unternehmen mit kommerziellen Internet-Produkten interessant.

1.1 Java – Was ist das?

1.1.1 Einleitung

Wir leben in einer Zeit allgemeiner Internet-Euphorie. Natürlich braucht heute jeder, der einigermaßen „in“ sein will, zuhause seinen Internet-Anschluß mit „E-Mail“, „News“, „Web-Browser“ zum Surfen und natürlich „Telebanking“.

Diese Begriffe kommen mittlerweile schon den Menschen leicht über die Lippen, die vor geraumer Zeit noch nicht einmal richtig verstanden haben, wozu eine Festplatte im Computer eingebaut ist. Heute surfen sie lieber abends im Internet, als in die Diskothek zu gehen oder sich um die Familie zu kümmern.

Zu dieser Erscheinung gesellt sich seit neuestem noch eine weitere, die Java-Euphorie. Wenn man Sun (die Erfinder von Java) Glauben schenken darf, dann gibt es in naher Zukunft keine Softwarepakete mehr, die für die Textverarbeitung 30MB auf der Festplatte verbrauchen, zum Aufruf mindestens 3MB freien Arbeitsspeicher benötigen und dazu spätestens zu jedem Jahreswechsel oder Betriebssystem-Upgrade für mehrere hundert Mark „upgedated“ werden müssen.

Die Vision von Sun und anderen Firmen, wie zum Beispiel Oracle, ist die folgende: Dank Java braucht man kein großes Softwarepaket mehr kaufen, auch die Softwarepflege gehört dann ins Reich der Legenden. Wenn man einen Brief schreiben will, dann schaltet man einfach sein Internet-Terminal an, drücken den Knopf „Text erstellen“, das entsprechende Programm wird via Internet geladen und man schreibt los.

nicht), bietet Java noch andere Vorteile. Die objektorientierte Programmiersprache Java bietet beispielsweise Vorteile gegenüber den Standard-Programmiersprachen C und C++. Bislang mußte der Programmierer sich neben dem Entwickeln der Software noch mit der immer größer werdende Anzahl von inkompatiblen Hardwarearchitekturen, den unterschiedlichen inkompatiblen Betriebssystemen und den unterschiedlichsten plattformabhängigen Graphical-User-Interfaces herumschlagen. Zu diesen Stolpersteinen kommt jetzt noch hinzu, daß die Software in einer verteilten Client/Server-Umgebung laufen muß. Die Entwicklung des Internet und das World Wide Web haben neue, bislang unbekannte Komplexität in den Entwickungszyklus neuer Software gebracht. Es gibt zwar viele unterstützende Werkzeuge, aber diese wollen natürlich auch erst einmal gelernt werden.

So wie es momentan aussieht, haben anscheinend die modernen objektorientierten Techniken neue Probleme hinzugefügt, ohne die alten Probleme zu lösen [50].

1.1.2 Vorteile von Java

Java verspricht nun Besserung. Folgende Merkmale von Java sollen die oben angesprochenen Probleme lösen [16].

- Java ist objektorientiert und einfach zu erlernen.
- Der Entwicklungszyklus ist viel schneller, weil Java interpretiert wird. Es entfällt der Compilier-Link-Start-Teste-Absturz-Debug Zyklus.
- Java ist portabel. Die Applikationen, die mit Java entwickelt werden, laufen auf verschiedenen Betriebssystemen und Hardwarearchitekturen, ohne daß das Java-Programm modifiziert werden muß. Voraussetzung ist lediglich, daß der Java-Interpreter auf der Zielplattform existiert.
- Java-Programme sind robust, weil das Java-Laufzeitsystem die Speicherverwaltung übernimmt.
- Die interaktiven grafischen Java-Programme besitzen eine hohe Leistung, da durch das multithreading in Java konkurrierende Threads unterstützt werden.
- Die Java-Programme sind anpassungsfähig bei wechselnden Umgebungen, da Module dynamisch via Netzwerk von überallher ladbar sind.
- Java-Programme sind sicher. Das Java-Laufzeit-System besitzt eingebaute Schutzvorrichtungen, um ungewollte Systemzugriffe von fremden Code zu verbieten.

Allein diese Merkmale genügen schon, um Java an der Stelle von C++ in der Softwareentwicklung einzusetzen, speziell für verteilte Programme. Hinzu kommt, daß das Java Developers Kit (JDK) via Internet von Sun kostenlos zur Verfügung gestellt wird (*http://www.javasoft.com*). Bislang haben laut Sun mehrere tausend Interressierte das JDK via Internet kopiert und es existieren heute schon mehr als tausend Java-Programme, die von kleinen Spielen bis hin zu Finanz-Diensten reichen.

Der Beweis, daß Java in der Zukunft eine bedeutende Rolle spielen wird, ist die Tatsache, daß mehr als 25 führende Unternehmen der Computerbranche Java und JavaScript (siehe später) unterstützen werden. Darunter sind so namhafte Unternehmen wie IBM, Silicon Graphics, Toshiba, Oracle, Borland, Netscape, Adobe, Apple, Macromedia und Microsoft(!) [53].

In der Java-Welt existieren einige neue Begriffe, die bei einem Java-Neuling zu Verwechselungen führen können. Deshalb findet sich an dieser Stelle eine grundlegende Begriffserklärung für die Java-Welt [43].

- *Java* (im amerikanischen Sprachgebrauch: umgangssprachlich für Kaffee) ist eine objektorientierte Programmiersprache, die von Sun Microsystems entwickelt wurde. Sie enthält viele Ähnlichkeiten zu C++, ist aber unter anderem aus Sicherheitsgründen um einige Möglichkeiten ärmer. Eine genauere Beschreibung der Programmiersprache Java ist in Abschnitt 4 zu finden.
- *Java Applets* sind Java-Programme, die in HTML-Seiten integriert sind und nur in einem geeigneten, Java-tauglichen Browser ablaufen können.
- *JavaScript* ist eine Ergänzung von Java (ähnlich wie C und die C-Shell unter UNIX). Mit JavaScript können Java-Applets in HTML-Seiten eingebunden werden.
- *Java Applications* sind Java-Programme, die unabhängig von HTML-Seiten ablaufen können.
- *HotJava* ist ein in Java geschriebener WWW-Browser von Sun. Mit HotJava können HTML-Seiten angezeigt werden und eingebundene Applets werden automatisch geladen und gestartet (Netscape 2.0 und Spyglass Mosaic können das auch).
- *JDK* ist das Java Developers Kit und ist die Entwicklungsumgebung von Sun für Java Applets und Java Applications.
- *Bytecode* ist eine Liste von Instruktionen, die aussehen wie Maschinencode, aber unabhängig vom ausführenden Prozessor sind. Java Programme (Applets und Applications) werden zunächst in Bytecode übersetzt bevor sie dann mit einem Interpreter gestartet werden.

1.1.4 Zusammenfassung

Da Java einfach, robust und sicher ist, ist es prädestiniert für den Einsatz zur Entwicklung von verteilten Internetprogrammen. Java wächst mit der wachsenden Internetgemeinde mit, da das Java Developers Kit (JDK) kostenlos via Internet von Sun zu beziehen ist. Viele Computerunternehmen, darunter auch „Goliath“ Microsoft, unterstützen Java und somit ist Java schon nach kurzer Zeit zu einer echten Alternative zu herkömmlichen Entwicklungsumgebungen geworden.

1.2 Java – Vergangenheit

Zunächst soll in diesem Abschnitt die Vergangenheit von Java beleuchtet werden. Wie war es möglich, daß Sun in relativ kurzer Zeit ein solches Produkt entwickeln konnte, ohne daß die Konkurrenz ein adäquates Produkt dagegenstellen konnte?

1.2.1 Historie

Im folgenden soll die Historie der Entwicklung von Java dargestellt werden. Die folgenden Angaben sind aus den WWW-Artikeln „The Java Saga“ [4] und „Java: The Inside Story“ [19] entnommen.

wurde damals noch hauptsächlich von FTP- und E-Mail-Diensten genutzt. Vereinzelt waren auch Gopher-Server (ein Vorgänger des heutigen WWW) schon in Betrieb. Welche ungenutzten Möglichkeiten das Internet noch zu bieten hatte, wurde damals von niemanden geahnt.

Der PC-Markt hatte schon Hochkonjunktur und war schon derartig von Microsoft bestimmt, daß man Microsoft als Gewinner und Bestimmer des Massen-PC-Marktes bezeichnen konnte. Sun und auch andere Unternehmen der Computerbranche verpennten total die Entwicklung. Aber ähnlich wie bei IBM machte Sun der Verlust des PC-Marktes überhaupt nicht zu schaffen, da das Workstation- und Servergeschäft auf Hochtouren liefen.

Eine bedeutende Rolle in der Java-Entwicklung spielten ein gewisser McNealy, ein Vertreter der oberen Führungsschicht von Sun, und Patrick Naughton, ein Programmierer bei Sun. Naughton spielte damals in McNealys Eishockey-Team und er erzählte eines Abends McNealy, daß er nach nur drei Jahren Sun verlassen wollte und zu NeXT gehen wollte (Naughton: „They [NeXT] are doing it right."). Das wollte McNealy so nicht akzeptieren und bat Naughton alles aufzuschreiben, was ihm bei Sun in der jetzigen Situation nicht gefällt (McNealy: „Tell me what you would do if you were God.").

Naughton bemängelte folgende Punkte: Sun sollte einen Künstler engagieren, der die altbackenen Interfaces aufpoliert. Desweiteren sollte man sich auf ein Programming Toolkit beschränken und nicht auf mehrere unterschiedliche. Er führte noch ein paar weitere Zustände auf, die ihm nicht gefielen. Erstaunlicherweise kündigte Naughton nicht sofort sondern wartete eine Reaktion ab.

McNealy verschickte per E-Mail die Kritikpunkte von Naughton an verschiede Mitarbeiter von Sun. Eine überwältigende Mehrheit stimmte den Kritiken von Naughton zu und diese Tatsache stimmte McNealy nachdenklich.

Januar 1991. Naughton wurde zu einem Brainstorming-Meeting von führenden Entwicklern bei Sun eingeladen, dessen Ergebnisse zu einem neuen Projekt führen sollte. Das Team, das das neue Produkt entwickeln sollte, sollte relativ klein sein (klein genug, um sich beim Chinesen an einen Tisch setzen zu können). Die Zielgruppe des neuen Produktes sollten Konsumenten sein, also quasi Jedermann. Und das Produkt sollte auf jeden fall realisiert werden, auch falls es nicht in die eigentliche Produktlinie von Sun passen sollte. Schließlich sollte auch eine neue Generation von Maschinen eingeschlossen werden, die „normale" Menschen einfach bedienen können.

Februar 1991. Das Projekt wurde schließlich vom Top-Management von Sun genehmigt. Das Projektjektteam von „Green", wie das Projekt genannt wurde, nahm seine Arbeit außerhalb von Sun auf, um eventuellen Innovationsgegnern bewußt aus dem Weg zu gehen. Das Projekt wurde geheim gehalten, nur dem Top-Management wurde regelmäßig über den Projektstand berichtet. Die Entwicklung der Soft- oder Hardware muß nicht unbedingt kompatibel zu der gegenwärtigen Produktlinie von Sun sein. Das Budget des ersten Jahres für „Green" betrug 1 Mio. Dollar.

Wohlgemerkt war die eigentliche Richtung des Projektes noch vollkommen offen. Das Team hatte sehr großen Handlungsspielraum. Nur eines war klar, Microsoft hatte den PC-Betriebssystem- und Anwendungssoftware-Markt im Griff. Das hatte für „Green" zur Folge, daß die „Green"-Software praktisch auf jeder denkbaren Hardware laufen sollte. Desweiteren sollte die Soft- und Hardware kompakt und einfach sein (genau das Gegenteil von dem, was Sun verkaufte).

Das Team machte folgende Beobachtungen. Computerchips werden heutzutage in Toaster, Videorecorder und andere Haushaltsgeräte eingebaut. Fernbedienungen existieren für Viedeorecorder, Fernseher und HIFI-Anlagen, die sämtlich programmierbar sind. Aber die Mehrzahl

schlecht beschrieben ist. Warum sollte man die gewünschte Funktionalität nicht in ein Gerät integrieren und es zusätzlich noch leicht bedienbar machen? Die Vorstellung war ein kleines, handliches Gerät mit LCD-Bildschirm und einem Touchscreen, das sämtliche in einem Haushalt befindlichen Geräte steuern kann. Auf diesem Display könnte ein kleiner Film ablaufen, wie alles zu bedienen sei.

Schon zu diesem frühen Zeitpunkt kann man feststellen, daß „Green“ die Richtung Multimedia einschlug. Man entschied sich, einen Prototypen zu bauen.

April 1991. Es wurde nun beschlossen, das Team weit weg von Sun zu plazieren. Nach dem Umzug hatte man keine LAN-Anbindung an Sun mehr. Das Team war der Überzeugung: „We thought if we stayed over there, we would end up with just another workstation.“

Ein erste Entscheidung wurde bezüglich der Entwicklungsumgebung getroffen. C++ war zwar der defacto Standard und schnell aber in bezug auf Robustheit und Sicherheit zu unzuverlässig. Und im Bereich der Konsumerelektronik ist Zuverlässigkeit wichtiger als Schnelligkeit. Folglich mußte eine neue Entwicklungssprache her, die der Einfachheit halber auf C++ basierte.

Mai-August 1991. Auf zwei Gebieten wurde bei „Green“ hauptsächlich gearbeitet. Zum einen wurde die neue Interpreter-Sprache entwickelt, die zunächst Oak und später dann Java genannt wurde, da der Name Oak schon als Warenzeichen vergeben war. Zum anderen wurde an einer grafischen Benuzungsschnittstelle gearbeitet, auf der kleine Filmchen ablaufen mußten, eine virtuelle Haus-Welt darstellen und auch Interaktion mit dem Benutzer garantieren sollte.

Ab August 1991. Die Entwicklung eines Prototypen wurde auch vorangetrieben. Ein Name war zumindest schon gefunden, er lautete Star7. Nun ging es noch darum, die geeignete Hardware zu finden. Man testete verschiedene Gebrauchsgegenstände und fand schließlich die geeigneten Teile. Man entschied sich für die Aktivmatrix-Farb-LCD Technik aus einem Minifernseher von Sharp und setzte einen Touch-Screen darauf. Zudem baute man ein Stereo-Lautsprechersystem ein, das sie von einem Nintendo-Gameboy abgeschaut hatten.

August 1992. Die Demonstration von Star7 vor dem versammelten Top-Management von Sun wurde zu einem vollen Erfolg. Der Prototyp war aber nicht das einzige Ergebnis des „Green“-Projekts. Ein Abfallprodukt war schließlich Oak/Java, eine objektorientierte Programmiersprache, die gerade in verteilten Systemen eingesetzt werden kann und bei der auf Sicherheit sehr großen Wert gelegt wurde.

Zum ersten Mal wurden auch Gespräche mit anderen Elektronik-Konzernen geführt. Mitsubishi Electronic und France Telecom interessierten sich für Star7 und wollten es vor allem in Fernsehern und Telefonen einsetzen.

Dezember 1992. In Palo Alto wurde FirstPerson als eine Tochtergesellschaft von Sun mit insgesamt 60 Mitarbeitern gegründet. Aber kurz darauf ließen Mitsubishi und France Telecom den Auftrag platzen, weil sie der Meinung waren, daß 50 Dollar für Star7 zu teuer sei. Sie kaufen lieber einfache Chips für wenige Dollar und beschlossen,m die Bedienung ihrer Geräte durch verändertes Design zu verbessern.

warten. Der Medienkonzern Time Warner startete ein Interactive-TV-Projekt. Star7 sollte als Set-Top-Box im interaktive TV eingesetzt werden. Eine Set-Top-Box ist die Verbindung zwischen dem Fernsehgerät und dem Information-Highway. Ein solches Gerät soll die riesige Datenflut von Bildern, Daten und Filme koordinieren und die Abrechnung sich über ein Netzwerk realisieren.

Wiederum mußte eine Niederlage verkraftet werden, da Time Warner den Auftrag für eine Set-Top-Box Silicon Graphics Inc. (SGI) vergab. Pikanterweise ist SGI der größte Rivale von Sun im lukrativen Workstation-Bereich. Allerdings endete für Time Warner und SGI das Projekt in einem finanziellen Desaster.

Juni 1993. NCSA Mosaic 1.0 kam als erster grafischer Internet-Browser auf den Markt und verhalf dem WWW zu seinem Siegeszug. Interessanterweise war es nicht Sun, die als erstes einen Internet-Browser entwickelte, obwohl mehr als 50% der Internet-Server damals Sun-Maschinen waren.

Ende 1993. Nach weiteren Pleiten mit einer CD-ROM-Game-Machine und einer nie realisierten Konferenz-Software (TED5) wurde das Top-Management von Sun aktiv. Es wollte nicht noch weiter Geld in einen Bereich pumpen, der keinen Profit versprach (Management:„Find something to produce profit. NOW!!!“).

April 1994. FirstPerson wurde zugunsten eines neuen Sun-Ablegers namens SunInteractive aufgelöst. SunInteractive sollte nun ein neues Projekt angehen, dessen Ziel es war, Oak/Java endgültig als eine Internet-Sprache zu realisieren.

Juli 1994. James Gosling – heute Vizepräsident von Sun – arbeitete verstärkt an Oak. Er gab Oak den vermarktbaren Namen Java und realisierte die Sprache, wie sie heute existiert.

Patrick Naughton realisierte angeblich in einem Wochenend-Hack die nächste Internet-Killer-Applikation: den Internet-Browser HotJava.

29. September 1994. Java/HotJava wurde zum ersten Mal dem Top-Management vorgeführt, das zunächst geteilter Meinung über einen etwaigen Erfolg von Java/HotJava war.

Dezember 1994. Java/HotJava wurde ausgewählten Personen, die zum größten Teil nicht bei Sun beschäftigt waren, zur Probe gegeben. Zu den Testpersonen gehörte auch Marc Andreessen von Netscape, das durch ihren gleichlautenden Internet-Browser binnen kürzester Zeit zum Marktführer dieser Technologie aufstieg. Eben dieser Andreessen lobte Java/HotJava in den höchsten Tönen („What these guys are doing is undeniably, absolutely new. It's great stuff.“). Dieses Lob von sogenannten „Godfather of the Internet“ gab den Ausschlag, daß das Java/HotJava-Projekt bis zur Vollendung vorangetrieben wurde.

23. Mai 1995. Die erste offizielle Präsentation von Java/HotJava auf der Sun-Hausmesse „SunWorld '95“ wurde ein großer Erfolg. Viele Firmen wurden auf dieses Produkt aufmerksam, das schon fast revolutionären Charakter besaß.

Apple, AT&T, Borland, Digital Equipment, Hewlett-Packard, Informix, Macromedia, Novell, Oracle, Sillicon Graphics, Sybase und Netscape, unterstützen Java bzw. JavaScript in ihren Produkten. Insgesamt waren es 28 Firmen. Diese Zusammenarbeit mit den Firmen war deshalb so wichtig, da Microsoft ein ähnliches Produkt („Blackbird“) im Januar 1996 auf den Markt bringen wollte, das allerdings nur auf Windows95/NT zugeschnitten war. Sun machte auch Microsoft das Angebot, Java bzw. JavaScript zu unterstützen, aber Bill Gates lehnte zu diesem Zeitpunkt noch ab. Zwei Monate später lizensierte auch Microsoft Java.

Die Lizensierungsmodalitäten waren recht moderat. Sun verlangte eine einmalige Lizensierungsgebühr von 125 000 Dollar und 2 Dollar je verkaufter Kopie. Laut Sun sah man diese moderate Lizensierung als eine strategische Investition in die Zukunft an.

Januar 1996. Ein neuer Geschäftsbereich von Sun namens JavaSoft wurde gegründet. Zudem konnte ab sofort die erste offizielle Version von Java 1.0 via Internet kostenlos kopiert werden. Das sogenannte Java Developers Kit (JDK) war zunächst für die Betriebssysteme Windows95, WindowsNT und Solaris auf SPARC verfügbar. Portierungen auf MacintoshOS 7.5, Linux und anderen Plattformen waren schon in Vorbereitung.

1.2.2 Zusammenfassung

Eine etwas ungewöhnliche Entwicklung führte zum Erfolg von Java/HotJava. Dazu trug noch bei, daß viele Firmen – auch Microsoft – bereit waren, Java zu unterstützen. Schließlich ermöglichte Sun vielen „hungrigen“ Programmieren, Java zu benutzen, indem das JDK kostenlos via Internet zur Verfügung gestellt wurde. Genau dieser Punkt verhalf Java in relativ kurzer Zeit zu einer weiten Verbreitung.

1.3 Java – Gegenwart

1.3.1 Anforderungen

Da Java noch eine sehr junge Technologie ist, kann man momentan noch nicht voraussehen, wie weit sie sich entwickeln wird. Prinzipiell läßt sich sagen, daß überall da, wo der Java-Interpreter läuft, auch Java-Bytecode ausgeführt werden kann. Die Entwickler von Sun behaupten, daß der Bytecode darauf ausgelegt ist, auf jedem System einfach interpretierbar und übersetzbar zu sein. Momentan gilt dies für die folgenden Plattformen (Stand Frühjahr 1996):

- SunOS 4 und 5
- Solaris x86
- SGI
- DEC OSF/1
- HP/UX
- Windows95 und NT
- Macintosh System 7.5
- Linux

Zunächst sollen einmal die herausragenden Merkmale von Java angesprochen werden [16]. Java ist ...

- *architekturneutral.* Überall, wo der Java-Interpreter existiert (s.o.), kann der Bytecode ausgeführt werden. Die Java-Programme laufen ohne jegliche Änderungen am eigentlichen Programmcode.

- *robust.* In Java existiert keine Pointer-Arithmetik. Das wird für denjenigen ungewöhnlich sein, der es gewohnt ist, in C oder C++ zu programmieren. Durch die fehlende Pointer-Arithmetik werden tatsächlich viele C-Hacks unmöglich gemacht, aber dafür ist Java weniger fehleranfällig. Jeder Datentyp in Java ist eigentlich ein Objekt, und ein Objekt kann nicht als ein Feld von Bytes betrachtet werden, in dem man beliebig Werte ändern kann. Ein wenig pedantisch wird die Sprache jedoch, wenn man mit `if (objekt)` eigentlich `if (objekt != null)` meint, und letzteres tippen muß, weil *objekt* nunmal nicht von Typ *boolean* ist.

- *objektorientiert.* Die Entwickler von Java haben bewußt die Sprache C++ als Basis für Java verwendet, um den vielen Anwendern der Standardsprache C++ den Umstieg auf Java zu erleichtern. Im Gegensatz zu C++ ist Java aber streng objektorientiert konzipiert. Was das im einzelnen bedeutet wird in Abschnitt 4.3 noch genauer beleuchtet.

- *multithreaded.* Java ist in der Lage, mehrere Operationen quasi-parallel auszuführen. Dazu werden zur Kontrolle besondere Schlüssellwörter, wie zum Beispiel `synchronized`, zur Verfügung gestellt. Das macht sich in der Praxis deutlich bemerkbar. Wenn man zum Beispiel für lange dauernde Berechnungen oder Animationen nicht einen eigenen Thread startet, blockiert man schnell den Browser. Böswillige Programmierer können so auf einfache Art und Weise den Browser eines Benutzers lahmlegen. Etwaige Schutzmechanismen dagegen sind aber vorgesehen.

- *schnell.* Der Java-Compiler erzeugt binären Bytecode. Der Java-Interpreter verliert also keine Zeit mit lexikalischer Analyse und Parsing. Er kann den geladenen Code sofort ausführen. Der Bytecode kann auch in maschinenabhängigen Code übersetzt werden. Laut Sun soll dieser Code in Punkto Ausführungsgeschwindigkeit genauso schnell sein wie ein übersetztes C++-Programm.

- *verteilt.* Objekte können von irgendwoher aus dem Netz geladen werden. Natürlich sind auch Schutzmechanismen vorgesehen, damit man sich nicht ein „Trojanisches-Pferd“ auf seinen Rechner einschleppt. So wird zum Beispiel beim Laden eine Objektes prinzipiell zunächst auf dem lokalen Rechner nachgeschaut, ob das Objekt vorhanden ist, und nur nach einem Mißerfolg wird erst bei der angegeben Quelle im Internet gesucht.

- *erweiterbar.* Der Java-Compiler kennt ein Schlüsselwort `native`, das Methoden bezeichnet, die in einer maschinenabhängigen Sprache – etwa C – implementiert sind. Diese Eigenschaft kann allerdings nur in Java-Programmen und nicht in Java-Applets ausgenutzt werden.

Schließlich besitzt der Java-Interpreter noch einen Garbage-Collecting Mechanismus. Bislang mußte jeder C- oder C++-Programmierer bei der expliziten Speicherbelegung durch `new` immer dafür sorgen, daß der so bereitgestellte Speicher auch wieder freigegeben wird. Der Java-Interpreter besitzt dagegen einen automatischen Garbage-Collecting Mechanismus, so daß sich

management in Java basiert auf einem Modell von Objekten und Referenzen auf Objekte. Alle Referenzen auf einen Speicherbereich (=Objekt) werden durch symbolische Handles gespeichert. Wenn ein Objekt keine Referenz auf sich mehr besitzt, dann ist es ein erster Kandidat für das Garbage-Collecting.

1.3.3 Unterschiede zu C++

Wie schon wiederholt erwähnt wurde, war im Entwicklungszeitraum von Java die Programmiersprache C++ der Standard bei universellen Programmiersprachen. Die Entwickler wollten es einerseits Umsteigern von C++ einfach machen aber auch andererseits die Redundanzen vermeiden, die in C++ existieren. Das Resultat war eine neue Programmiersprache, die sehr viele Ähnlichkeiten mit C++ hat aber auch einige markante Unterschiede aufweist. Man könnte Java eigentlich auch als „C++−−++“ bezeichnen. C++ war die Basis der Entwicklung von Java, „−−“ bezeichnet die Vermeidung von Redundanzen, die in C++ existieren und das letzte „++“ bezeichnet die zusätzlichen Merkmale, die in Java hinzugefügt wurden. Im folgenden sollen die Unterschiede zu C++ aufgeführt und genauer betrachtet werden [16].

- Kein `typedef`, `#define` und kein Präprozessor mehr

  Ein großes Problem beim Verstehen von C- und C++-Programmen ist, daß man manchmal eine Unmenge von sogenannten „Header Files “ lesen und alle notwendigen `typedef`- und `#define`-Anweisungen lokalisieren muß, bevor man an das eigentliche Analysieren des Programmes gehen kann.

  Java-Programme sind dagegen einfach zu analysieren und zu verstehen. Es gibt kein `typedef`, kein `#define` und keinen Präprozessor mehr in Java. Folglich sind auch die Header Files überflüssig geworden. Alle Definitionen von Klassen stehen direkt in den Programmdateien. Anstatt der `#define`-Anweisung verwendet man in Java Konstanten und anstatt `typedef` wird einfach eine neue Klasse definiert, die dann den neuen Typ repräsentiert.

  Diese Einschränkung führt dazu, daß die Programmierer Java-Programme einfacher verstehen können und wichtiger noch, sie können Java-Programme einfacher und schneller wiederverwenden.

  Es sei noch angemerkt, daß Konstanten und auch Klassen natürlich auch in C++ vorhanden sind. Aber in C++ wird der Programmierer nicht dazu gezwungen, bei einem neuen Typ eine neue Klasse zu definieren, wie es beim objektorientierten Entwurf eigentlich vorgesehen wird. Aufgrund der Redundanz in C++, die wegen der Einbettung von C in C++ existiert, ist der Entwickler jedoch nicht gezwungen, streng objektorientiert zu denken. In Java dagegen muß ein neuer Typ durch eine Definition einer neuen Klasse realisiert werden.

- Kein `struct` oder `union` mehr

  Ebenfalls gibt es keine Notwendigkeit mehr, `struct` oder `union` zu verwenden, wenn Klassen vorhanden sind. Den Effekt, den `struct`- oder `union`-Konstrukte erzielen, kann durch eine Deklaration einer Klasse mit den entsprechenden Variablen als Attribute der Klasse erreicht werden. Hierdurch wird wiederum Redundanz wie in C++ vermieden.

- Keine Funktionen mehr

tung von Methoden in Klassen. Eine Funktion modifiziert Daten einer bestimmten Datenstruktur. Da die Datenstrukturen beim objektorientierten Programmieren durch Klassen repräsentiert werden und die Methoden dieser Klassen diese Daten modifizieren, gibt es keine Notwendigkeit für Funktionen und Prozeduren.

- Keine Mehrfachvererbung mehr

  Wenn eine Klasse mehr als eine Oberklasse besitzt, nennt man das Mehrfachvererbung. Was geschieht aber nun, wenn in mehr als einer Oberklasse Methoden mit demselben Namen und denselben Parametern existieren, die aber unterschiedlich implementiert sind? In Java wird diese Frage gar nicht erst aktuell, da eine Mehrfachvererbung ausgeschlossen wird.

- Kein Überladen von Operatoren mehr

  In Java ist es nicht möglich, in der Sprache existierende arithmetische Operatoren zu überladen, d.h. neu zu implementieren. Das ist allerdings in meinen Augen bei C++ ein sehr nützlicher Mechanismus gewesen, Programme besser interpretierbar zu gestalten. Warum soll man nicht den +-Operator bei einer Addition zweier Objekte der Klasse Vektor benutzen? Statt dessen muß man nun in der Klasse eine Methode zur Verfügung stellen, die zwei Vektoren addiert. Was ist einfacher zu verstehen: `vektor1 = vektor2 + vektor3;` oder `vektor1 = vektor2.Add(vektor3);`?

- Keine automatische Typumwandlung mehr

  Um Programmierer auf ungenaues Programmieren in Java aufmerksam zu machen, wurde die in C++ übliche automatische Typumwandlung von Ausdrücken aufgegeben. Wenn zum Beispiel einem `int` ein `double` zugewiesen würde, würde es in Java zwangsweise zu einem Fehler führen, da es ja unter Umständen zu einem ungültigen Wert im `int`-Ausdruck führen könnte. Manche C/C++-Compiler geben an solchen Stellen meistens nur eine Warnung aus, die dann meistens von den Programmierern nicht Ernst genommen werden. Dabei handelt sich es aber um eine mögliche Fehlerquelle.

- Keine Zeiger-Arithmetik mehr

  Aufgrund unterschiedlicher Beobachtungen hat man herausgefunden, daß Zeiger-Konstrukte diejenigen Konstrukte sind, die den Programmierer verleiten, unbeabsichtigt Fehler in seinem Programm einzubauen. In Java wurde deshalb auf die Zeiger-Arithmetik verzichtet. Statt dessen werden, wo in C/C++-Programmen Zeiger verwendet würden, einfach Objekte und Arrays von Objekten verwendet. Wohlgemerkt sind Arrays auch Objekte, die z.B. eine Methode zur Abfrage der Größe des Arrays besitzen. Das Java-Laufzeitsystem überprüft den Zugriff auf die Arrays und verbietet einen ungültigen Zugriff außerhalb der definierten Arraygrenzen.

  Somit ist die Fehlerquelle von „dangling pointers“ (Zeiger, der auf einen mittlerweile ungültigen Speicherbereich zeigt, der unsinnige Werte besitzt, da der Bereich schon von anderen dynamischen Strukturen belegt wird), die aus der C/C++-Welt bekannt sind, aus der Welt geschafft.

In der nachfolgenden Tabelle werden nochmals die gerade erwähnten Unterschiede von Java und C++ kurz und bündig dargestellt.

Merkmal	Java	C++
Zeiger	–	`int *a,b; a = &b;`
Zeichen	`char a = ü;` a wird als 16-Bit-Unicode gespeichert	`char a = ü;` a wird als ein Byte gespeichert
Zeichenketten	*String*-Objekte für konstan-te, *StringBuffer*-Objekte für variable Zeichenketten	`char Name[30];` Feld von 8-Bit-Charaktern
numerische Datentypen	nur vorzeichenbehafte Datentypen	vorzeichenbehafte und vorzeichenlo-se Datentypen
Boolsche Werte	*boolean* ist ein Basis-Datentyp	(-)
Felder (Arrays)	Felder sind Objekte. Zur Laufzeit wird eine Überprüfung auf korrek-ten Zugriff innerhalb der Dimension des Feldes durchgeführt.	`int a[5]; a[10]=4711;` Felder werden statisch alloziert und ver-hindern keine Zugriffe außerhalb der definierten Grenzen des Feldes.
Aufzählungstyp	–	`enum {...};`
typedef	–	`typedef {...};`
zusammengesetzte Datentypen und Varianten	–	`struct a {...}; union b {...}`
Präprozessor	–	√
#include	`import <interface>` stellt Schnitt-stellen aus anderen Objektmodulen zur Verfügung	√
NULL	ist Schlüsselwort	`#define NULL 0`. NULL wird durch Präprozessor im Quelltext ersetzt.
Kommentare	// ... und /* ... */. /** ... */ zur au-tomatischen Dokumentationsgewin-nung von Schnittstellen	// und /* ... */
Mehrfachverer-bung	`class a extends b implements c {...}`	`class a: public b, public c {...}`
Templates	–	`template <class a>`
Überladen von Operatoren	–	√
Linker	nicht benötigt, da interpretiert	essentiell vorhanden
Thread-Unterstützung	`synchronized` {...} klammert kri-tischen Abschnitt	–
Heap-Verwaltung	Garbage Collection durch Prozeß im Hintergrund	Explizite Freigabe von Speicher-blöcken ist durch Programmierer erforderlich.

1.3.4 Hauptbestandteile der Sprache Java

Java folgt im wesentlichen der Syntax von C++ (genaueres in [20]). Diese Tatsache macht es Umsteigern von C++ leicht, Java zu verwenden. Hier sollen nur die wesentlichen Sprachelemente kurz vorgestellt und die Unterschiede zu C/C++ aufgezeigt werden.

- Numerische Datentypen

  Integer: byte 8-bit, short 16-bit, int 32-bit, long 64-bit

  In Java existieren keine vorzeichenlose Ganzzahldatentypen wie in C++ (unsigned).

  Real: float 32-bit, double 64-bit

  Diese Datentypen und ihre arithmetischen Operatoren sind durch die IEEE 754 Spezifikation definiert. Ein literaler Ausdruck, wie z. B. 4711.0815 wird immer als double-Wert behandelt. Der Anwender muß diesen Ausdruck explizit zu einem float umwandeln, wenn er diesen einer float-Variablen zuweisen will.

- Alphanumerische Datentypen

  Der einzigen alphanumerischen Datentyp ist char als 16-bit Unicode. Dieser wurde wegen den Ansprüchen der Internationalisierung als 16-bit-Typ realisiert.

- Boolsche Datentypen

  In Java existiert im Gegensatz zu C/C++ ein boolscher Datentyp. Der Typ boolean kann lediglich die Werte true oder false annehmen und kann nicht zu einem numerischen Typ gewandelt werden, was in C/C++ möglich ist.

- Arithmetische und relationale Operatoren

  Es existieren die gleichen Operatoren wie in C/C++. Da aber in Java der unsigned-Datentyp fehlt, existiert als zusätzlicher Operator „>>>“. Dieser bezeichnet einen vorzeichenlosen right shift. Java benutzt zusätzlich den +-Operator zum Verknüpfen zweier Strings.

- Arrays

  Im Gegensatz zu C und C++ ist ein Array in Java ein in der Sprache verankertes Objekt. Die Handhabung von Arrays ist einfach. Ein Array muß an einer Stelle im Programm deklariert werden, ohne daß zunächst die Länge des Arrays spezifiziert werden muß. Mit dem Ausdruck `Point myPoints[];` wird die Variable `myPoints` als Array vom Typ `Point` deklariert. Wohlgemerkt ist nach der Deklaration der Speicherplatz des Arrays noch nicht reserviert. Deshalb muß noch vor dem Initialisieren des Arrays der Speicher reserviert werden. Das geschieht durch den Ausdruck `myPoints = new Point[10];`. Schließlich kann das Array nun noch mit Null-Elementen vom Typ Point initialisiert werden. Das geschieht in folgender for-Schleife.

  ```
  for (int i=0; i<10; i++)
      myPoints[i] = new Point();
  ```

  Die Klasse Array besitzt *length* als einzige Methode, mit der man die Länge eines Arrays ermitteln kann (`howmany = myPoints.length();`).

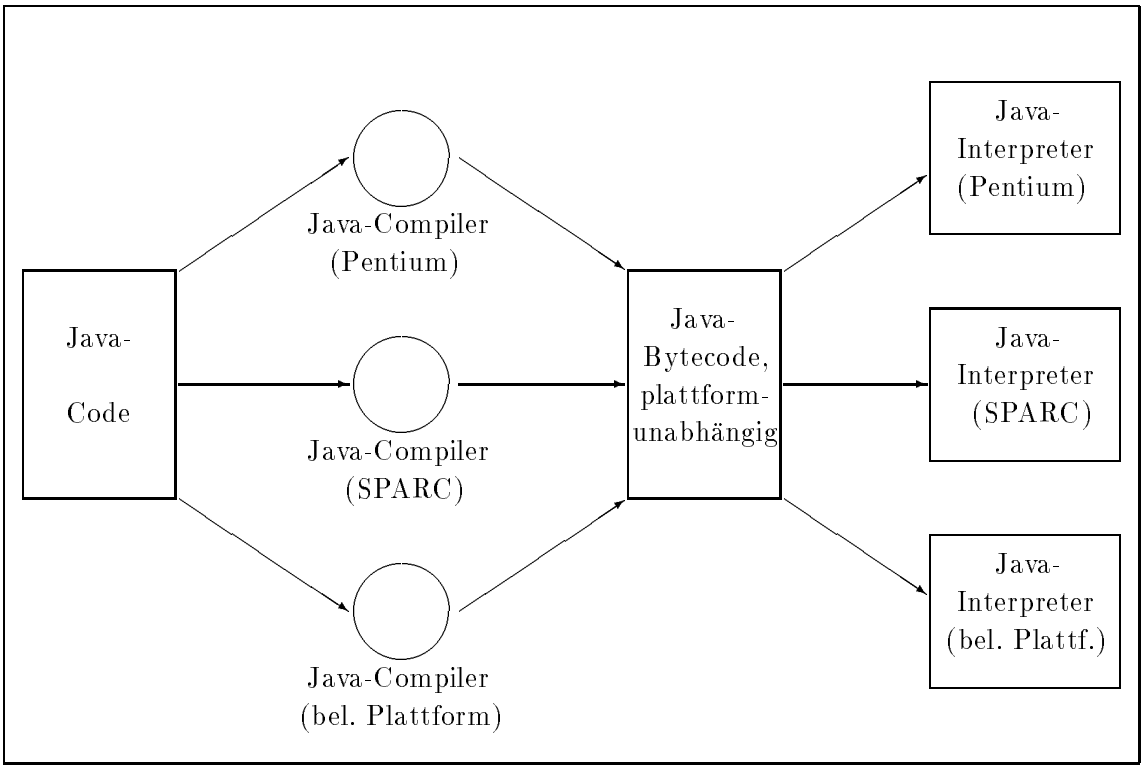
  Bei jedem Arrayzugriff wird in Java zur Laufzeit überprüft, ob der Zugriff auch gültig ist, d.h. ob er innerhalb der spezifizierten Arrayintervallgrenze liegt. Wie in C/C++ können Arrays der Länge *N* von *0* bis *N-1* indiziert werden. Jeder ungültige Zugriff wird erkannt und führt zu einer *exception*. Das verhindert die in C/C++ so beliebten „delayed-crashs“. Das sind Programmabstürze, die daraus resultieren, daß beim Schreiben jenseits der Arraygrenze eventuell andere gültige Speicherbereiche verfälscht werden. Auf diese wird dann zu einem späteren Zeitpunkt zugegriffen, wobei die Orginalinhalte durch die Arrayüberschreitung nicht mehr vorhanden sind und das Programm beispielsweise wegen eines ungültigen Speicherzugriffs abbricht.

Die in C/C++ bekannte pseudo-Array-Representation der Strings ist in Java nun vorbei. In Java sind Strings echte Objekte. Es gibt sogar zwei Klassen von Strings: die Klasse `String` ist für unveränderbare Strings gedacht. Diese werden einmalig initialisiert und sind danach unveränderbar. Die Klasse `StringBuffer` realisiert dagegen variable Strings, dessen Inhalt sich während der Lebensdauer des Strings beliebig ändern darf.

Wie bei Arrays existiert eine Methode *length*, die als Ergebnis die Länge des Strings liefert. Wie schon erwähnt, konkateniert der +-Operator zwei Strings zu einem. Die Initialisierung eines Strings geschieht durch Einschließen der gewünschten Zeichenkette in doppelten Anführungszeichen (`String hello = "Hello World!";`).

1.3.5 Programmieren in Java

Nach vielen einführenden Features zur Sprache Java wollen wir uns nun der Programmierung in Java widmen. Die folgende Abbildung verdeutlicht, wie der Java-Programmcode in ein ausführbares Format gebracht wird. Prinzipiell haben die Java-Quelldateien die Endung *.java*. Der Java-Compiler übersetzt das Programm in den sogenannten Bytecode. Die Bytecode-Datei besitzt die Endung *.class*. Nun läßt sich das Programm mit Hilfe des Java-Interpreters (auch Virtuelle Maschine genannt) ausführen. Im Falle eines Java-Applets kann ein geeigneter Browser (HotJava, Netscape 2.0, o.ä.) das Applet ausführen.



Wo genau unterscheiden sich nun Java-Applications und Java-Applets? Das soll nun an folgenden zwei kleinen Beispielen verdeutlicht werden.

```
class hello
{
      public static void main(String argv[])
      {
            System.out.println("Hello World!");
      }
}
```

Listing 2:

```
class hello
import java.awt.Graphics;
public class HelloApplet extends java.applet.Applet
{
       public void paint(Graphics g)
       {
             g.drawString("Hello World!", 5, 25);
       }
}
```

Natürlich darf als einführendes Programmierbeispiel das übliche Hello-World-Programm nicht fehlen. Hier ist es jeweils einmal als Java-Application (Listing 1) und als Java-Applet (Listing 2) realisiert.

Listing 1 beinhaltet nur eine einzige Klasse *hello*, die die einzige statische Methode *main* besitzt. Der Compiler *javac* übersetzt nun wie besprochen das Programm in Bytecode, der dann mit Hilfe des Java-Interpreters *java* ausgeführt werden kann. Der Interpreter stellt fest, daß die Klasse eine Methode enthält, die *public* und *static* ist und beginnt, die Instruktionen von *main* abzuarbeiten. Jede Klasse, die eine derartige *main*-Methode enthält, kann Startpunkt eines Java-Programmes sein.

Andererseits findet man in einem entsprechenden Java-Applet-Code (Listing 2) keine *main*-Methode wie eben besprochen. Anstatt der *main*-Methode findet man beim Applet eine Methode *paint*, die eine Methode der Klasse *Component* ist, welche wiederum über die Klasse *Applet* vererbt und automatisch ausgeführt wird. Wenn die Methode anders hieße, würde gar nichts passieren. Andere Methoden, die automatisch ausgeführt werden, lauten *init, start* und *stop* [43].

Wie man nun ausführlichere und kompliziertere Programme mit Java schreibt, soll der nun schon massenhaften Java-Literatur überlassen sein ([28], [38], [14] oder The Java Programmer's Guide unter *http://java.sun.com/progGuide/index.html*). Hier sollte lediglich ein kleiner Einblick in die Java-Programmierung gewährt werden.

1.3.6 Produkte

Es ist schon erstaunlich, daß nach solch kurzer Zeit nach der Veröffentlichung von Java, schon eine Vielzahl von Java-Tools zur Verfügung stehen. Neben dem JDK von Sun, dem eigentlichen

vorgestellt werden. Diese Vorstellung soll allerdings keine Wertung beinhalten. Diese stehen nur repräsentativ für viele andere verfügbare Werkzeuge für Java.

JDK. Wie schon erwähnt, ist das JDK von Sun via Internet kostenlos zu erhalten. Die Anzahl der Portierung auf die unterschiedlichsten Plattformen steigt natürlich mit der Zeit. Zur Zeit sind für die gängigsten Plattformen das JDK erhältlich. Das JDK selbst, Informationen zu Portierungen und die aktuellen Versionen sind unter der Adresse *http://www.jafasoft.com* oder *http://java.sun.com* erhältlich.

Nach erfolgreicher Installation sind die folgenden Verzeichnisse und Dateien zu finden:

Verzeichnis oder Datei	Beschreibung
bin/	Verzeichnis der Java-Tools (s.u.)
include/	Header-Files für die Einbindung von C/C++ in Java
lib/	Java-Packages und plattformspezifische Bibliotheken (s.u.)
demo/	Beispielprogramme
src.zip	Einige Dateien mit Sourcecode
COPYRIGHT	Informationen über das Copyright
README	Informationen über die aktuelle Version

Die für die Entwicklung von Java benötigten Werkzeuge im *bin*-Verzeichnis sollen an dieser Stelle etwas genauer beschrieben werden [44].

- *appletviewer*

  Wer keinen Web Browser wie HotJava oder Netscape zur Verfügung hat, kann seine Applets auch mit dem Programm *appletviewer* ausführen. Auch sonst kann der *appletviewer* nützlich eingesetzt werden, da er die HTML-Seite durchsucht und nur die Applets in der definierten Größe ausführt. Er kann also als Werkzeug für ein schnelles Überprüfen des Design von Applets verwendet werden.

- *java*

  Der Java-Interpreter (auch virtuelle Maschine genannt) lautet *java*. Mit diesem Interpreter können die Java-Applications gestartet werden. Die Größe dieses Basisinterpreters für die Unterstützung der Basisklassen liegt bei rund 40 KBytes (Alpha-Version). Standardbibliotheken und eine Thread-Unterstützung heben diesen Wert um zusätzliche 175 KBytes an.

- *javac*

  Mit dem Java-Compiler *javac* werden die Java-Quellprogramme in Bytecode übersetzt, einer Art „maschinenunabhängigen Maschinencode“.

- *javadoc*

  Mit Hilfe dieses Dokumentengenerators kann anhand einer vorgegebenen Quelldatei eine Dokumentationsseite im HTML-Format generiert werden. Wie in C/C++ kann in Java ein Kommentar durch /* ... */ geklammert oder mit // ... (gilt bis zum Ende der Zeile) eingeleitet werden. Zusätzlich kann in Java noch ein Kommentar in /** ... */ gekapselt werden. Dieser Kommentar wird von *javadoc* erkannt und wird in die generierte Dokumentationsseite aufgenommen.

Das Werkzeug *javah* dient der Generierung von Header Files für C/C++.

- *javap*

  Auch ein Disassemblierer für Java-Bytecode ist mit dem Werkzeug *javap* vorhanden.

- *jdb*

  Im Beta-Entwicklungspaket findet sich derzeit lediglich ein zeichenorientierter Java-Debugger namens *jdb*. Dieser ist ähnlich zu seinem Vorbild, dem UNIX-Debugger *dbx*, zu bedienen. An einem thread-fähigen, graphischen Debugger wird momentan noch gearbeitet.

In Java zu programmieren ist nicht leichter und nicht schwieriger, als ähnliches in C++ anzugehen. Die meiste Zeit bei der Einarbeitung in Java dürfte es erfordern, die Klassen kennenzulernen, die bereits im Entwicklerpaket vorhanden sind. Die Programmierschnittstelle (engl. API, Application Programming Interface) sind die Java Packages. Packages sind Sammlungen von miteinander verwandten Klassen. Insgesamt besteht die Java-API aus Klassen, die in sechs Packages unterteilt sind, die hier kurz vorgestellt werden sollen [44].

Für eine genauere Beschreibung sei auf die mittlerweile umfangreiche Literatur über die Verwendung der Java-Packages verwiesen ([28] oder Übersicht über Java-Literatur: *http://www.december.com/works/java.bib.html* oder auf die Online-Dokumentation der gesamten API auf dem JavaSoft-Server unter der Adresse: *http://java.sun.com/doc/programmer.html*.

- *java.lang*

  java.lang enthält Klassen, die für die Sprache selbst zentral sind, und muß nicht explizit importiert werden. Die Klasse *Object* ist in diesem Package definiert und bildet die Wurzel der gesamten Klassenhierarchie in Java. Desweiteren sind hier die Klassen der primitiven Datentypen wie *Boolean, Character, Double, Float, Integer* und *Long* (nicht zu verwechseln mit den primitiven Datentypen, die keine Objekte sind) sowie die Zeichenkettenklassen *String* und *StringBuffer* definiert. Einige Low-Level-Methoden für das Java-Laufzeitsystem wird durch die Klasse *Runtime* zur Verfügung gestellt. Die Klasse *Math* unterstützt die Gleitkomma-Arithmetik, und *Process* definiert eine plattformunabhängige Schnittstelle für Prozesse, die außerhalb des Java-Interpreters laufen. Schließlich unterstützt die *Thread*-Klasse den quasiparallelen Ablauf mehrerer Programmteile innerhalb desselben Java-Interpreters.

- *java.applet*

  Dieses Package enthält die Klasse *Applet*. Dieses ist die Superklasse aller Klassen, die über die Einbindung in Web-Dokumente ausführbar sein sollen.

- *java.awt*

  Hinter der Abkürzung ‚awt' verbirgt sich das ‚Abstract Windowing Toolkit'. Die in diesem Package enthaltenen Klassen lassen sich nochmals in folgende Kategorien unterteilen:

  - Klassen für grafische Grundkomponenten (Farben, Fonts, Bilder, Polygone usw.)
  - Klassen für GUI-Komponenten, wie Buttons, Menüs, Listen und Dialog-Boxen.
  - Klassen für das Layout-Management mit verschiedenen Container-Klassen.

Diese Package enthält eine große Menge von Klassen, die die unterschiedlichsten Ein- und Ausgabemechanismen zur Verfügung stellen. *java.io* kann man nochmals zwei Kategorien unterteilen:

  - Stream-Klassen, die Methoden zum Lesen und Schreiben von Daten realisieren.
  - File-Klassen, die Methoden zu Verwaltung von sowohl einzelnen Dateien als auch von ganzen Verzeichnissen bereitstellen.

- *java.net*

  Dieses Package stellt eine flexible und mächtige Infrastruktur zur Verfügung, um mit dem Netz zu arbeiten. Beispielsweise existiert eine *URL*-Klasse für Web-bezogene Funktionen oder Klassen wie *Socket*, *DatagramPacket* und *DatagramSocket* für Interprozeßkommunikation auf hoher und niedriger Stufe. Viele dieser Klassen werden aber von „normalen" Programmen überhaupt nicht verwendet.

- *java.util*

  java.util enthält schließlich einige sehr nützliche Klassen, wie zum Beispiel die Klasse *Hashtable*, die eine Hash-Tabelle oder ein assoziatives Array bereitstellt. Damit können Objekte über zugehörige Schlüssel gespeichert und gelesen werden. Die Klasse *Vector* ist eine Klasse, die ein Array zur Verfügung stellt, das seine Größe ändert, je nach den Objekten, die eingefügt werden.

Object Engineering Workbench for Java: OEW/Java. Die Innovative Software GmbH bietet seit Februar 1996 ein plattformübergreifendes, grafisches Java-Entwicklungswerkzeug an: Das Object Engineering Workbench for Java (OEW/Java). Die Version OEW/Java V0.9 kann zunächst gratis von der Homepage der Innovative Software GmbH (*http://www.isg.de*) bezogen werden.

Da die Firma mit OEW/C++ eine sich bewährte Forward- und Reverse-Engineering-Technologie erprobt hat, soll OEW/Java die Entwicklungszeiten für Java-Applikationen und Java-Applets erheblich steigern. Der Syntax-Highlighting-Editor ermöglicht komfortable Eingabe von Java-Funktionen. Zur Modellierung von Klassen und Beziehungen genügen wenige Mausklicks. Dieses Werkzeug unterstützt also die in Abschnitt 4.6 angesprochenen Entwurfsrichtlinien. Ein Code-Generator setzt die Entwürfe anschließend automatisch in Java-Programme um. Der Aufruf des Java-Compilers ist bereits in die OEW-Entwicklungsumgebung integriert.

Die Symbolic-Parser-Technologie von OEW bietet Reverse Engineering für vorhandenen Java-Code und ermöglicht damit eine einfache Wiederverwendung von Java-Klassen. Ein Programmierer kann vollständige Applikationen mit grafischen Übersichten, Werkzeugen und Programmierdialogen abbilden und dem vorhandenen Code neue Methoden hinzufügen. Auch ist OEW/Java in der Lage, Reverse Engineering von fehlerhaftem Code durchzuführen, indem es sich beim Parsen an Symbolen im Quelltext orientiert und versucht, Syntax-Fehler zu übergehen.

Ab sofort stehen Versionen für Sun Solaris, Windows 95/NT, HP-UX und OS/2 Warp zur Verfügung. Portierungen nach PowerMac und IBM AIX sind in Vorbereitung [53].

JFactory 1.0. Das Produkt JFactory der US-amerikanischen Softwarefirma Rogue Wave stellt ein (allerdings) kommerzielles grafisches Entwicklungswerkzeug für Java dar. JFactory setzt eine

tan Versionen für Sun Solaris und Windows 95/NT. JFactory kann man übrigens schriftlich oder online via Internet (*http://www2.roguewave.com/rwpav/products/ JFactory*) beziehen. Der Preis beläuft sich auf 748 DM.

Daß Rogue Wave diese Umgebung derart schnell ins Rennen schicken konnte, läßt sich einfach erklären. Neben dem JDK von Sun konnte die Softwareschmiede auf die hauseigene GUI-Klassenbibliothek zApp zurückgreifen. Sie erlaubt plattformunabhängiges Entwickeln grafischer Oberflächen und ist mittlerweile für zahlreiche Betriebssysteme verfügbar. Da auch die Benutzungsoberfläche von JFactory offensichtlich mit zApp entwickelt wurde, dürfte ein Portieren des Produkts auf andere Systeme sehr schnell möglich sein.

Die Vorteile von JFactory dürfte zum einen in dem geringen belegten Plattenspeicherplatz von nur 3 MB (zuzüglich des Platzes für das JDK). Desweiteren dürfte derjenige, der sich mit anderen Programmierumgebungen, wie z.B. Visual Basic oder Delphi, auskennt, sich sehr schnell in der Philosophie der Entwicklungsumgebung von JFactory zurechtfinden. Eine genauere Beschreibung des Produkts ist auf der WWW-Seite der Firma Rogue Wave zu finden.

Spannend dürfte sein, ob sich JFactory gegen andere zukünftige Java-Entwicklungsumgebungen wird behaupten können. Konkurrenten wie Borland, IBM, SGI, Sun oder auch Microsoft hocken schon in den Startlöchern [47].

1.3.7 Zusammenfassung

Auf den momentan gängigsten Betriebssysteme SunOS, SGI, DEC OSF/1, Windows 95/NT und Linux ist das JDK von Sun erhältlich. Portierungen auf „exotischere“ Plattformen sind in Vorbereitung.

Bei der Programmiersprache Java ragen die Merkmale architekturneutral, robust, sicher und objektorientiert heraus. Auf die vielen C/C++-Programmieren wurde bei der Entwicklung von Java Rücksicht genommen. Java ähnelt sehr C++ und macht es Umsteigern leicht, Java schnell zu erlernen.

Das JDK von Sun ist ein komplettes und via Internet kostenloses Werkzeug zur Erstellung von Java-Programmen und Java-Applets. Aber auch schon heute gibt es kommerzielle GUI-Builder für Java, die raschere Entwicklungszyklen versprechen.

1.4 Java – Zukunft

Die zu diesem Zeitpunkt bestehende Werkzeuge zur Erstellung von Java-Programmen und die Mächtigkeit der Programmiersprache Java entspricht zweifelsohne vergleichbaren kommerziellen Produkten anderer Firmen und übertrifft diese sogar betreffend Portabilität und Netzwerkeinsatz. Aber in Zukunft sollen noch andere Features hinzukommen, die sehr vielversprechend sind. Allerdings wird erst die nähere Zukunft beweisen, wie weit die Visionen realisierbar sind und ob sie überhaupt Akzeptanz bei den Anwendern finden wird.

1.4.1 Integration von Datenbanken

„Das World Wide Web ermöglicht es Unternehmen wie auch Privatnutzern, auf Milliarden Byte unstrukturierter Daten zuzugreifen, nicht aber auf Trillionen Bytes, die in konventionellen Datenbanken gehalten werden. Das JDBC (Java Database Connectivity) API (Application Programmers Interface) erweitert die Möglichkeiten des Web, indem es erlaubt, daß spezielle Java

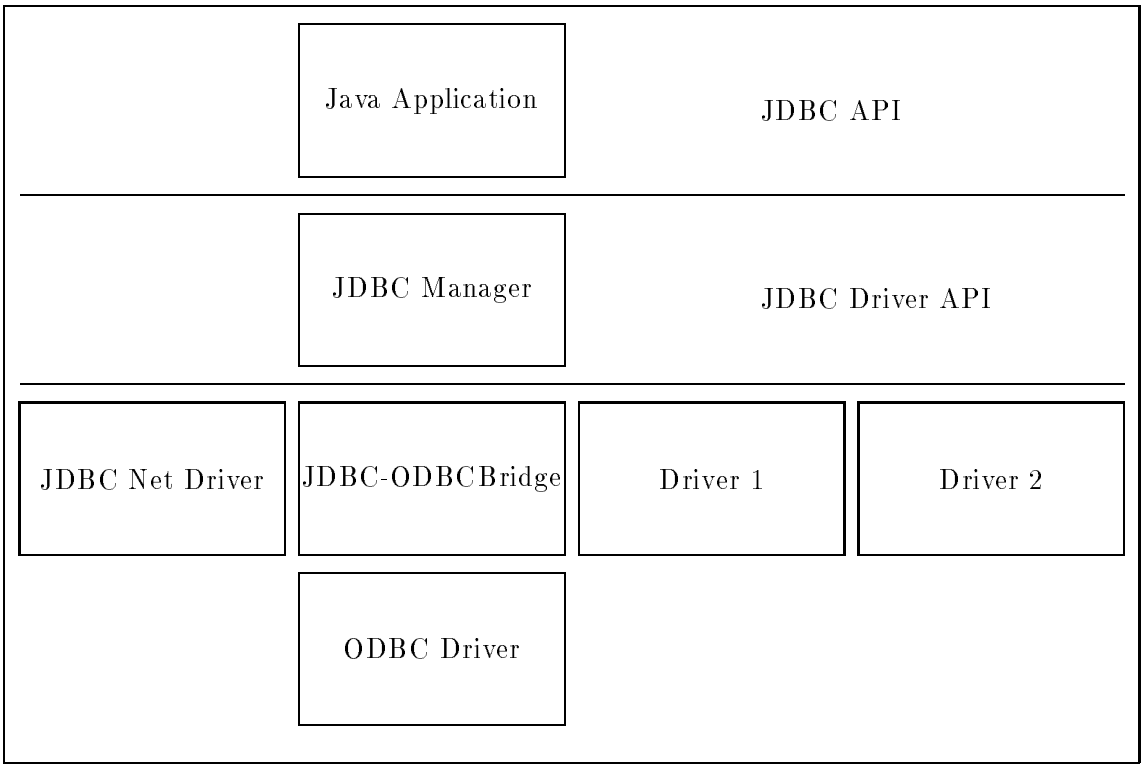
JavaSoft [53].

Java ist das ideale Werkzeug für Client/Server-Computing und nicht nur für Entwicklungen von Applets. („Wer fährt denn schon einen Porsche, um nur beim Bäcker die Brötchen zu holen?“) Mit der Hilfe von Java werden bei Verwendung des JDBC plattformunabhängige Datenbankentwicklungen möglich. Somit soll Java mit dem JDBC zur primären Anwendungsplattform von Unternehmen werden, die einen Zugriff auf ihre Unternehmensdaten benötigen. Dank der engen Kooperation mit Anbietern von Datenbanken, Werkzeugen und Treibern arbeitet JavaSoft an weiteren Optimierungen des JDBC. Bereits heute entwickeln Unternehmen wie Informix, IBM, Oracle, Borland und OpenLink Software auf Basis des JDBC API [55].

JDBC. Die Entwicklung des JDBC wurde schnell vorangetrieben und die Spezifikation inklusive einer Vorabversion zur Analyse wurde am 8. März 1996 veröffentlicht. Im Sommer 1996 werden eine Beta-Version und auch schon Produkte der schon erwähnten Unternehmen erwartet.

Das JDBC stellt eine Schnittstelle zur Kommunikation mit den unterschiedlichen Datenbanken zur Verfügung. Das Konzept ähnelt Microsofts Open Database Connectivity (ODBC), das sich zu einem Standard für PC und LANs entwickelt hat. Das JDBC basiert auch auf dem X/Open SQL Call Level Interface, wie auch das ODBC. Wegen diesen schon existierenden Vorüberlegungen und Standards konnte das JDBC so schnell entwickelt werden.

Die Klassen für die Datenbankoperationen sind komplett in Java geschrieben. Hierdurch wird für Java-Datenbank-Applikationen die Sicherheit, Robustheit und Portabilität beibehalten, die durch die Sprache Java vorgegeben wird.



Wie obige Abbildung verdeutlicht, besteht die JDBC aus zwei Hauptschichten. Die JDBC API Schicht unterstützt Applikation zu JDBC-Manager Kommunikation und die JDBC API Driver Schicht unterstützt die JDBC Manager zu Treiber Kommunikation. Der JDBC Manager kom-

folgende Packages realisiert, die das JDBC API bilden:

- *java.sql.Environment*
  Erlaubt das Bilden von neuen Datenbankverbindungen
- *java.sql.Connection*
  Verbindungsspezifische Datenstrukturen
- *java.sql.Statement*
  Container-Klasse für embedded SQL statements
- *java.sql.ResultSet*
  Zugangskontrolle für Ergebnisse eines Ausdrucks (access control to results of a statement)

Im JDBC Driver API Package besteht lediglich aus der Klasse *java.sql.Driver*. Jeder Treiber muß allerdings Implementierungen für folgende abstrakte Klassen vorsehen: *java.sql.Connection, java.sql.Statement, java.sql.PreparedStatement, java.sql.CallableStatement* und *java.sql.ResultSet*.

JDBC über das Internet. Die Sicherheit wird bereits in der Sprache Java verankert, jedoch andere Elemente wie adressenspezifische Instanzen von Datenbankanwendungen müssen in einem Aufruf aufgenommen werden. Deshalb macht JDBC Anleihen von der Uniform Resource Locator (URL) Syntax für globale und eindeutige Datenbankadressierung. Das Schema für eine Datenbankadressierung sieht folgendermaßen aus: *jdbc:odbc://host.domain.com:400/databasefile*.

Hier wird also JDBC als das Hauptprotokoll und ODBC als Subprotokoll identifiziert. Wie der JDBC Manager auf die Datenbankdatei zugreifen wird, ist im Subprotokoll festgelegt. In diesem Fall wird über die JDBC-ODBC-Brücke auf die Datenbankdatei zugegriffen. Wie bei URL bezeichnet der Rest der Adresse den Hostname, die Port-Nummer, das Verzeichnis und der Namen der Datenbasis.

1.4.2 Einbettung von Java in Betriebssysteme

Durch ein besonderes Lizensierungsverfahren von Sun wird es nun möglich, daß andere Unternehmen Java direkt in ihre Betriebssysteme integrieren können. Bislang konnte Java lediglich in Applikationen oder direkt in die Hardware integriert werden. Dadurch kann jedes Betriebssystem durch das neue Leistungsmerkmal „Java-tauglich“ bereichert werden, wenn sich der Hersteller dazu entschließt, Java in ihr Betriebssystem einzubetten [55].

Einige der Unternehmen und Betriebssysteme seien hier einmal genannt.

- Apple plant, Java in die Macintosh- und Newton-Familie, Autoren-Programme und Internet-Server zu integrieren.
- IBM wird Java in die nächste Version von Lotus Notes und in die Betriebssysteme OS/2, S/390 und OS/400 einbetten.
- Microsoft wird inzwischen auch Java in zukünftigen Versionen von Windows 95, Windows NT, Internet Explorer 3.0 und ActiveX einbeziehen.
- SCO wird Java in ihre SCO-Internet-Familie und in Gemini integrieren.

- Novell will schließlich den Java-Interpretierer noch in diesem Jahr in NetWare integrieren und im nächsten Jahr wird es noch in GroupWise eingebettet.

Da die „Multis“ unter den Computer- und Software-Firmen Java auf weiter Ebene unterstützen werden, ist der Siegeszug von Java als künftige Standardentwicklungsplattform für Internet-Applikationen nicht mehr aufzuhalten.

1.4.3 Spezielle Java-Prozessoren

Im Februar 1996 wurden die erste, auf Java optimierte Mikroprozessorfamilie von Sun Microelectronics vorgestellt. Das Java Prozessorspektrum umfaßt zunächst drei Mikroprozessoren, die im Vergleich zu herkömmlichen General Purpose Typen in realen Anwendungen ein Vielfaches der Leistung erreichen und zu einem Bruchteil der bisherigen Kosten zu erhalten sind, das behauptet Scott McNealy, der Chairman von Sun Microelectronics. Somit ist der Weg zu NC (Network Computers) geebnet und auch andere Internet-fremde Produkte können von dieser Technologie profitieren.

Die Java Prozessorfamilie besteht aus den drei Mikroprozessorlinien picoJAVA, microJAVA und UltraJAVA. Der picoJAVA ist für den Einsatz in Mobiltelefonen, Druckern und andere Konsum- bzw. Peripheriegeräten gedacht. Der Preis soll unter 25 Dollar betragen und wird voraussichtlich Mitte 1996 erhältlich sein. Der microJAVA-Prozessor basiert auf dem picoJAVA und ist um applikationsspezifische I/Os, Speicherbereiche und Kommunikations- und Steuerungsfunktionen angereichert. Das Einsatzgebiet dieses Prozessors soll allgemein der industrielle Produktmarkt sein. Der Prozessor wird zwischen 25 und 100 Dollar kosten. Muster des ersten microJAVA-Prozessors werden im ersten Quartal 1997 erwartet.

Der UltraJAVA ist der schnellste aller Java-Prozessoren. Der Einsatz dieser Prozessorlinie soll in fortschrittlichen 3-D- und Multimedia-Produkten erfolgen. Deshalb ist er auch mit zahlreichen Grafikfunktionen und allgemeinen Optimierungen angereichert worden. Ab 100 Dollar sollen ab Ende 1997 erste Muster der UltraJAVA-Prozessoren erhältlich sein [53].

1.4.4 Network Computer (NC)

Anfang Mai 1996 brachten die Firmen Sun Microsystems, Apple Computer, IBM, Netscape Communications und Oracle ein erstes Referenzprofil eines NC heraus. Dieses Referenzprofil legt die grundlegenden Strukturen und Internas der neuen NCs offen, die nur etwa 500 bis 1000 Dollar kosten sollen, also weit weniger als die Hälfte des Preises eines PCs. Diese NCs sind deshalb so billig, da sie ohne große Massenspeicher auskommen und einen einfachen Aufbau besitzen. Wohlgemerkt sollen die „alten“ PCs NC-kompatibel sein.

Im Referenzprofil sollen als Standards für IP-Protokolle TCP/IP, FTP, telnet, UDP, SNMP und Suns NFS vorgesehen werden. Für World Wide Web gilt dasselbe für HTML, HTTP und Java. Die Mail-Protokolle SMTP, IMAP4 und POP3 sowie die Multimedia-Formate JPEG, GIF, WAV und AU sollen auf jeden Fall unterstützt werden.

Ein Entwurf der NC-Spezifikation soll ab Juli 1996 verfügbar sein und kann unter der Adresse *http://www.nc.ihost.com* eingesehen werden. Zusätzliche Referenzprofile sollen in Zukunft folgen [55].

Zum Abschluß des Ausblicks in die Zukunft von Java sollen noch Ankündigungen von JavaSoft angesprochen werden, die den Stellenwert von Java als Plattform für Internet-Programmentwicklung weiter erhöhen soll, deren Realisierung aber (wenn überhaupt) erst in den nächsten Monaten und Jahren erfolgen soll [55].

- Eine Message-Spezifikation wurde angekündigt, mit deren Hilfe Entwickler Java-Programme schreiben können, die wiederum andere Java-Programme kontrollieren oder auch auf dessen Daten zugreifen können. Der Codename dieses Vorhabens lautet „Java Beans“ und funktioniert ähnlich zu Microsofts Active X, ehemals OLE, und Open Doc.
- Neue APIs für Audio-, Video-, Security-, Serverbasierte Applikationen, kommerzielle Internetnutzung und auf Internet basierende Unternehmensmanagement-Unterstützung sollen verfügbar sein.
- Ein erstes Java-Betriebssystem, JavaOS genannt, soll für Network Computer veröffentlicht werden.
- Ein völlig neuentwickelter HotJava-Browser soll ein „Operating Environment“ darstellen, der sich je nach laufender Java-Applikation selbst konfiguriert. So soll zum Beispiel bei einer Anwahl einer Mail-Adresse sich der HotJava-Browser automatisch in ein E-Mail-Umgebung umkonfigurieren, ohne daß ein separates Mail-Programm gestartet werden muß.
- Schließlich soll es einen generellen Entwickler-Online-Support geben.

1.4.6 Offene Fragen der Sicherheit

Trotz aller Sicherheitsvorkehrungen gibt es dennoch Berichte über neue Sicherheitslücken. Durch den Einsatz von Java im Internet, einem mittlerweile hochkomplexen und fast undurchschaubarem Gebilde, kann Sicherheit nicht zu 100% garantiert werden. Aber die immer größer werdende Anzahl von Java-Benutzer kann immer häufiger eventuell versteckte Fehler aufdecken, die dann durch eine Herausgabe einer neuen Version beseitigt werden.

Zum Beispiel ist ein Problem im Zusammenhang mit DNS (Domain Name Services) bekannt, das erlaubt, die die Zugriffskontrolle für Netzwerkoperationen zu umgehen. Dazu bedarf es allerdings eines sehr großen Spezialwissens, aber es zeigt, daß nach einiger Zeit immer wieder Fehler aufgedeckt werden. Trotz des Sicherheitskonzept von Java, das wesentlich weitergeht als bei anderen Systemen, ist ein uneingeschränktes Vertrauen in Java-Applets noch nicht angebracht!

Wichtig ist nur, daß man beim Bekanntwerden neuer Fehler (WWW-Seite von JavaSoft oder Fachzeitschriften lesen) sich schleunigst darum bemüht, die neueste Version von Java zu besorgen, um immer den größtmöglichsten Schutz vor Fehlern zu besitzen.

Ein großer Schritt, der Java noch bevorsteht, ist der Einsatz kryptographischer Verfahren, wie zum Beispiel Public-Key-Systemen. Solche Systeme können beim Übertragen von Java-Klassen über das Netz zur eindeutigen Identifizierung des Herstellers benutzt werden. Somit wird nicht nur die Ausführung von Java-Programmen noch sicherer, sondern eine Menge von sicherheitskritischen Anwendungen könnten über das Internet realisiert werden. Man denke nur an Finanztransaktionen über das Internet, oder an eine WWW-Bank, über die man online auf einer WWW-Seite seine Überweisungen tätigen kann. Eine Unzahl von kommerziellen Anwendungen könnten dann realisiert werden. Ob das aber überhaupt das Internet verkraften kann, wird die Zukunft zeigen [56].

Es gibt viele Überlegungen, wie Java zu dem Internet-Entwicklungs-Paket schlechthin gemacht werden kann. Es wird möglich sein, Java-Applikationen zu schreiben, die auf Datenbanken zugreifen können. Hierzu wird die Java Database Connectivity (JDBC) API entwickelt. Verteilte und portable Datenbankanwendungen können mit dem JDBC API realisiert werden.

Desweiteren wird Java in die gängigsten Betriebssysteme integeriert werden, so daß eine noch größere Anzahl von Entwicklern Java-Applikationen entwerfen können.

Schließlich werden sogar eigenständige Java-Prozessoren demnächst erhältlich sein. Gemäß ihrer Realisierung können sie in den verschiedensten elektrischen Geräten, wie zum Beispiel einfachen Mobiltelefonen bis hin zu einer 3-D-Multimedia-Grafikstation, eingesetzt werden.

Unbedingt zu beachten ist aber, daß bei einem solch jungen Produkt wie Java immer noch Fehler versteckt sind. Es ist also ratsam, nicht jedem Applet blind zu vertrauen. Man sollte immer sicherstellen, daß man die neueste Version von Java benutzt.

Seminarbeitrag von **Daniel Pfeifer**

Dieser Artikel gibt einen Überblick über einige Konzepte der Programmiersprache Java. Es werden unter anderem folgende Themen behandelt: Primitive Datentypen, Ausdrücke, Kontrollfluß, Klassen sowie Modifikatoren und Sichtbarkeitsregeln. Weitere Sprachelemente wie Paketbildung, Interfaces und Exceptions werden hier nur gestreift und sind Thema des nachfolgenden Artikels.

Ich setze voraus, daß der Leser Kenntnisse über Konzepte von Programmiersprachen besitzt bzw. eine Programmiersprache kennt.

Besondere Eigenschaften von Java

- Java ist eine komplett objektorientierte imperative Programmiersprache.
- Die Syntax ist stark an C und C++ angelehnt.
- Bisher ist Java eine Mischform aus Compiler- und Interpretersprache: Ein Compiler übersetzt Java-Quellcode in einen standardisierten Zwischencode für eine virtuelle Maschine, ein Interpreter führt diesen sogenannten Bytecode dann aus.
- In Java gibt es – anders als etwa in C – wenig "implementierungsabhängiges": So sind z. B. Größe, Format und Operationen auf den primitven Datentypen genau spezifiziert (siehe 2.2).
- Das Typsystem von Java ist sehr streng und wird auch zur Laufzeit unterstützt. Der Typ jedes Objekts kann zur Laufzeit ermittelt werden. Es sind ausschließlich typkonforme Zugriffe auf Objekte möglich.
- Java unterstützt (zumindest auf Klassen) keine Mehrfachvererbung. Es gibt jedoch andere Sprachkonstrukte, um diesen Mangel auszugleichen (siehe 2.3.4 bzw. nachfolgenden Artikel).
- Durch eine interne Speicherbereinigung wird „verlorener Speicher" immer wieder zuückgeholt. Der Programmierer braucht sich also um die Freigabe von Objektspeicher nicht zu kümmern.
- Java ist sehr sicher: Einmal übersetzte Programme können (eigentlich) nicht abstürzen.
- Für die Namensgebung von Java-Programmteilen gibt es genaue Konventionen, so daß diese Programmteile auch welt- bzw. Internet-weit eindeutig identifiziert und aufgefunden werden können. Dies erleichtert die Wiederverwendung erheblich.

Java Programme werden im Unicode-Zeichensatz[1] geschrieben.

Beim Einlesen des Quellprogrammes finden zu allererst drei einfache Transformationen auf dem einlaufenden Zeichenstrom statt:

1. Escape-Sequenzen in ASCII-Code der Form `\u`*4-Hex-Ziffern* werden in entsprechende „echte“ Unicode-Zeichen übersetzt. Dabei entspricht eine solche Escape-Sequenz einem bestimmten Unicode-Zeichen.
2. Der Eingabestrom wird durch die Zeilentrenner Carriage Return `CR` und Linefeed `LF` in Zeilen zergliedert.
3. Bestimmte Zeichenfolgen werden zu Tokens zusammengefaßt: Zeichen wie Leerzeichen `SPACE` und Carriage Return `CR` dienen dabei als Trennzeichen.

2.1.1 Kommentare

Es gibt drei Arten von Kommentaren:

`/* text */` Alles zwischen `/*` und `*/` einschließlich ist Kommentar und wird ignoriert.

`/** text */` Wie obiger Kommentar, nur wird `text` evtl. für Werkzeuge zur automatischen Dokumentationsgenerierung von Quellcode verwendet.

`// text` solche Kommentare gehen bis zum Zeilenende.

2.1.2 Bezeichner

Ein Bezeichner ist eine Unicode-Zeichenfolge unbeschränkter Länge aus Buchstaben und Ziffern. Das erste Zeichen muß ein Buchstabe sein. Ein Bezeichner darf kein Schlüsselwort sein.

2.2 Primitive Typen

Es gibt vier Gruppen primitiver Typen nämlich Boolscher Typ, Integer-Typen, Float-Typen und Character-Typ.

2.2.1 Der Boolsche Typ

Der Boolsche Typ `boolean` hat zwei Werte, denen die Literale `true` und `false` entsprechen. Er ist der Resultattyp aller Vergleichsoperatoren und wird in den Kontrollflußkonstrukten `if`, `while`, `do` und `for` im Bedingungsteil als Resultattyp gefordert. Anders als in C sind Integer-Typen dort nicht erlaubt.

Für `boolean` gibt es folgende Operatoren mit Syntax und Semantik analog zu C:

- logische Vergleichsoperatoren `==`, `!=`,

[1] Der Unicode Zeichensatz ist ein genormter internationaler Zeichensatz, der den ASCII-Zeichensatz als Teilmenge enthält. In ihm sind die meisten auf der Welt verwendeten Schriftzeichen repäsentiert.

Oder ^ (hier keine Kurzauswertung),

- logisches Und `&&` und logisches Oder `||` *mit* Kurzauswertung,
- einen Bedingungsoperator der Form *Test*`?`*Ausdruck1*`:`*Ausdruck2*, bei dem zuerst der Ausdruck *Test*, der vom Typ `boolean` sein muß, ausgewertet wird. Falls *Test* `true` liefert, wird *Ausdruck1* ausgewertet und geliefert, andernfalls *Ausdruck2*.

2.2.2 Integer-Typen

Die Werte der Integer-Typen `byte`, `short`, `int` und `long` sind in dieser Reihenfolge 8-bit, 16-bit, 32-bit bzw. 64-bit vorzeichenbehaftete Integers (Ganzzahlen) im Zweierkomplement.

Die Wertebereiche für diese Typen sind entsprechend festgelegt; so geht etwa der Wertebereich für den Typ `byte` von -256 bis +255.

Auch hier gibt es Literale, um Werte dieser Typen im Quellcode darstellen zu können. Die Syntax ist ähnlich der in C; Oktalzahlen beginnen mit `0`, Hexzahlen mit `0x` oder `0X`.

Beispiele für `int`-Literale:

```
1    0666    0xAffe    0XCafe
```

Beispiele für `long`-Literale:

```
1l   0666L   0xAffeL   0XCafel
```

Für Integer-Typen gibt es folgende Operatoren mit Syntax und Bedeutung analog zu C:

- die Vergleichsoperatoren `==`, `!=`,
- relationale Operatoren `<`, `>`, `<=` und `>=`,
- unäres `+` und `-`,
- Inkrement- und Dekrementoperator `++`, `--`,
- arithmetische Operatoren `+`, `-`, `*`, `/`,`%`,
- Shift-Operatoren mit und ohne Vorzeichenverschiebung, `<<`, `>>`, `<<<`, `>>>`,
- bitweise logische Operatoren `~`, `&`, `|`, `^`.

Das Ergebnis arithmetischer Operationen bleibt immer `int` bzw. `long`. Über- oder Unterlauf werden dabei nicht angezeigt. Falls bei `/` oder `%` durch Null dividiert wird, wird eine sogenannte Exception ausgelöst. Exceptions sind Signale, die Ausnahme- und Fehlerzustände im Programmablauf anzeigen. Mehr dazu im nachfolgenden Artikel.

Für den Restoperator `%` gilt die Identität `(a/b)*b+a%b` gleich `a`.

Die Werte der Float-Typen `float` und `double` sind 32-bit bzw. 64-bit Gleitpunktzahlen im IEEE 754-Format[2]. Damit sind das interne Format und die Bereichsgrenzen für `float` und `double` festgelegt.

Beispiele für `float`-Literale:

```
1e1f    2.f    .3f    3.14f    6.02e+23f
```

Beispiele für `double`-Literale:

```
1e1     2      .3     3.14     1e-9d
```

Für Float-Typen gibt es folgende Operatoren:

- Vergleichsoperatoren `==`, `!=`,
- relationale Operatoren `<`, `>`, `<=` und `>=`,
- unäres `+` und `-`,
- Inkrement- und Dekrementoperatoren `++` und `--`,
- arithmetische Operatoren `+`, `-`, `*`, `/`, `%`.

Die Java-Gleitpunktarithmetik löst keine Exceptions aus. Bei Überlauf, Division durch Null usw. werden gemäß IEEE-Norm besondere Werte erzeugt, die jedoch keine Zahlen im eigentlichen Sinne sind, z.B. der Wert `NaN` (not a number) bei Division durch Null. Achtung: Vergleichsoperationen auf `NaN` liefern immer `false`.

2.2.4 Der Character-Typ

Der Character-Typ `char` hat einzelne Unicode-Zeichen als Wert. Dies sind 16-bit Größen. Literale dieses Typs haben die Form '*einzelnesZeichen*' oder '*Escape*'. Dabei darf *einzelnesZeichen* bis auf Sonderzeichen wie Linefeed `LF` oder Quote ', ein beliebiges Unicode-Zeichen sein. Für die Sonderzeichen Zeichen benutzt man *Escape*-Sequenzen ähnlich wie in C, z.B. `'\n'` für Linefeed `LF`.

Achtung: Die unter 2.1 erwähnten Transformationen finden ganz zu Beginn der Übersetzung statt, so daß etwa der Ausdruck `'\u000a'` zu einem Fehler führt, weil dabei `\u000a` zu Linefeed `LF` transformiert und dann als echter Zeilentrenner interpretiert wird.

2.2.5 Standard-Default-Werte

Variablen haben in Java-Programmen niemals einen undefinierten Wert. Wird eine Variable erzeugt, bekommt sie, falls sie nicht explizit initialisiert wird, abhängig von ihrem Typ einen Default-Wert zugewiesen.

Die Default-Werte bei primitiven Typen sind:

[2]Dies ist eine internationale Norm für Gleitpunktarithmetik.

- `0.0f` für `float` und `0.0d` für `double`,
- `false` für `boolean`,
- `'\u0000'` für `char`.

2.2.6 Typumwandlungen auf primitiven Typen

Im Allgemeinen gibt es in Java drei verschiedene Arten von Typumwandlungen: Casting (die erzwungene Typumwandlung), Zuweisungsumwandlung und arithmetische Umwandlung.

Das Casting ist die mächtigste und einzige vom Programmierer explizit vorgenommene Typumwandlung. Es wird erreicht durch Voranstellen des eingeklammerten Typs, zu dem hin gewandelt werden soll, vor den entsprechenden Ausdruck, z. B. wandelt der Ausdruck `(int)3.5` ein `double`-Literal nach `int`. Aber auch durch Casting sind nicht beliebige Typumwandlungen möglich: So kann z.B. niemals ein primitiver Typ auf einen Referenztyp (siehe 2.3) gewandelt werden.

Typumwandlungen von „kleineren“ auf „größere“ arithmetische Typen oder `char`, wie z.B. von `char` nach `long` oder von `int` nach `float` sind immer möglich. Sie erhalten die numerische Information außer im Fall `int` bzw. `long` nach `float` oder `long` nach `double`. Die dann notwendige Rundung erfolgt nach IEEE-Norm.

Im umgekehrten Fall, also bei Umwandlungen von „größeren“ auf „kleinere“ arithmetische Typen oder `char` geht numerische Information verloren. Bei Umwandlungen von Integer-Typen in kleinere Integer-Typen oder `char` werden dann einfach die höchstwertigen Bits abgeschnitten. Bei Umwandlungen von `float` oder `double` in einen Integer-Typ oder `char` wird auf eine passende Ganzzahl gerundet. Ist dies nicht möglich, weil die Gleitpunktzahl zu groß oder zu klein war, wird die größte bzw. kleinste Zahl im Zieltyp erzeugt.

Zuweisungsumwandlungen finden statt, wenn ein Wert eines Typs an eine Variable eines anderen Typs zugewiesen wird. Sie führen zu keinem Informationsverlust und machen keine Laufzeitüberprüfungen notwendig. In nachfolgenden beiden Listen kann jeweils ein links stehender Typ auf einen weiter rechts stehenden Typ mittels Zuweisung umgewandelt werden[3]:

```
byte   short   int   long   float    double
char   int   long   float   double
```

Arithmetische Umwandlung wird bei binären Operatoren durchgeführt, wenn der eine Operand einen „größeren“ Typ besitzt als der andere: Es wird dann der „kleinere“ auf den „größeren“ Typ umgewandelt. Bei unären Operatoren werden `short` oder `byte` Operanden nach `int` umgewandelt.

Achtung: Es gibt *keine* Umwandlungen von oder nach `boolean`.

2.3 Klassen

Klassen sind die wichtigsten Sprachkonstrukte von Java. Fast der ganze Quellcode eines typischen Programms besteht aus Deklarationen von Klassen und deren Komponenten.

[3] So ähnlich steht das in der Java-Spezifikation. Ich sehe hier allerdings einen Widerspruch zur oben erwähnten Informationserhaltung etwa im Fall: Umwandlung von `long` nach `float`.

welche Auswirkungen das auf die Sichtbarkeit deklarierter Klassenkomponenten hat, wird in nachfolgendem Artikel diskutiert.

2.3.1 Klassendeklarationen

Eine Klassendeklaration wird durch das Schlüsselwort `class` eingeleitet und beinhaltet sogenannte Felder als Elemente. Felder können Variablen, Methoden oder Konstruktoren sein.

Beispiel:

```
class EineEinfacheKlasse {
      boolean eineVariable;
      int eineMethode() {
           if (eineVariable = true)
             return 1;
           else
             return 0;
      }
};
```

Hier wurde eine Klasse namens `EineEinfacheKlasse` deklariert, in der weiterhin die Variable `eineVariable` und die Methode `eineMethode` vereinbart wurden.

Durch eine Klassendeklaration wird gleichzeitig ein neuer Typ deklariert mit dem Namen der Klasse. Ein solcher sogenannter Referenztyp kann ebenso wie die primitiven Datentypen bei der Deklaration von Variablen verwendet werden.

2.3.2 Variablen als Felder

Variablen sind in Java getypt und können mit einem primitiven Typ oder mit einem Referenztyp deklariert werden. Mit einem primitiven Typ deklarierte Variablen sind im Prinzip Speicherorte zur Aufnahme eines Wertes des entsprechenden Typs. Im Gegensatz dazu können mit einem Referenztyp deklarierte Variable keine Objekte der entsprechenden Klasse aufnehmen sondern nur *Verweise auf Objekte* dieser Klasse.

Für Referenztypen gibt es – ähnlich wie bei primitiven Typen – den einheitlichen Standard-Default-Wert `null`, der anzeigt, daß eine Variable des gegebenen Referenztyps auf kein Objekt verweist.

Innerhalb einer Klassendeklaration können beliebig viele Variablen deklariert werden. Eine solche Deklaration besteht aufeinanderfolgend aus einem oder mehreren optionalen Modifikatoren, dem Variablentyp, einem oder mehreren Variablennamen und einer optionalen Initialisierung. Die Initialisierung besteht aus einem Gleichheitszeichen `=` gefolgt von einem Ausdruck (siehe 2.6) oder einem Array-Initialisator (siehe 2.4.2).

Beispiele für Variablendeklarationen:

```
int x,y;
float z = 1.0;
Object O = foo();
```

`1.0` initialisiert wurde. Die Variable `O` mit Referenztyp `Object` wurde durch die Methode `foo()` initialisiert.

2.3.3 Methoden als Felder

Methoden sind die ausführbaren Felder einer Klasse. Die Syntax ist ähnlich wie für Funktionen in C.

Beispiele:

```
void eineSehrEinfacheMethode() {
}

int eineMethodeDieEinsLiefert() {
     return 1;
}

int eineMethodeMitFormalenParametern(boolean parameter, int x) {
     if (parameter)
          return 1;
     else
          return x;
}
```

Die Beispiele sollen zeigen, wie die Deklaration einer Methode im Prinzip aufgebaut ist:

- Der Typ ganz links ist der Resultattyp der Methode. Dies kann natürlich auch ein Referenztyp sein. Falls die Methode kein Resultat liefert, wird dies durch das Schlüsselwort `void` angezeigt.
- Es folgt der Name der Methode.
- Dann schließt sich die geklammerte Liste der formalen Parameter an. Auch wenn die Liste leer ist, müssen hier Klammern stehen (siehe erstes und zweites Beispiel).

  Jeder Parameter ist eine Variable, der ihr Typ vorausgeht. Die so deklarierten Parameter werden bei Aufruf einer Methode neu erzeugt und mit *Kopien* der entsprechenden Werte aus dem Aufruf initialisiert. Es herrscht also ausschließlich Wertübergabe. Dies gilt auch für Referenztypen, da in diesem Fall die Parameter *Kopien der Verweise* aufnehmen.

  Die so initialisierten Parameter stehen dann als lokale Variable im Rumpf der Methode zur Verfügung. Mit dem Beenden der Ausführung einer Methode endet auch die Existenz der Parameter.
- Der Rumpf der Methode steht in den geschweiften Klammern.
- Einer Methodendeklaration dürfen sogenannte Modifikatoren vorausgehen. Siehe hierzu 2.3.6.

Im Rumpf können nun weitere lokale Variable deklariert werden. Mit Hilfe von Anweisungen werden hier Algorithmen implementiert. Jede Anweisung wird durch ein Semikolon abgeschlossen.

endet in ihrer Ausführung entweder mit dem Erreichen der äußersten geschweiften Klammer oder durch eine `return`-Anweisung.

Eine `return`-Anweisung hat die Form `return` *Ausdruck*. Dabei wird der Wert von *Ausdruck* als Resultat geliefert. Dieser Wert muß ein (Unter-)typ (siehe 2.3.4) des vereinbarten Resultattyps der Methode sein. `return`-Anweisungen können mehrfach in einer Methode auftauchen.

Eine Methode kann Exceptions auslösen. Mit Hilfe der optionalen `throws`-Klausel werden diese bei der Deklaration spezifiziert, mehr dazu in nachfolgendem Artikel. Ebenfalls im nächsten Artikel wird diskutiert, wie die Ausführung einer Methode durch eine `raise`-Anweisung beendet werden kann.

2.3.4 Vererbung auf Klassen

In Java gibt es auf Klassen nur Einfachvererbung, außerdem hat jede Klasse eines Programmes mit Ausnahme der Klasse `Object` (genau) einen Vorgänger im Vererbungsgraphen. Der Vererbungsgraph ist also ein Baum mit der allgemeinsten abstrakten Klasse `Object` als Wurzel.

Durch das Schlüsselwort `extends` kann bei einer Klassendeklaration die direkte Oberklasse festgelegt werden. Unterläßt man dies, wird automatisch `Object` zur Oberklasse.

Beispiel:

```
class Unterklasse extends Oberklasse {
  // hier ist Unterklasse direkt von "Oberklasse" abgeleitet
  // weiter mit Rest der Klassendeklaration ...
}

class Klasse {
  // Klasse wird direkt von Object abgeleitet
  // weiter mit Rest der Klassendeklaration ...
}
```

Die angegebene Oberklasse muß in der Deklarationsumgebung sichtbar sein und darf nicht `final` sein (siehe unten).

Durch sogenannte Interfaces besteht eine weitere Möglichkeit, Vererbung zu betreiben. Interfaces sind ähnliche Konstrukte wie Klassen und können an diese Eigenschaften vererben. Mehr dazu im nachfolgenden Artikel.

Eine Klasse erbt von ihrer direkten Oberklasse alle Felder, die nicht als `privat` deklariert sind (siehe 2.3.6) und keine Konstruktoren sind. Zu diesen ererbten Feldern kommen dann alle in der Klasse selbst deklarierten Felder hinzu. Alle diese Felder sind gleichzeitg innerhalb der Klasse sichtbar.

Eine Klassendeklaration darf durch sogenannte Modifikatoren eingeleitet werden. Dies sind allgemein Schlüsselwörter, die Eigenschaften von Programmelementen – hier also Klassen – festlegen:

- Eine als `abstract` deklarierte sogenannte abstrakte Klasse darf nur sogenannte abstrakte Methoden besitzen. Abstrakte Methoden müssen ebenfalls mit dem Modifikator `abstract` deklariert sein und ihr Rumpf darf nur aus einem Semikolon ; bestehen. Von abstrakten Klassen können keine Objekte erzeugt werden.

Oberklasse hat muß alle abstrakten Methoden der Oberklasse implementieren, d.h. sie muß die abstrakten Methoden durch nicht abstrakte also „gewöhnliche“ Methoden, die einen „richtigen“ Rumpf besitzen, überschreiben (zu Überschreiben siehe 2.3.7).

- Eine mit `final` deklarierte Klasse darf nicht als Oberklasse auftauchen, sie darf also hinter keinem `extends` stehen.
- Eine mit `public` vereinbarte Klasse kann in jedem anderen Package sichtbar gemacht werden. Innerhalb eines Packages darf höchsten eine mit `public` deklarierte Klasse auftauchen. Eine nicht mit `public` deklarierte Klasse ist automatisch ausschließlich im Package sichtbar, in dem sie deklariert ist.

2.3.5 Feldzugriffe auf Klassen

Mit Feldzugriffen werden Felder von Objekten einer Klasse gelesen (bzw. geschrieben bei einer Zuweisung). Dies kann auf verschiedene Weisen erfolgen:

- Ein Feldzugriff über ein Objekt hat die Form *Ausdruck*.*Feld*. Der *Ausdruck* muß dabei eine Referenz auf ein Objekt der Klasse liefern, in der das Feld vorkommt.
- Direkt über den Feldnamen kann man zugreifen, wenn das Feld innerhalb des aktuellen Sichtbarkeitbereichs liegt (z.B. innerhalb einer Methode in der Klasse, zu der das Feld gehört). Der Feldname darf dann nicht durch andere Variable überschattet sein (siehe 2.3.7).
- Ein Feldzugriff über eine Klasse hat die Form *Klassenname*.*Feld*. Dabei muß es sich entweder um ein Klassenfeld handeln oder das Feld muß im aktuellen Sichtbarkeitberich liegen. Auf diese Weise können auch überschattete Felder angesprochen werden.

2.3.6 Modifikatoren

Alle Felder einer Klasse dürfen durch die Modifikatoren `public`, `proctected` und `private` eingeleitet werden, um die Sichtbarkeit dieser Felder im Programm zu beeinflussen:

- Ein `public`-Feld ist überall sichtbar, wo auch die Klasse sichtbar ist.
- Ein `protected`-Feld ist innerhalb des Packages, zu dem die Klasse gehört und in jeder deren Unterklassen sichtbar.
- Ein `private`-Feld ist nur innerhalb der deklarierten Klasse sichtbar.
- Ein Feld ohne diese Modifikatoren ist ausschließlich in dem Package, in dem es deklariert wurde, dort aber überall sichtbar.

Ein Modifikator für Feldvariable darf zusätzlich eines der Schlüsselworte `static` oder `final` sein[4]:

[4] Auf zwei weitere sehr spezielle Modifikatoren (nämlich `transient` und `volatile`) möchte ich hier nicht eingehen.

Klasse, zu der diese Variable ein Feld ist, enthalten, sondern ist der Klasse selbst zugehörig und kommt dort genau einmal vor. Die Variable kann selbst dann angesprochen werden, wenn kein Objekt der Klasse instanziiert ist.

- Eine nicht mit `static` deklarierte Variable heißt Instanzvariable. Wann immer ein Objekt der entsprechenden Klasse erzeugt wird, umfasst dieses Objekt auch eine so deklarierte Variable.

- Eine mit `final` deklarierte Variable muß bei der Deklaration initialisiert werden und kann dann nicht mehr geändert werden. Solche Variablen verhalten sich wie Konstante.

Auch Methodendeklarationen dürfen besondere Modifikatoren vorausgehen:

- Eine mit `static` deklarierte Methode heißt Klassenmethode und ist wie eine Klassenvariable der Klasse selbst zugehörig. Sie kann auf Felder dieser Klasse nur zugreifen, falls diese ebenfalls mit `static` deklariert sind. Klassenmethoden müssen nicht über ein Objekt dieser Klasse aufgerufen werden.

- Eine nicht mit `static` deklarierte Methode heißt Instanzmethode und kann nur über ein Objekt der entsprechenden Klasse aufgerufen werden.

- Eine als `final` deklarierte Methode darf in Unterklassen nicht überschrieben werden (siehe auch 2.3.7).

- Eine als `abstract` deklarierte Methode hat keinen Rumpf (siehe 2.3.4).

2.3.7 Sichtbarkeit

Im Zusammenhang mit Vererbung und Polymorphie können verschiedene Sichtbarkeitprobleme auftreten:

Bei Namensgleichheit zwischen einer Feldvariable innerhalb der deklarierten Klasse und einer geerbten Variable aus einer Oberklasse wird die geerbte Variable von der Feldvariable überschattet: Zwar ist die geerbte Variable in der Klasse vorhanden, bei einem Zugriff direkt über den Variablennamen wird jedoch auf die Feldvariable der Klasse zugegriffen. Durch das Schlüsselwort `super` oder durch Verwendung des Namens der Oberklasse der überschatteten Variable kann dennoch auf deren Felder zugegriffen werden.

Beispiel:

```
class A {
  int x;
};

class B extends A {
  double x;

  void eineMethode() {
    x = x + 1.0;           // erhoeht x vom Typ double aus Klasse B
    A.x = A.x + 1;         // erhoeht x vom Typ int aus Klasse A
    super.x = super.x + 1; // erhoeht nochmal x aus Klasse A
```

```
};
```

Lokale Variablen (siehe 2.5.1), die zu einer Feldvariable namensgleich sind, überschatten die Feldvariable ebenfalls. Und zwar innerhalb eines Blocks ab der Stelle, wo die lokalen Variablen deklariert sind. In diesem Fall kann durch das Schlüsselwort `this` oder die Verwendung des Namens der Klasse der überschatteten Variable auf diese zugegriffen werden.

Innerhalb von Klassen dürfen namensgleiche Methoden mit verschieden großen Parameterlisten und/oder Parametertypen auftreten. Man spricht in diesem Zusammenhang von polymorphen Methoden. Beim Methodenaufruf wird dann zur Übersetzungzeit anhand der Argumentliste bestimmt werden, welche Methode gemeint ist.

Eine von einer Oberklasse geerbte Methode darf in der Unterklasse mit der gleichen Parameterliste bzgl. Größe und Typen neu deklariert werden. In diesem Fall spricht man vom Überschreiben der Methode. Die überschreibende Methode muß dabei jedoch folgende Eigenschaften haben:

- Sie muß mindestens so weit sichtbar sein wie die überschriebene Methode, d.h. wenn z. B. die überschriebene Mehtode `protected` ist, muß die überschreibende Methode `protected` oder `public` sein.
- Sie muß einen Resultattyp besitzen, der durch Zuweisung auf den der überschriebenen Methode per Zuweisungsumwandlung umgewandelt werden kann.
- Methoden, die in der Oberklasse mit `final` deklariert sind, dürfen nicht überschrieben werden.

Achtung: Private Methoden werden nicht vererbt!

2.3.8 Laden von Klassen

Eine Klasse wird bei Ausführung eines Programms normalerweise erst bei Bedarf geladen, wenn sie z. B. von einer anderen Klasse gebraucht wird. Dabei wird dann Speicherplatz für ihre Klassenvariablen bereitgestellt und ein Objekt der besonderen Klasse `Class` erzeugt. Dieses Objekt repräsentiert die geladene Klasse.

Klassenvariable können bei ihrer Deklaration oder durch einen Klasseninitialisator initialisiert werden. Ein Klasseninitialisator ist ein einfach ein Block von Anweisungen, der beim Laden der entsprechenden Klasse ausgeführt wird. Er wird durch das Schlüsselwort `static` eingeleitet. Alle statischen Initialisierungen werden beim Laden der Klasse sequentiell in der textuellen Reihenfolge ausgeführt.

Beispiel:

```
class Z {
  static int a = 1;
  static int b = 2;
  static { a++;      // dies ist ein Klasseninitialisator
           c = 7;}
  static int c = 2;
};
```

Wenn diese Klasse geladen wird, passiert folgendes: zuerst wird `a` auf `1` gesetzt, dann `b` auf `2`, `a` wird nun inkrementiert, `c` auf `7` gesetzt und schließlich wird `c` auf `2` gesetzt.

Ein Konstruktor ist im Prinzip eine besondere Methode zum Erzeugen neuer Objekte einer Klasse. Konstruktoren zu einer Klasse haben denselben Namen wie die Klasse selbst. Sie verhalten sich ähnlich wie gewöhnliche Methoden, abgesehen von einigen Ausnahmen:

- Konstruktoren haben keinen deklarierten Resultattyp.
- Enthält eine Klasse keinen explizit deklarierten Konstruktor, wird ein Default-Konstruktor erzeugt, der den Konstruktor der Oberklasse aufruft.
- Im Rumpf eines Konstruktors darf als erste Anweisung der Aufruf eines Konstruktors der direkten Oberklasse oder der Aufruf eines (anderen) Konstruktors derselben Klasse stehen. Dies wird mit Hilfe der Schlüsselwörter `super` bzw. `this` erreicht. Kommt eine solche Anweisung nicht vor, wird beim Aufruf des Konstruktors automatisch der Default-Konstruktor der Oberklasse aufgerufen. Danach werden sequentiell alle expliziten Initialisierungen in der Klasse vorgenommen.
- Der Aufruf erfolgt über den `new`-Operator (siehe 2.6).

Die weiteren Details möchte ich hier nicht ausführen. Stattdessen ein Beispiel:

```
class Linie {
  Punkt a, b;
  Farbe f = blau;
  int dicke = 1;
  Linie(Punkt pa, Punkt pb) {
    // hier impliziter Aufruf des Konstruktors der Oberklasse
    // also von Object
    // hier zuerst Initialisierung von f
    // dann Initialisierung von dicke
    a = pa;
    b = pb;
  }

  // hier der Defaultkonstruktor:
  Linie() {
    // er ruft explizit den anderen Konstruktor der Klasse
    // Linie auf
    this(Punkt(0,0),Punkt(0,0));
  }
}
```

2.3.10 Finalizer

Um das Löschen von Objekten muß sich in Java der Programmierer nicht selbst kümmern. Eine automatische Speicherbereinigung findet nach gewisser Zeit nicht mehr referenzierte Objekte und „räumt sie auf“, d.h. der belegte aber nicht mehr benötigte Speicherplatz wird freigegeben.

diese Methode auf, anstatt das betroffene Objekt dieser Klasse zu löschen. `finalize` wird jedoch höchstens einmal pro Objekt aufgerufen. Durch `finalize` können z. B. „verlorengegangene“ Objekte dem Java-Programm wieder zugänglich gemacht werden.

2.3.11 Typumwandlungen auf Referenztypen

Auch bei Umwandlungen auf Referenztypen gibt es Casting und Zuweisungsumwandlung aber keine aber keine arithmetische Umwandlung, da auf Referenztypen keine entsprechenden Operatoren definiert sind.

Eine Zuweisungsumwandlung auf Referenztypen ist möglich wenn der Zieltyp zu einer Oberklasse der Klasse des Ausgangstyps gehört. So kann z. B. jeder Referenztyp einer Klasse nach `Object` (als Referenztyp) gewandelt werden, da `Object` zu jeder Klasse eine Oberklasse ist.

Bei Castings darf zudem in die andere Richtung, d.h. von der Oberklasse zur Unterklasse gewandelt werden. Dies kann jedoch Laufzeitüberprüfungen erfordern: Besitzt etwa eine Oberklasse zwei Unterklassen, so könnte ein Objekt, das durch ein Casting vom Typ der Oberklasse auf den Typ der ersten Unterklasse gewandelt wird, vom (speziellen) Typ der zweiten Unterklasse sein. Damit das Objekt dann nicht falsch typisiert wird, löst die Laufzeitüberprüfung in solchen Fällen eine Exception aus.

Die Syntax eines Castings auf Referenztypen ist analog zu der auf primitiven Typen.

Zwischen primitiven Typen und Referenztypen sind keine Umwandlungen möglich.

Auch für Interfaces gibt es Referenztypen. Umwandlungen auf Referenztypen zwischen Klassen und Interfaces sind unter bestimmten Bedingungen möglich. Siehe hierzu nachfolgenden Artikel.

2.4 Strings und Arrays

2.4.1 Strings

Strings sind in Java Objekte der Klasse `String`, einer direkten Unterklasse von `Object`. Für `String` stehen Literale zur Erzeugung von Strings und der Konkatenationsoperator `+` zur Verfügung. `+` verkettet zwei String-Objekte zu einem neuen String-Objekt. Falls (genau) eines seiner Argumente kein String ist, wird dieses zuvor in seine entsprechende String-Darstellung konvertiert.

String-Literale werden durch Gänsefüßchen " begrenzt und bestehen aus einzelnen Unicode-Zeichen. Innerhalb von String-Literalen sind, ähnlich wie bei `char`, Escape-Sequenzen anwendbar, um bestimmte Unicode-Zeichen zu erzeugen.

Beispiele für String-Literale:

```
""    // der leere String
"\""  // ein String der nur " enthaelt
"Ein ganz normaler String."
"Ein konkatenierter " + "String."
```

Achtung: String-Literale werden nicht wie in C implizit durch das Nullzeichen `NUL` terminiert, und sind inbesondere keine Arrays aus `char`s.

Arrays sind in Java Objekte der Klasse `Array`, einer direkten Unterklasse von `Object`. Sie sind eindimensional, jedoch können durch Verschachtelung mehrdimensionale Arrays simuliert werden. Im Prinzip ist ein Array-Objekt eine Anzahl von Variablen gleichen Typs, die durch einen nicht-negativen Integer-Wert angesprochen werden.

Eine Variable vom Typ Array wird durch Angabe des Komponententyps des Arrays gefolgt von eckigen Klammern gefolgt vom Variablennamen deklariert.[5] Damit wird zwar der Typ der Variablen festgelegt aber noch kein Speicher allokiert.

Beispiele:

```
int[] ai;      // Array aus int's
short[][] as;  // Array aus Arrays aus shorts's
Object[] ao;   // Array aus Referenzen auf Object der Klasse Object
```

Offensichtlich ist die Größe eines Arrays durch den entsprechenden Variablentyp nicht festgelegt.

Auch Variablen vom Typ Array können bei der Deklaration intialisiert werden. Ein Array-Initialisator hat dabei eine ähnliche Funktion wie Literale primitiver Typen. Er besteht aus geschweiften Klammern mit durch Komma getrennten Array-Komponenten als Elementen.

Beispiel:

```
int[] quadrat = { 1, 4, 9, 16, 25, 36, 49 };
char[] ac = { 'k', 'e', 'i', 'n', ' ', 'S', 'r', 'i', 'n', 'g' };
```

Durch den `new`-Operator kann – wie auch für Objekte anderer Klassen – Speicher für ein Array allokiert werden, wobei die Göße des anzufordernden Arrays in eckigen Klammern angegeben sein muß. Die Werte der Array-Komponenten sind dann einfach der Default-Wert des entsprechenden Komponententyps.

Beispiele:

```
Object[] ao = new Object[10]
// initialisiert eine Variable vom Typ
// Array auf Object mit Array aus 10 Referenzen

int[][] matrix1 = new int[2][3]
// ein Variable vom Typ Array aus Arrays aus int's
// initialisiert mit einem Array aus genau zwei Komponenten,
// die jeweils Arrays aus int's mit drei Komponenten referenzieren

int[][] matrix2 = new int[5][]
// eine Variable vom Typ Array aus Arrays aus int's
// initialisiert mit einem Array aus genau fuenf Komponenten
```

Das mittlere Beispiel zeigt, daß durch `new` bei verschachtelten Arrays die „Komponenten-Arrays“ gleich mit allokiert werden können. Es muß zumindest in der Klammer ganz links eine Array-Größe eingetragen sein und dann optional in den *direkt* darauffolgenden Klammern für die Komponenten-Arrays.

[5] Eine Deklarationsvariante die syntaktisch mehr an C orientiert ist, semantisch jedoch analog zur vorigen ist, möchte ich hier vernachlässigen.

die „äußeren“ Arrays.

Die Größe eines Arrays kann im Programm durch *Array*`.length` ermittelt werden. Sie ist fest bezüglich des allokierten Array-Objekts.

Eine Array-Komponente wird durch Angabe der Array-Variable gefolgt von eckigen Klammern, die den Komponentenindex enthalten, angesprochen. Der Index muß ein Integer-Wert zwischen 0 und der entsprechenden Array-Länge `.length-1` sein, sonst wird ein Exception ausgelöst. Falls die Variable den Wert `null` hat, wird ebenfalls eine Exception ausgelöst.

Beispiele:

```
a[3] = 4;   // die 4. Komponente eines durch die Arrayvariable a
            // referenzierten Arrays erhaelt den Wert 4
x = a[0][2] // x wird der Wert der Komponente 3 eines Arrays, das die
            // Komponente 0 eines Arrays ist, auf das a verweist,
            // zugewiesen
```

2.5 Blöcke und Anweisungen

Blöcke und Anweisungen sind im Rumpf von Methoden, Konstruktoren und bei statischen Initialisierungen (siehe 2.3.8) anwendbar. Die Syntax und Semantik ist dabei größtenteils analog zu der in C. Ein Block beginnt und endet, wie schon gesehen durch geschweifte Klammern `{}`.

Achtung: Unerreichbare Anweisungen, also solche, von denen der Übersetzer nachweisen kann, daß sie niemals ausgeführt werden, sind verboten.

2.5.1 Lokale Variablen

Innerhalb eines Blocks dürfen an beliebiger Stellen lokale Variablen deklariert werden. Die Lebensdauer lokaler Variable reicht von der Stelle ihrer Deklaration bis zum Ende des direkt umgebenden Blocks. Falls die lokale Variable namensgleich zu einer sichtbaren Feldvariable ist, wird die Feldvariable durch die lokale Variable überschattet (siehe 2.3.7).

2.5.2 Anweisungen

Anweisungen stehen innerhalb von Blöcken und werden durch ein Semikolon ; abgeschlossen. Als Anweisungen dürfen

- markierte Anweisungen,
- fast beliebige Ausdrücke,
- die leere Anweisung (nur ein Semikolon ;),
- Kontrollflußkonstrukte und Sprunganweisungen,
- Synchronisationsanweisungen und
- Exception-Anweisungen

Markierte Anweisungen sind Anweisungen, denen eine Marke (das ist ein gewöhnlicher Bezeichner gefolgt von einem Doppelpunkt :) vorausgeht. Die Marke dient entweder als eine Art Sprungziel bei Sprunganweisungen oder als Auswahlstelle bei der `switch`-Anweisung.

2.5.3 Kontrollflußkonstrukte

Die Kontrollflußkonstrukte sind in Syntax und Semantik ähnlich wie in C definiert. Ich möchte hier nur auf interessante Unterschiede eingehen.

Die `if`-Anweisung ist wie in C verwendbar. Der Bedingungsausdruck nach `if` muß jedoch immer einen Wert vom Typ `boolean` liefern.

Die `switch`-Anweisung erwartet im Bedingungsausdruck einen Wert vom Typ `char`, `byte`, `short` oder `int`. Im Auswahlteil dürfen höchstens eine `default`-Marke und Null oder mehr `case`-Marken auftauchen. Hinter den Marken wird ein konstanter Integer-Wert erwartet.

Die `while`-, `do`- und `for`-Anweisung erwarten im entspechenden Bedingungsausdruck ebenfalls einen `boolean`-Wert und werden ansonsten fast wie in C behandelt. Nur bei der `for`-Anweisung existiert ein kleiner Unterschied: Obwohl es in Java den Kommaoperator (wie in C) nicht gibt, dürfen im Initialisierungs- und Inkrementierungsteil von `for`-Schleifen mehrere durch Komma getrennte Ausdrücke angegeben werden. Damit bietet sich die Möglichkeit `for`-Schleifen auch ohne Kommaoperator so zu schreiben, wie man es von C her gewohnt ist.

2.5.4 Sprunganweisungen

Auch in Java gibt es `break`, `continue` und `return` allerdings *kein* `goto`. Bei den Sprunganweisungen sind die Unterschiede zu C jedoch etwas größer als bei den Kontrollflußkonstrukten:

Der `break`-Anweisung darf optional eine Marke folgen. Wird diese weggelassen, funtkioniert `break` wie in C. Ist eine Marke angegeben, so muß eine umgebende Anweisung eine markierte Anweisung mit genau dieser Marke sein. Das Programm wird dann nach der letzten Anweisung fortgeführt, die noch zu der umgebenden Anweisung gehört.

Beispiel:

```
test:
if (b == 0) {
  while(a < 100) {
    a++;
    if (check(a))
      break test;
  }
}
// hier geht's nach einem break weiter
```

Die `continue`-Anweisung darf nur innerhalb von `do`-, `while`- und `for`- Anweisungen erscheinen. Auch hier darf wie bei `break` optional eine Marke angegeben werden. Ohne Marke wird durch `continue` am Ende der unmittelbar umgebenden Schleife fortgefahren. Ist eine Marke angegeben, so muß auch eine umgebende Schleife existieren, vor der diese Marke steht. Dann wird durch `continue` ganz am Ende dieser Schleife fortgefahren.

Beispiel:

```
while(1) {
  for(;;) {
    ... continue outer; // hier kommt irgendwo diese Anweisung
  }
  ...
  // hier wird nach continue outer; fortgefahren
}
```

Die `return` Anweisung funktioniert wie in C. Sie wurde auch in 2.3.3 schon angesprochen.

Im Zusammenhang mit Exception-Anweisungen können sich die Sprunganweisungen noch etwas anders als hier erläutert verhalten. Siehe dazu nachfolgenden Artikel.

2.6 Ausdrücke

Ein Ausdruck kann in Java drei Resultate liefern: Eine Variable, einen Wert oder nichts (d. h. `void`). `void` wird beim Aufruf einer Methode mit Resultattyp `void` geliefert.

2.6.1 Der Typ eines Ausdrucks

Jeder Ausdruck hat einen Übersetzungszeittyp. Der Typ eines zur Laufzeit berechneten Werts des Ausdrucks ist immer ein Untertyp zum Übersetzungszeittyp: Ist etwa der Übersetzungszeittyp ein Klassentyp, dann ist der Wert des Ausdrucks `null` oder eine Referenz auf ein Objekt der entsprechenden (Unter-)Klasse. Ist der Übersetzungszeittyp ein primitver Typ, dann hat der Wert denselben primitiven Typ.

In einigen Fällen hat der Laufzeittyp eines Ausdrucks Auswirkungen, die nicht zur Übersetzungszeit bekannt sein können:

- Bei einem Methoden- oder Konstruktoraufruf kann evtl. zur Übersetzungzeit nicht bestimmt werden, welche Methode genau aufzurufen ist. Dies tritt im Zusammenhang mit Überschreiben von Methoden auf. Zur Laufzeit wird dann die für den jeweiligen Objekttyp spezifischste Methode aufgerufen.
- Bei einer Umwandlung von einem Oberklassentyp auf einen Unterklassentyp muß der Laufzeittyp des jeweiligen Objektes herangezogen werden.
- Beim `instanceof`-Operator (siehe 2.6.6) und noch in einigen weiteren (weniger interessanten) Fällen ist dies auch der Fall.

2.6.2 Auswertungsreihenfolge

Die Auswertungsreihenfolge ist im Gegensatz zu C auf allen Ausdrücken genau festgelegt. Es wird im Prinzip immer von links nach rechts ausgewertet:

- Bei binären Operatoren wird erst der linke dann der rechte Operand ausgewertet. Seiteneffekte aus dem linken Operanden sind im rechten sichtbar.
- Bei einem Array-Zugriff wird die Array-Referenz vor dem Indexausdruck in den eckigen Klammern berechnet. Das ist z. B. kritisch bei dem Ausdruck `a[(a=b)[3]]`.

(links vom Punkt) berechnet und dann erst der Methodenaufruf (rechts vom Punkt).

- In einem Methodenaufruf werden die Argumentausdrücke von links nach rechts berechnet.
- Bei einer mehrdimensionalen Array-Allokierung werden zuerst die äußeren Array, zu denen die Klammern weiter rechts gehören, allokiert.

2.6.3 this und super

`this` und `super` dürfen nur in Instanzmethoden und Konstruktoren verwendet werden. Sie haben im Prinzip die Funktion von Variablen, die auf das Objekt verweisen, über das die entsprechende Methode bzw. der Konstruktor aufgerufen wurde.

Bei `this` ist der Übersetzungszeittyp der Typ der entsprechenden Klasse des Objekts, bei `super` hingegen der Typ der direkten Oberklasse. Dadurch kann mit Hilfe von `super` etwa auf eine überschriebene Methode aus einer Oberklasse zugegriffen werden. `super` darf nicht in der Klasse `Object` verwendet werden.

Bei Konstruktoraufrufen kommt `super` noch eine besondere Bedeutung zu (siehe 2.3.9).

2.6.4 Methodenaufrufe

Eine Methode wird über ihren Namen gefolgt von der Liste ihrer Argumente wie in C aufgerufen. Die Liste kann auch leer sein (nur geschweifte Klammern). Damit der Methodenaufruf korrekt ist, muß unter anderem folgendes gelten:

- Der entsprechende Methodenname muß im aktuellen Sichtbarkeitsbereich liegen, d.h. die Methodendeklaration muß sichtbar sein.
- Die Zahl der Argumente muß mit der Zahl der Parameter in der Deklaration übereinstimmen.
- Die Argumenttypen müssen per Zuweisungsumwandlung auf die entsprechenden Parametertypen umgewandelt werden können.

Aufgrund der möglichen Polymorphie von Methoden ist es unter Umständen kompliziert zur Übersetzungszeit die aufzurufende Methode zu bestimmen, da mehrere Methoden zu einem Aufruf passen könnten. Grundsätzlich wird dann diejenige Methode ausgewählt die am spezifischsten auf die Übersetzungszeittypen der Argumente paßt. Eine Methode ist in diesem Fall spezifischer als eine andere, wenn alle ihre Parametertypen durch Zuweisungsumwandlung auf die Parametertypen der anderen Methode gewandelt werden können. Falls es keine spezifischste Methode gibt, ist diese Mehrdeutigkeit ein Fehler.

Im Zusammenhang mit Überschreiben können ebenfalls Konflikte in Bezug auf die aufzurufende Methode entstehen: Wird eine Methode (implizit oder explizit) über ein Objekt aufgerufen und ist die Methode überschrieben worden, so kann es evtl. notwendig sein zur Laufzeit an Hand des Objekttyps zu entscheiden, welche Methode nun genommen wird. Java entscheidet sich dann für die erste passende Methode, die erscheint, wenn man den Vererbungsbaum von der Klasse des Objekts nach oben (d.h. zu `Object` hin) durchläuft.

Mit dem `new`-Operator kann man Objekte eines bestimmten Klassentyps anfordern. Das geht aber nur, wenn es sich nicht um abstrakte Klassen handelt. Hierzu schreibt man `new` gefolgt von einem entsprechenden Konstruktoraufruf. Bei polymorphen Konstruktoren erfolgt die Auswahl des passenden wie im Fall von Methoden. Die Allokierung von Arrays wurde schon in 2.4.2 betrachtet.

2.6.6 Der instanceof-Operator

Die meisten Operatoren wurden in 2.2 schon angesprochen. Wenn man die analogen Operatoren von C her kennt, wird dieser Überblick genügen.

Interessant ist allerdings der binäre `instanceof`-Operator: Sein linker Operand muß ein Ausdruck mit Klassen- (oder Interface-) Typ sein, der rechte Operand der Name eines Referenztyps. `instanceof` liefert `false`, wenn der Wert der linken Operands `null` ist. Der Operand liefert `true`, wenn der Laufzeittyp des linken Openands auf den rechten Operand (der ein Typ ist) durch Casting gewandelt werden kann, ansonsten `false`.

Seminarbeitrag von **Matthias Winkel**

Jede Programmiersprache hat ihre besonderen Eigenschaften und Features, die ihren Charakter ausmachen. Packages, Interfaces und Exceptions sind in der gleichen Weise typische Merkmale der Programmiersprache Java, wie es Zeiger und der fast völlig freizügige Umgang mit ihnen bei C und C++ sind.

Packages und Interfaces sind Eigenschaften, die den Programmierer vor allem in der Entwicklungsphase unterstützen. Packages praktizieren das Prinzip der Kapselung und sorgen für eine strukturierte Verwaltung von Klassen und für ein einheitliches Verfahren, um auf diese zuzugreifen. Interfaces ermöglichen die getrennte Betrachtung von Design und Implementierung.
Exceptions erfüllen, konsequent angewandt, ein Programm mit Robustheit. Gleichzeitig stellen sie ein neues Prinzip zur Behandlung von Ausnahmen dar, welches den Programmtext übersichtlicher gestaltet, und zwar durch die Trennung des Codes zur Fehlerbehandlung vom eigentlichen Programmcode.

Durch die Möglichkeit, Nativen Code in ein Java-Programm einzufügen, können unter anderem spezielle Routinen des jeweiligen Betriebssystems nutzbar gemacht und zeitkritische Funktionen realisiert werden, welche der ohnehin schon mächtigen Klassenbibliothek nahezu unbegrenzte Möglichkeiten in Aussicht stellen.

3.1 Packages

Packages sind das oberste Element in der Programmgliederungshierarchie. Im Wesentlichen erfüllen sie zwei Funktionen. Zum einen fassen sie mehrere Klassen nach Gesichtspunkten, die der Programmierer festlegt, zu Paketen zusammen. Die Namen der Pakete spiegeln dabei die Orte wieder, wo diese aufzufinden sind. Zum anderen sind sie ein grundlegender Bestandteil des Prinzips der Kapselung in Java.

3.1.1 Hinführung

Eines der wichtigsten Prinzipien der objektorientierten Programmierung ist die Wiederverwendung von Klassen. Dies setzt einerseits programmiertechnische Bedingungen voraus, wie zum Beispiel die Klassen ausreichend allgemein zu halten. Andererseits werden gewisse Voraussetzungen an die Organisation geknüpft, so müssen unter anderem Klassen auffindbar, ihre Namen eindeutig und die Zusammengehörigkeit verschiedener Klassen von außen sichtbar sein. Das Konzept, Klassen in Pakete zusammenzufassen, bedeutet für die organisatorischen Aufgaben eine einheitliche und systematische Lösung.

Java-Programms über ein Netzwerk verteilt sein können, benötigt man ein Verfahren zu ihrer Auffindung. Dies geht so weit, daß eine globale Übereinkunft über die Namensvergabe getroffen werden muß, damit auf jede Klasse mit einem eindeutigen Bezeichner zugegriffen werden kann.

Mit zunehmender Entwicklung von Java-Programmen wächst auch die Anzahl der verwendeten Klassen. Oftmals vergibt der Programmierer kurze einfache Namen, was jedoch dazu führen kann, daß diese Namen mit denen anderer Klassen in Konflikt geraten. Dieser Fall kann zum Beispiel bei Klassen eintreten, die zu einem früheren Zeitpunkt oder von anderen Programmierern geschrieben wurden, und von denen der Anwender vergessen hat oder nicht einmal weiß, daß sie möglicherweise gleiche Namen verwenden. Hier ist es also angebracht, Klassen zu kapseln und in Paketen zu verbergen.

3.1.2 Namensvergabe und Auffinden von Paketen

Durch das Konzept der Packages werden also mehrere Klassen zu einer Einheit zusammengefaßt; sie werden zu einem Paket unter einem Namen vereinigt. Dieser Paketname besteht in der Regel aus mehreren durch Punkte getrennten Bezeichnern.
Die Pakete selbst werden meist in Analogie zum Verzeichnisbaum eines Datenträgers ähnlich hierarchisch angeordnet. Der erste, ganz links im Paketname stehende Bezeichner stellt dann eine sehr grobe und sehr allgemeine Klassen-Gruppierung dar. Jeder weitere Bezeichner, der im Paketnamen vorhanden ist, repräsentiert eine kleinere spezifischere Gruppe von Klassen.
Die beim JDK[6] mittglieferte Klassenbibliothek ist in dieser Weise strukturiert. Die oberste Ebene trägt den Bezeichner `java`. Die nächste Ebene wird durch Namen wie `awt`, `io` oder `util` bezeichnet. `Awt` ist durch den Bezeichner `image` ein weiteres Mal untergliedert. Der Paketname lautet also volständig `java.awt.image`, und darin finden wir Klassen für Bildverarbeitung in einer Windowsoberfläche.
Durch Anhängen des Namens der gewünschten Klasse in diesem Paket mittels eines Punktes an den Paketnamen kann nun an jeder Stelle im Java-Source-File diese Klasse referenziert werden. Beispielsweise für den Zugriff auf die Klasse `ImageFilter` im Paket `java.awt.image` lautet der Ausdruck `java.awt.image.ImageFilter`.

Die Paketnamen stehen im direkten Zusammenhang mit dem Ort, an dem die Pakete und die darin enthaltenen Klassen abgelegt wurden. Für gewöhnlich enthält jedes Java-Source-File genau eine Klassenbeschreibung[7]. Wenn dem so ist, dann entspricht die hierarchische Anordnung der Pakete genau der Gruppierung von Dateien in einem Verzeichnisbaum. Und tatsächlich erwartet der Java-Compiler innerhalb des Klassenverzeichnisses[8] einen Verzeichnisbaum, der genau die Hierarchie der Pakete widerspiegelt. Die Klassen eines Paketes müssen innerhalb dieser Verzeichnisstrukur so abgelegt werden, daß der Name des Verzeichnisses genau dem Namen des Paketes entspricht.
Als Beispiel soll die mitgelieferte Java-Klassenbibliothek dienen, deren Verzeichnishierarchie exakt die Pakethierarchie widerspiegelt. Im Klassenverzeichnis existiert ein Unterverzeichnis `java`. Dieses enthält weitere Verzeichnisse wie zum Beispiel `awt`, `io`, `util`. Jeder durch Punkte abgetrennte Namensteil eines Paketnamens repräsentiert somit einen Knoten des Verzeichnisbaumes. Unter dem Betriebssystem UNIX ist die Klasse, die durch `java.awt.image.ImageFilter`

[6] JDK ist eine gängige Abkürzung für das Java-Development-Kit von Sun.

[7] Tatsächlich können auch unter gewissen Bedingungen mehrere Klassen in einem Source-File beschrieben werden.

[8] Es handelt sich hierbei um eines der in der Umgebungsvariablen `classpath` festgelegten Verzeichnisse. Meist hat es den Namen `.../java/classes/`.

der Pfadname einzusetzen, unter dem die Klassenbibliothek installiert wurde. Liegen die Klassen also beispielsweise unter dem Verzeichnis `/usr/java/classes`, so ist die Klasse `ImageFilter` unter dem Pfad `/usr/java/classes/java/awt/image/ImageFilter.class` aufzufinden.
Anders formuliert: Der Java-Compiler ersetzt in einer Referenz auf eine Klasse alle Punkte durch das Trennungssymbol des verwendeten Filesystems, unter UNIX also durch den Schrägstrich, dadurch erhält er direkt den Dateinamen zusammen mit dem Pfadnamen der entsprechenden Klasse.

Java ist als Programmiersprache zur Verwendung in einem Netzwerk konzipiert worden. Eines der damit verbundenen Prinzipien ist, daß Programme und Programmteile nicht alle lokal auf einem Rechner vorhanden sein müssen, sondern über das Netzwerk verteilt liegen können.
Deshalb ist die richtige Vergabe von Paketnamen hinsichtlich zweier Aspekte wichtig. Einerseits legt der Paketname fest, wo das Paket in der Verzeichnisstruktur zu finden ist, wie wir gerade gesehen haben. Andererseits muß der Programmierer darauf achten, daß er eindeutige Namen wählt, wenn er selbstgeschriebene Programme und Klassen weitergeben oder über das Netzwerk zur Verfügung stellen möchte. Es ist also ein globales Verfahren nötig, das zum einen das Auffinden von Klassen nach einem einheitlichn Muster ermöglicht, und zum anderen die Eindeutigkeit bei der Namensvergabe für neuhinzukommende Klassen gewährleistet.
Sun hat aus diesen Gründen eine Konvention für die Vergabe von Namen eingeführt, welche die Tatsache ausnutzt, daß Domainnamen im Internet weltweit eindeutig sind. In den zu veregebenden Namen fließt also der Domainname des Rechners mit ein, auf dem die Klasse abgelegt wird. Der Name für das an oberster Stelle in der Hierarchie stehende Package ist reserviert für das in Großbuchstaben geschriebene „Länderkürzel“, zum Beispiel DE, FR, US, EDU, COM, etc. Dieses bildet den ersten Teil des Namens. An ihn schließen sich die restlichen Teile des Domainnamens in umgekehrter Reihenfolge an. Danach folgt die Verzeichnisstruktur im bisherigen Sinne.
Ein Programmierer der Firma Sun würde die Namen der Pakete, in denen er seine gerade entwickelten Klassen ablegen möchte, mit dem Ausdruck `COM.sunsoft.sun` beginnen lassen.
Durch die Voranstellung der Internetadresse in umgekehrter Reihenfolge wird gewährleistet, daß der Paketname für das Internet weltweit eindeutig ist.[9]

3.1.3 Definition von Packages

Ein Paket wird durch das Schlüsselwort `package` im Quellcode kreiert.
Die Syntax sieht wie folgt aus:

package *Paketname* ;

wobei

Paketname:

Bezeichner
Paketname . Bezeichner

Wenn der Package-Befehl in einem Java-Quelltext verwendet wird, so muß er am Anfang der Datei stehen und zwar noch vor der Klassendefinition. Lediglich Kommentare und White Spaces dürfen vor dem Package-Kommando auftauchen.

[9]Befindet sich der Rechner zusätzlich in einem lokalen Netzwerk, ist er zum Beispiel Teil des örtlichen Universitätsnetzwerks, muß der zuständige Administrator dafür Sorge tragen, daß die Eindeutigkeit der Rechneradresse innerhalb dieses Netzwerkes erhalten bleibt.

angegebenen Packetes erklärt. Der Java-Interpreter erwartet, den übersetzten Bytecode unter dem durch den Paktenamen repräsentierten Verzeichnis aufzufinden.

Beispiel 1:

```
package TopPackage.SecondLevelPackage.ThirdLevelPackage;

public class AClass extends BClass
{ ... }
```

Die Klasse AClass erweitert BClass. Damit der Java-Übersetzer den Quelltext (`AClass.java`) findet, muß dieser sich im Verzeichnis `.../TopPackage/SecondLevelPackage/ThirdLevel-Package/` befinden. Der Übersetzer legt den erzeugten Bytecode im selben Verzeichnis unter dem Dateinamen `AClass.class` ab, so daß der Interpreter die Datei finden kann. Sowohl der Übersetzer als auch der Interpreter erzwingen und erwarten diese Hierarchie.

Da das Package-Kommando optional ist, soll an dieser Stelle erläutert werden, was es bedeutet, wenn `package` nicht im Quelltext auftaucht. In diesem Fall wird die Klasse einem standardmäßigen, namenlosen Paket hinzugefügt. Das übersetzt File mit der Endung `.class` wird im aktuellen Verzeichnis abgelegt. Wenn auf solch eine Klasse referenziert wird, dann sucht der Java-Compiler diese in sämtlichen Verzeichnissen, die in der Umgebungsvariablen `classpath` eingetragen sind; meist sind dies das aktuelle und das `/java/classes`-Verzeichnis.

3.1.4 Importieren von Klassen und Paketen

Durch die Hinzunahme von Klassen aus anderen Paketen, muß es für den Programmierer eine Möglichkeit geben, externe Klassen- und Paketnamen im eigenen Programm sichtbar zu machen. Java unterstützt keine Include-Dateien. Falls der Java-Übersetzer Informationen benötigt, liest er diese direkt aus bereits übersetzten Klassendateien und Packages. Dem Übersetzer muß lediglich mitgeteilt werden, wo er die Definition eines verwendeten Klassentyps findet.
Eine Möglichkeit ist es, den Klassennamen immer in Verbindung mit dem Namen des Paketes, in dem sich die Klasse befindet, anzugeben. Dies ist sogar die zu empfehlende Methode, falls die Klasse nicht all zu oft im Programmtext vorkommt. Allerdings kann der Referenzausdruck recht lang werden, wenn die entsprechende Klasse sich mehrere Ebenen tief in einer Pakethierarchie befindet. Eine Referenz wie zum Beispiel `project.one.windows.utils.MessageBox` jedesmal vollständig auszuschreiben, ist recht mühsam.
Hier schafft das Import-Konzept von Java Abhilfe. Am Anfang eines Programmtextes werden durch eine Reihe von Import-Anweisungen Klassendefinitionen und Pakete für den Übersetzer sichtbar gemacht.

Die Syntax des Import-Kommandos sieht folgendermaßen aus:

`import` *Paketname* ; oder
`import` *Paketname* . *Klassenname* ; oder
`import` *Paketname* . * ;

Im ersten Fall wird das angegebene Paket unter der Bezeichnung seiner letzten Namenskomponente sichtbar gemacht. Die Zeile `import java.io;` führt dazu, daß auf die Klassentypen in `java.io` mittels `io.`*name* zugegriffen werden kann, wobei *name* der Name eines Klassentypes oder eines Paketes innerhalb von `java.io` ist.

`java.util.Vector;` wird die Klasse `Vector` im Paket `java.util` durch `Vector` referenziert, genau so wie der Angabe `java.util.Vector` ohne die Verwendung von `import .`
Der dritte Fall macht durch den Stern alle Namen des angegebenen Pakets sichtbar, wenn sie gebraucht werden. Das gleiche Ergebnis erzielt man, allerdings mit erheblich mehr Schreibaufwand, wenn für sämtliche Namen innerhalb des Paketes jeweils eine Importbefehlszeile, die der zweiten Syntaxvariante entspricht, in den Programmtext eingefügt wird.
Der Stern darf allerdings nur hinter einem Punkt stehen, und hat keineswegs die gleiche Funktion, wie der gleichaussehende Stern des UNIX-Filesystems, der zum Abkürzen von Dateinamen verwendet wird. Es ist nicht möglich, zum Beispiel sämtliche Namen innerhalb eines Paketes, die mit dem Buchstaben `A` beginnen, sichtbar zu machen, etwa durch durch eine Angabe wie `java.util.A*`.

Stößt Java auf einen Namen, den es nicht im eigenen Programm auflösen[10] kann, so steht der Name für eine Klasse innerhalb eines Paketes, das auf eine der drei Arten importiert wurde. Kann es den Namen dort auch nicht finden, so wird eine Fehlermeldung ausgegeben. Wenn mehrere Pakete mit gleichem Namen vorliegen, so geht Java beim Auflösen der Namen von innen nach außen vor. Das bedeutet, daß zuerst das eigene Programm, dann die lokal auf dem Rechner befindlichen Pakete und dann Pakete von außen abgearbeitet werden. Dies hat sicherheitstechnische Gründe. So ist es zum Beispiel nicht möglich, irgendwelche fragwürdigen Pakete heimlich von außen unterzujubeln, oder gar vorhandene Pakete durch externe Packages zu ersetzen.

Die Klassen des Paktes `java.lang` werden automatisch in jeder Übersetzungseinheit sichtbar gemacht und müssen also nicht importiert werden. Dies liegt daran, daß dieses Paket Klassen mit absolut grundlegenden Funktionen und Definitionen enthält. Hier wird zum Beispiel die Klasse `String` definiert.

3.2 Interfaces

Interfaces beschreiben Typen, die aus Methodensignaturen und Konstanten bestehen, ohne ihre Implementierung anzugeben. Diese Typen werden von Klassen ausgefüllt, indem die geforderten Methodensignaturen mit Funktionalität ergänzt werden. Interfaces, oder zu deutsch Schnittstellen, sind im Wesentlichen abstrakte Klassen.

Interfaces können auch einfach als Gruppierungen von Funktionen aufgefaßt werden, von denen sichergestellt ist, daß die Klassen, die Schnittstellen implementieren, mindestens die Funktionalität für eben diese Funktionen bereitstellen.

Interfaces enthalten im Gegensatz zu abstrakten Klassen überhaupt keine implementierten Methoden; alle Methoden existieren nur als Funktionssignaturen. Eine Schnittstelle legt somit eine Menge von Methoden fest, die bei einem Objekt angwendet werden können.
Abstrakte Klassen dagegen enthalten zum Teil bereits implementierte Methoden, um den von ihnen abgeleiteten Klassen ein gewisses Grundverhalten mitzugeben.

3.2.1 Sinn und Zweck von Interfaces

Bekanntermaßen befolgt Java das Prinzip der Einfachvererbung. Dies bedeutet, daß jede abgeleitete Klasse genau eine übergeordnete Klasse besitzt. Der Graph der abgeleiteten Klassen bildet folglich einen Baum.

[10] Unter *Auflösen* versteht man die Erkennung der Zugehörigkeit einer Klasse zu einem Namen.

die Klasse der rationale Zahlen zu der Klasse „Float“ führt, lassen sich in diesem Baum problemlos darstellen. Schwieriger wird es bei komplexeren Entwicklungen, wenn zwischen Teilen des Baumes Querbeziehungen notwendig werden. Hier kann die zur Gewohnheit gewordene Disziplin der hierarchischen Denkweise und jede zum Programmdesign beitragende Idee, jweils nur an einer Stelle in den Baum einzufügen, als äußerst restriktiv empfunden werden.
Klassen, die zu einem späteren Zeitpunkt entwickelt wurden, können nachträglich nicht ohne Weiteres in den Vererbungsbaum eingefügt werden. Dies führt dazu, daß Klassen, die dem Anwender als Schnittstellen zur Verfügung gestellt werden sollen, relativ früh und mit Vorausblick auf das Endprodukt konzipiert werden müssen. Dieses implementationsnahe Entwickeln des Programmdesigns widerspricht aber der objektorientierten Softwaretechnik. Eines der wichtigsten Ergebnisse der objektorientierten Softwareentwicklung ist die klare Trennung zwischen Programmdesign und Programmimplementierung.
Andere objektorientierten Programmiersprachen lösen diese Restriktion durch Einführung von komplexeren Vererbungshierarchien auf, wie zum Beispiel durch die Möglichkeit der Mehrfachvererbung, wobei allerdings neue Konflikte und mehr Fehler in Kauf genommen werden. Hinsichtlich dieser größeren Anzahl von Inkonsistenzen erlaubt Java ein solches Vererbungssystem nicht. Das Konzept der Interfaces soll hier weiterhelfen.

3.2.2 Konzept der Interfaces

Das neueingeführte Konzept ist eine Hierarchie von Interfaces. Diese Hierarchie ist nicht an das Einfachvererbungsprinzip der Klassen gebunden. Jede Schnittstelle kann von mehreren Schnittstellen abgeleitet sein, ebenso kann jede Klasse mehrere Schnittstellen implementieren, unabhängig von der Klasse, die sie erweitert. Dadurch wird dem Programmierer eine Art Mehrfachvererbung zur Verfügung gestellt, die aber bei weitem nicht die gleiche Freizügigkeit der Mehrfachvererbung im üblichen Sinne aufweist. Dementsprechend gering sind auch die dadurch neu auftretenden Konflikte.

3.2.3 Definition von Interfaces

Interfaces werden ähnlich wie Klassen definiert. Jedes Interface wird in einem eigenen Source-File beschrieben, das die Dateinnamenendung `.java` trägt. Am Anfang der Datei, welche die Schnittstellenbeschreibung enthält, kann wie auch bei Klassen die Zugehörigkeit zu einem Paket mittels `package` festgelegt werden. Das nach dem Übersetzungvorgang erzeugte File besitzt ebenfalls die Endung `.class`.
Im allgemeinen bezeichnen Java-Programmierer mit dem Ausdruck „Klassen“ beides, sowohl Klassen, als auch Interfaces. Tatsächlich kann der Begriff „Klasse“ fast überall dort, wo er auftaucht, durch „Schnittstelle“ ersetzt werden. So trifft zum Beispiel alles im Abschnitt 3.1 über Klassen und Pakete gesagte auch in dieser Form auf Schnittstellen zu.

Der grundlegende Unterschied zwischen Klassen und Schnittstellen ist, daß von Schnittstellen keine Instanzen erzeugt werden können. Der Befehl `new` kann ausschließlich Instanzen von Klassen generieren.

Ähnlich wie Klassen durch das Schlüsselwort `class` definiert werden, geschieht dies bei Schnittstellen durch den Befehl `interface`. Er besitzt folgende Syntax:

[*Modifizierer*] interface *Interfacename* [extends *Interfaceliste*] *Interfacekörper*

Modifizierer:

`public` oder `abstract`

Interfacename:

Bezeichner

Interfaceliste:

Interfacename
Interfaceliste , Interfacename

Interfacekörper:

{ Deklaration von Methoden und Konstanten }

Die Angabe eines *Modifizierers* vor `interface` ist optional. Es kann entweder `public` oder `abstract` eingesetzt werden. Durch die Angabe von `public` kann auf die Schnittstelle von jedem anderen Paket aus entweder direkt oder über die Verwendung von `import` zugegriffen werden. Fehlt dagegen `public`, so ist die Benutzung auf das Paket beschränkt, in dem sich das Interface befindet. Jedes Interface wird implizit als abstrakt angenommen, daß heißt, es spielt keine Rolle, ob `abstract` angegeben wird oder nicht.
Für jede Schnittstelle, die mittels `public` für öffentlich erklärt wurde, gilt, daß sämtliche Methoden, die sie enthält, automatisch auch `public` (und `abstract`) sind. Alle in ihr deklarierten Variablen sind automatisch `public`, `static` und `final`. Es dürfen ausschließlich diese Modifizierer in öffentlichen Schnittstellen Verwendung finden.
Für alle Schnittstellen gilt, daß in ihnen im Bezug auf Variablen die Modifizierer `synchronized`, `transient` und `volatile`, und im Bezug auf Methodensignaturen die Modifizierer `final`, `native`, `static` und `synchronized` nicht auftauchen dürfen.

Nach dem Befehl `interface` steht der Name der Schnittstelle. Für die Wahl dieses Bezeichners gelten die gleichen Regeln wie bei der Namensvergabe von Klassen.

Durch das Hinzufügen des optionalen Schlüsselwortes `extends` kann eine Liste von Interfacenamen angegeben werden, von der sich die Schnittstelle ableitet. `Extends` hat hier eine ähnliche Bedeutung wie bei der Klassenvererbung, mit der Außnahme, daß eine Schnittstelle mehrere andere Schnittstellen erweitern darf. In diesem Fall erhält die Schnittstelle alle Methoden und Konstanten jedes der nach `extends` aufgeführten Interfaces. Jede Klasse, welche diese Schnittstelle implementiert, implementiert auch alle Interfaces, die von eben dieser Schnittstelle erweitert werden.
Es gibt auch Interfaces, die sich von keinen anderen Schnittstellen ableiten. In diesem Punkt unterscheiden sich Schnittstellen von Klassen, denn Klassen leiten sich alle von der Klasse `Object` ab. Es existiert in der Tat aber keine Schnittstelle, die von allen anderen Schnittstellen erweitert wird.

Der *Interfacekörper* wird, wie von der Klassendefinition her gewohnt, durch geschweifte Klammern eingeschlossen. Er enthält lediglich Deklarationen von konstanten Variablen und Methodensignaturen. Die Funktionalität der Methoden werden von den Klassen ergänzt, welche die Schnittstelle implementieren.

```
package aktuellpackage;

public interface myInterface extends asuperInterface, anotherInterface
{
    /* Alle Methoden innerhalb dieser Schnittstelle sind implizit
    oeffentlich und abstrakt. Alle Variablen sind implizit public,
    final und static. Das Interface selbst ist oeffentlich, und es
    kann aus allen anderen Paketen darauf zugegriffen werden.
    Ausserdem schliesst es alle Methodensignaturen und Variablen
    der uebergeordneten Schnittstellen asuperInterface und
    anotherInterface ein. */
}
```

Beispiel 3:

```
public interface newInterface extends Interface1, Interface2
{
   public static final theParfume = 4711;
   public abstract String howDoesItSmell();

   // beide oberen Zeilen sind korrekt.

   int prize=75;
   // korrekt; wird automatisch public, static, final.

   int getAmount();
   // korrekt; wird automatisch public, abstract.

   private protected int wrongConstant;
   // Fehler, da nur public und abstract erlaubt.

   private long wrongFunction();
   // Fehler, da in oeffentlichen Schnittstellen alle Methoden
   // implizit public sind.
}
```

3.2.4 Implementieren von Schnittstellen

Interfaces stellen also im Grunde eine Art Schablone dar, deren Funktionalität von den Klassen ergänzt werden muß, die sie implementieren. Daß eine Klasse eine Schnittstelle implementiert, muß in der Klassendefinition durch Angabe des Schlüsselwortes `implements` mitgeteilt werden. `Implements` ist eine Erweiterung des Klassendeklarators `class`.

Die Syntax:

[*Modifizierer*] class *Name* [extends *Name*] [implements *Interfaceliste*]

soll hier nur noch auf die Erweiterung durch `implements` eingegangen werden.[11] Da Klassen nicht unbedingt Schnittstellen implementiern müssen, ist `implements` optional. Wenn es angegeben wird, so wird eine Liste von Interfacenamen, die mindestens einen Interfacenamen enthält, erwartet. Mehrere Interfacenamen werden durch Kommata getrennt.

Beispiel 4:

```
public interface answer
{
    int getTheAnswer();
    ...
}

public interface question
{
    String getTheQuestion();
    ...
}

public class UniverseClass extends aSuperClass implements question,answer
{
    /* Diese Klasse versieht die Methoden aus den Schnittstellen
    question und answer mit Funktionalitaet. In diesem Beispiel
    werden einfach nur Werte zurueckgegeben. */

    int getTheAnswer() { return 42; }

    String getTheQuestion { return "Unknown!"; }
}
```

3.2.5 Interfaces in der Programmierung

Interfaces kommen zum Einsatz, wenn der eigentliche Typ einer Klasse nicht bekannt ist, oder bewußt nicht genauer angegeben wird. Es kann dann auf gewisse Komponenten der Klasse zugegriffen werden, nämlich auf diejenigen, die in der Schnittstelle aufgeführt werden, ohne daß irgendwelche Kenntnisse über die übrigen Komponenten oder über den Klassentyp selbst vorhanden sein müssen. Zur Verdeutlichung ziehen wir nocheinmal Beispiel 4 heran. Folgende Zeilen sind tatsächlich korrekt:

```
answer anObject = constructAnObject();      // Zeile 1
String antwort = anObject.getTheAnswer();   // Zeile 2
```

Wir setzen voraus, daß `constructAnObject()` eine Funktion ist, die eine Referenz auf ein Objekt zurückliefert, sie könnte zum Beispiel der Konstruktor einer Klasse sein.
Die erste Zeile deklariert eine Variable `anObject` die vom Typ `answer`, also ein Interfacetyp ist. Dies bedeutet, daß die Variable auf ein Objekt verweist, welches die Schnittstelle, dessen Typs sie ist, implementiert. Mit dieser Variablen kann nur auf diejenigen Methoden und Variablen

[11] Definition von Klassen werden im vorhergehenden Kapitel behandelt.

In unserem Beispiel enthält die Variable `anObject` also eine Referenz auf ein Objekt, welches das Interface `answer` implementiert. Mittels dieser Referenz kann `anObject` eingeschränkt auf die Methoden und konstanten Variablen, die in der Schnittstelle `answer` definiert sind, auf das Objekt zugreifen. Der eigentliche Klassentyp des Objektes, das die Funktion `constructAnObject()`zurückliefert, ist hierbei weder bekannt noch relevant. Wichtig ist nur, daß das Objekt tatsächlich eine Schnittstelle vom Typ `answer` implementiert.
Nach wie vor entspricht `anObject` immer noch einer Referenz auf das Objekt, auch wenn der Wirkungsbereich auf die Methoden und Variablen der Schnittstelle beschränkt ist. Falls nämlich der Typ des Objektes bekannt sein sollte, kann mittels des Cast-Operators die Beschränkung aufgehoben und alle Komponenten des Objektes erreichbar gemacht werden.

Das folgende Beispiel soll das oben gesagte verdeutlichen. Es ist entnommen aus [28, p.337-338]. In diesem Beispiel wird die Schnittstelle `java.util.Enumeration` verwendet, die in der Java-Klassen-Bibliothek mitgelieferte wird.

Beispiel 5:

```
package collection;

public class LinkedList
{
   private Node root;
   ...
   public Enumeration enumerate()
   {
      return new LinkedListEnumerator(root);
   }
}

class Node
{
    private Object contents;
    private Node next;
    ...

    public Node next()
    { return next; }

    public Object contents()
    { return contents; }
}

class LinkedListEnumerator implements Enumeration
{
    private Node currentNode;

    LinkedListEnumerator(Node root)
    { currentNode = root; }

    public boolean hasMoreElements()
```

```
        public Object nextElement()
        {
            Object anObject = currentNode.contents();
            currentNode = currentNode.next();
            return anObject;
        }
    }
```

Die Benutzung könnte etwa wie folgt aussehen:

```
collection.LinkedList aLinkedList = createLinkedList(); // Zeile 1
java.util.Enumeration e = aLinkedList.enumerate();      // Zeile 2

while (e.hasMoreElements())
{
    Object anObject = e.nextElement();
    // und mache etwas N"utzliches mit anObject
}
```

In Zeile 1 wird eine Variable `aLinkedList` des Klassentypes `LinkedList` deklariert und gleichzeitig über den Konstruktoraufruf initialisiert. In Zeile 2 wird durch den Aufruf der Methode `enumerate` des gerade erzeugten Objektes (`aLinkedList`) eine Instanz der Klasse `LinkedList-Enumerator` geschaffen und in der Variablen `e` abgelegt. Die Variable `e` ist vom Typ eines Interfaces. Mit `e` kann dadurch nur auf die Methoden und Variablen des Objektes zugegriffen werden, die in diesem Interface enthalten sind. In unserem Fall handelt es sich um das Interface `enumerate`. Über den Rest des Objektes oder seiner Klasse muß nichts bekannt sein.
Zwar wird `Enumeration e` so benutzt, als wüsste der Anwender, worum es sich dabei handelt, tatsächlich aber sieht er nicht, daß dies eine Instanz der versteckten Klasse `LinkedList-Enumerator` ist, geschweige denn er kann in irgendeiner Weise auf sie zugreifen. Die Klasse `LinkedListEnumerator` stellt also dem Benutzer eine transparente Schnittstelle zur Verfügung, über die er ihr wesentliches Verhalten steuern kann, während der Rest von ihr, sowie die beiden Hilfsklassen abgkapselt und verborgen bleiben.
Wenn dem Benutzer der Typ des Objektes, das `enumerate` implementiert, bekannt sein würde, könnte er sich mittels „Casting“ Zugriff auf weitere oder alle Komponenten des Objektes verschaffen, auf das `anObject` verweist.

Diese Art, Objekte zur Verfügung zu stellen, entspricht dem in der objektorientierten Programmierung als „verkauft“ bezeichneten Prinzip. Hierbei stellt der Verkäufer ein Objekt zur Verfügung, das der Käufer selbst nicht erzeugen kann, von dem er aber weiß, wie es bedient wird. Durch Aufruf dieser bekannten Funktionen kann der Käufer Eigenschaften verändern, Werte überprüfen oder Handlungen ausführen, ohne viel über das verkaufte Objekt zu wissen.

3.3 Exceptions

Exceptions sind während des Programmablaufs aufgetretene Ausnahmen, oder auch Fehler. Es handelt sich dabei nicht um Fehler, die im Programmcode verankert sind, sondern die durch die Ablaufumgebung hervorgerufen werden. Dies kann z.B. das Abreißen einer Verbindung zu

das laufende Javaprogramm aber gerade hätte zugreifen müssen. Zur Behandlung von Exceptions wird der normale Programmfluß unterbrochen und entweder in einen Teil des Programmes verzweigt, der Routinen zur Bearbeitung der aufgetretenen Ausnahme enthält, oder, falls nicht, das Programm beendet.

3.3.1 Terminologie

Da in den folgenden Abschnitten eine recht neue Art der Ausnahmebehandlung erläutert wird, ist es notwendig, zunächst die Bedeutung einiger Begriffe festzulegen und mit den sonst üblichen Begriffen anderer Programmiersprachen zu vergleichen.
Der eingetretene außergewöhnliche Zustand zur Laufzeit eines Programmes heißt, wie eingangs schon gesagt, *Exception* oder deutsch *Ausnahme*.
Das Auftreten einer Ausnahme wird als *throwing*, deutsch *werfen* bzeichnet. In anderen Programmiersprachen ist hier der Begriff *raising* gebräuchlich[12].
Das *Auffangen* einer gerade eingetretenen Ausnahme und das Ausführen von Befehlen zu deren Behandlung in irgendeiner Weise wird in Java *catching* , in anderen Programmiersprachen *handling* genannt. Der Programmteil, der dies macht, heißt unter Java *catch block*, ansonsten *handler*.
Die Reihe von Funktionsaufrufen, die zu der Methode führte, in welcher die Ausnahme auftrat, wird in Java *stack trace*, bei anderen Sprachen *call chain* genannt.

3.3.2 Herkömmliche Vorgehensweise bei der Ausnahmebehandlung

Bislang mußte diejenige Funktion, die eine Ausnahme registrierte, dies durch einen speziellen Rückgabewert an die sie aufrufende Funktion melden. Etwa auf folgende Weise:

Beispiel 6:

```
int erg = doSomethingThatWorksMostTime();

if (erg == -1)
{
    /* Ein Fehler ist in der Funktion aufgetreten. Sie meldet dies
    durch die Rueckgabe von -1, damit der Fehler hier behandelt
    werden kann. */
}
else
{
    // Normale Programmfortsetzung.
}
```

Die mit diesem Verfahren verbundenen Nachteile liegen auf der Hand:
Ein bestimmter Rückgabewert wird als Fehlerindikator vereinbart. Problematisch wird es, wenn alle regulären Rückgabewerte, die während des fehlerfreien Programmablaufs erreicht werden können, den gesamten Bereich aller möglichen Rückgabewerte bereits belegen, wenn also kein freier Rückgabewert existiert, der als Fehlerindikator verwendet werden könnte.

[12] Im Deutschen wird dies offensichtlich nicht so differenziert betrachtet. Hier heißt es nach wie vor, *ein Fehler* oder *eine Ausnahme ist aufgetreten.*

ausreichend, wenn in der kritischen Funktion mehrere verschiedene Fehler auftreten können. Hier wird ein Verfahren benötigt, das in der Lage ist, die Art des Fehlers an die aufrufende Funktion zu übermitteln, damit diese eine entsprechende, für diesen Fehler vorgesehene Fehlerbehandlung einleiten kann.
Bei diesem Verfahren muß nach jedem Aufruf der fehlerkritischen Funktion deren Rückgabewert auf den Fehlerindikator überprüft werden. Dies hat zur Folge, daß der Programmcode, der lediglich für die Registrierung und Behandlung von Ausnahmen zuständig ist, über den gesamten Programmtext verteilt liegt. Eine klare Trennung des normalen Programmcodes von dem Code zur Ausnahmbehandlung wäre hier wünschenswert.
Dem Benutzer einer Methode, in deren Verlauf Ausnahmen auftreten können, muß auf irgendeine Weise mitgeteilt werden, welche Fehler auftreten können, und wie sie in den Rückgabewerten angezeigt werden. Im allgemeinen geschieht dies in der Programmdokumentation, zum Beispiel durch Hinzufügen von Kommentarzeilen im Source-Text. Diese Dokumentation steht aber in keiner Weise in irgendeiner Verbindung zum Programmcode, das heißt, es können Inkonsistenzen auftreten, wenn beispielsweise der Source-Code geändert, die Dokumentation aber nicht angepaßt wird.

3.3.3 Verfahren der Ausnahmebehandlung in Java

Ausnahmen können auf zwei Arten erzeugt werden. Zum einen kann durch eine illegale Handlung, beispielsweise durch eine Division durch Null, implizit eine Ausnahme auftreten. Zum anderen kann eine Ausnahme explizit im Programmtext durch den Befehl `throw` hervorgerufen werden.
Eine Ausnahme unterbricht den normalen Progammablauf. Wenn der Programmierer entsprechende Vorsorge getroffen hat, wird jetzt zu einem Teil des Programmes verzweigt, der Routinen zur Behandlung der aufgetretenen Ausnahme beinhaltet. Dieser Programmteil wird zunächst in derjenigen Funktion gesucht, in welcher der Fehler aufgetreten ist. Wenn Java dort nicht fündig wird, setzt es die Suche in der Methode fort, die diese Funktion aufgerufen hat. Bleibt die Suche auch dort erfolglos, durchsucht Java den Aufrufer dieser Methode, etc. Wenn der gesamte Stack Trace durchsucht, also die oberste Ebene in der Aufrufkette erreicht wurde, übernimmt der Java-Interpreter selbst die Ausnahmebehandlung. Diese besteht in der Ausgabe der Ausnahme mit anschließendem Programmabbruch.

3.3.4 Ausnahmeobjekte

In den bisherigen Ausführungen wurden Ausnahmen als etwas Abstraktes beschrieben. An dieser Stelle soll nun erläutert werden, worum es sich bei Ausnahmen konkret handelt.
Eine Ausnahme oder auch Exception ist eine Instanz der Klasse `Throwable` oder einer ihrer Subklassen. Wenn eine Ausnahme auftritt, wird ein Objekt erzeugt, dessen Typ die Art der Ausnahme repräsentiert.

Die in der Java-Klassenbibliothek vordefinierten Exceptions und ihre Klassenhierarchie sind in dem Paket `java.lang` zu finden.

Der Programmierer kann eigene Exceptions entwerfen. Seine Klasse muß lediglich `Exception` aus dem Paket `java.lang` der Java-Klassenbibliothek erweitern.

Für die Programmentwicklung ist es wichtig, mittels eines einheitlichen Verfahrens Methoden zu kennzeichnen, die Gefahr laufen, eine Ausnahme zu erzeugen. Aus dieser Kennzeichnung muß hervorgehen, welche Ausnahmen während der Abarbeitung dieser Methode auftreten können.

Hierzu wird die Deklaration von Methoden um das Schlüsselwort `throws` erweitert. Es hat folgende Syntax:

`throws` *Exceptionliste*

wobei

Exceptionliste:

Exceptionname
Exceptionliste , Exceptionname

`Throws` steht nach der Parameterliste in der Methodensignatur. Nach `throws` wird eine durch Kommata getrennte Liste von Bezeichnern für Exceptions erwartet. Es können dies die Namen der Ausnahmenklassen sein, die in `java.lang` definiert sind, oder vom Programmierer selbst geschriebene Ausnahmeklassen.

Beispiel 7:

```
public class aClassWithAnExceptionalMethod
{
    public void canCauseAnException() throws anException
    {  ...  }
}
```

Der Benutzer dieser Klasse wird durch `throws` darauf hingewiesen, daß die Methode `canCause-AnException` möglicherweise eine Ausnahme namens `anException` verursachen kann, und daß er selbst für ein Behandeln dieser Exception Sorge zu tragen hat. `AnException` muß eine Subklasse von `java.lang.Throwable` sein.

Sobald der Programmierer weiß, welche Ausnahmen in einer Funktion auftreten können, hat er zu entscheiden, ob die Funktion diese Ausnahmen selbst behandelt, oder ob der Benutzer die Behandlung der Ausnahmen übernehmen soll. Der Benutzer ist also nur über Ausnahmen des letzteren Falls zu informieren. Das bedeutet, daß nur solche Ausnahmen durch `throws` im Methodenkopf anzuzeigen sind, deren Behandlung nicht oder nicht vollständig in der Methode erfolgt, in der sie auftreten können.
Ausnahmen, die zu jeder Zeit auftreten können, müssen ebenfalls nicht in der `throws`-Liste erscheinen. Ein Beispiel hierfür ist `OutOfMemoryError`, eine Exception, die überall und zu jedem Zeitpunkt auftreten kann. Es handelt sich dabei um Runtime-Exceptions. Ein ausführlicher Blick in das Paket `java.lang` zeigt, daß dies alle Subklassen der Klasse `RuntimeExceptions` und der Klasse `Error` sind.
Selbstverständlich können diese Art der Ausnahmen trotzdem in die `throws`-Liste aufgenommen werden. Der Benutzer kann sie aber ignorieren, er muß sie nicht in der aufrufenden Funktion behandeln. Dies erledigt der Java-Interpreter für ihn.
Es genügt also, lediglich vor Nicht-Runtime-Exceptions mittels `throws` zu warnen. In der Java-Klassenbibliothek sind sechs solcher Ausnahmen vordefiniert:

```
ClassNotSupportetdException
IllegalAccessException
InstantionException
InterruptedException
NoSuchMethodException
```

Jeder dieser Namen bezeichnet einen Fehler, der explizit vom Programmierer hervorgerufen wird. Alle Ausnahmen dieser Art müssen vom Benutzer aufgefangen werden, falls sie in einer Funktion auftreten können.

3.3.6 Erzeugen von Ausnahmen

Weiter oben wurde gesagt, daß es eine implizite und eine explizite Möglichkit gibt, Ausnahmen zu erzeugen. Die implizite Möglichkeit kommt zum Beispiel bei Runtime-Exceptions zur Anwendung, die jederzeit auftreten können. Mit dem expliziten Verfahren kann der Programmierer selbst Ausnahmen erzeugen. Dies wird zum Beispiel bei selbstgeschriebenen Exceptions benötigt.
Das Auftreten einer Ausnahme wird mit dem Befehl `throw` forciert. Er hat folgende Syntax:

`throw` *AusnahmeObjekt*

Bei *AusnahmeObjekt* handelt es sich um eine Referenz auf ein Objekt, das eine Instanz einer Ausnahmeklasse ist.

Beispiel 8:

```
public class aGlass
{
    boolean checkEmpty()
    {
        /* Ueberpruefe den Fuellstand des Glases und gib True
        zurueck, falls das Glas leer ist, sonst False. */
    }

    void drink() throws glassIsEmptyException
    {
        if (checkEmpty()) { throw new glassIsEmptyException; }
    }
    ...
}
```

Wenn die Methode `drink()` aufgerufen wird, obwohl das Glas leer ist, wird mittels `new` ein Objekt der Klasse `glassIsEmptyException` erzeugt. Die Referenz auf dieses Objekt wird durch `throw` an den Aufrufer dieser Methode weitergeleitet.

Damit Ausnahmebehandlungen eingeleitet werden können, muß die aufgetretene Ausnahme aufgefangen und identifiziert werden. Dies geschieht in den `try-catch-finally`-Blöcken. Diese sind folgendermaßen aufgebaut:

```
try { ... }
catch (Argument) { ... }
finally { ... }
```

Erscheint im Programmtext ein `try`-Block, so muß ihm mindestens ein `catch`- oder genau ein `finally`-Block folgen. Es können beliebig viele `catch`-Blöcke aufeinanderfolgen. Es kann entweder keinen oder genau einen `finally`-Block geben.

Innerhalb des `try`-Blockes stehen die Anweisungen und Funktionsaufrufe, die mögliche Ausnahmen verursachen können. Der `try`-Block kann in etwa so interpretiert werden, als daß er versucht, die in ihm enthaltenen Befehle auszuführen. Sobald eine Exception auftritt, wird die Abarbeitung in diesem Block unterbrochen und zu den `catch`-Blöcken verzweigt.

Jeder `catch`-Block ist mit einem *Argument* versehen, das einen Klassentyp und eine Variable als Parameter enthält. Bei dem Klassentyp muß es sich um den Namen einer Exceptionklasse handeln, also eine Unterklasse der Klasse `Throwable`. Anhand dieses *Argumentes* erkennt Java, welcher `catch`-Block für welche Ausnahme zuständig ist. Gleichzeitig wird dem entsprechenden `catch`-Block die Exception mittels der Variablen im *Argument* als Referenz zugänglich gemacht. *Argument* gleicht also einem Parameter in der Parameterliste einer Funktionsdeklaration.
Zum Beispiel bewirkt `catch (ArithmeticException e)`, daß in diesen Block im Falle einer Ausnahme namens `AithmeticException` verzweigt wird, wobei `e` eine Referenz auf das Ausnahmeobjekt erhält.
Besitzt keiner der `catch`-Blöcke ein passendes Argument für die aufgetretene Ausnahme, so wird auch in keinen der Blöcke verzweigt. In diesem Fall wird, nachdem ein eventuell vorhandener `finally`-Block ausgeführt wurde, weiter zurückgesprungen in die aufrufende Funktion bzw. zum umgebenden `try`-Block. Falls der Befehl, der den Aufruf bewirkte, sich in einem `try`-Block einer dort vorhanden `try-catch-finally`-Struktur befinden sollte, wird in deren `catch`-Blöcken weitergesucht, etc.
An dieser Stelle muß darauf hingewiesen werden, daß Java in den ersten passenden `catch`-Block verzweigt und alle nachfolgenden Blöcke dann ignoriert werden. Ein `catch`-Block gilt dann als passend, wenn das Objekt der aufgetretenen Ausnahme eine Instanz der Klasse oder eine Instanz einer der Unterklassen des in *Argument* enthaltenen Klassentyps ist. Dies bedeutet, daß die Reihenfolge der `catch`-Blöcke wichtig ist. Blöcke, die genauer spezifierte Exceptions als Klassentyp in ihrem *Argument* erwarten, sollten vor denen in Erwartung allgemeiner gefaßten Exceptions stehen.
Anders formuliert: `catch (Exception e)` würde alle Ausnahmen abfangen, samt allen darin enthaltenen Runtime- und Non-Runtime-Exceptions, kurzerhand alle Exceptions, die sich von der Klasse `Exception` ableiten. Ein solcher Ausdruck macht in der Regel keinen Sinn. Stattdessen ist es besser nur einzelne Exceptions abzufangen, und die anderen den Stack Trace hinaufwandern und vom Interpreter behandeln zu lassen.

An die Reihe der `catch`-Blöcke kann sich ein `finally`-Block anschließen. Wenn er existiert, werden die in ihm stehenden Anweisungen auf jeden Fall ausgeführt, gleichgültig, ob eine Exception aufgetreten ist oder nicht, und gleichgültig ob einer der `catch`-Blöck zur Ausführung gekommen ist oder nicht; selbst dann wird die `finally`-Anweisung abgearbeitet, wenn in einem der `catch`-Blöcke ein `return`-Befehl in irgendeiner Form auftaucht. Der `finally`-Block ist sozusagen die

handlung an anderer Stelle zu bereinigen. Es handelt sich in der Regel dabei um Dinge, die im `try`-Block in Erwartung eines fehlerfreien Durchlaufs angefangen wurden, deren Fortführung aber nach dem Auftreten einer Ausnhame keinen Sinn mehr machen. Im `finally`-Block werden sie zu einem Ende gebracht.

Beispiel 9:

```
public void aMethod()
{
    try
    {
        somethingVeryExceptional(); //versucht die Funktion auszufuehren
    }
    catch(NullPointerException n)   //Unterklasse von RuntimeException
    { ... }
    catch(RuntimeException r)       //Unterklasse von Exception
    { ... }
    catch(IOException i)            //Unterklasse von Exception
    { ... }
    catch(Exception e)              //Unterklasse von Throwable
    // Faengt alles ab, was bis hierher durchgekommen ist.
    { ... }

    finally // Wird in jedem Fall noch ausgefuehrt
    { ... }
}
```

3.3.8 Nützliche Funktionen für die Verwendung von Ausnahmen

Abschließend möchte ich zwei für die Programmierung von Ausnahmebehandlungen nützliche Funktionen vorstellen, die in der Klasse `java.lang.Throwable` verankert sind.
Mithilfe der Funktion `String getMessage()` ist es möglich, einem Ausnahmeobjekt bei seiner Erzeugung einen erklärenden Text mitzugeben. In Beispiel 8 ließe sich damit die Funktion `drink()` wie folgt abändern:

```
void drink() throws glassIsEmptyException
{
    if (checkEmpty())
    { throw new glassIsEmptyException("Das Glas ist leider leer."); }
}
```

In einem `catch`-Block könnte dann dieser Text als Hilfe an den Benutzer ausgegeben werden:

```
catch (glassIsEmptyException g)
{
  System.out.println(g.getMessage());
  ...
}
```

```
Das Glas ist leider leer.
```

Die zweite nützliche Funktion lautet `printStackTrace()`. Mit ihr kann der Inhalt des Stack Trace, also die Aufrufkette bis zu der Funktion, welche die Ausnahme „geworfen“ hat, ausgegeben werden. In Verbindung mit `getMessage()` kann dies nützlich sein, um dem Benutzer Ort und Art einer Exception mitzuteilen.

3.4 Nativer Code

Nativer Code ist übersetzter, plattformabhängiger Code einer anderen Programmiersprache als Java. Beispiele hierfür sind C- oder Assembler-Routinen. Java ist in der Lage, Programmcode dieser Art zu integrieren.

3.4.1 Gründe für die Verwendung von Nativem Code

Zwei Gründe können dazu führen, daß die Einbindung von Nativem Code in ein Java-Programm notwendig wird.

Der eine Grund ist, daß spezielle Funktionen des Betriebssystems nutzbar gemacht werden sollen, die von der Java-Klassenbibliothek nicht unterstützt werden. Ebenso kann es notwendig sein, bestimmte Hardwarekomponenten anzusteuern, wie zum Beispiel einen 3D-Beschleuniger für Grafikkarten. Auch für dieses Vorhaben findet man keine Unterstützung seitens der Java-Klassenbibliothek.

Der andere Grund ist, daß besonders zeitkritische Funktionen realisiert werden sollen. Native Methoden nutzen in diesem Fall den Vorteil aus, daß Java hinsichtlich der Geschwindigkeit nicht eine so gute Leistung zu vollbringen vermag, wie äquivalente C-Programme. Die übliche Vorgehensweise ist hier, zum Beispiel das Innere einer Schleife in C zu programmieren und in einen Körper, der als Java-Programmcode vorliegt, einzubetten. Dies bedeutet, daß der native Teil des Programms dem Anwender verborgen bleibt, und er mit dem Programm, als wäre es ausschließlich in Java programmiert, umgehen kann. Tatsächlich sind einige Funktionen der Java-Klassenbibliothek auf diese Weise realisiert, um die Gesamteffizienz des Systems zu steigern.

3.4.2 Nachteile von Nativem Code

Durch die Implementierung von Nativem Code in einem Java-Programm geht eine wesentliche Eigenschaft des Programmes verloren: Es ist nicht mehr plattformunabhängig. Das heißt, das Programm ist nicht mehr in jeder beliebigen Java-Umgebung auf der Welt lauffähig.

Dies bedeutet besonders harte Konsequenzen für Applets. Bekanntermaßen werden Applets von sogenannten Browsern ausgeführt. Aus Sicherheitsgründen läßt es keiner der Browser zu, daß Nativer Code zusammen mit einem Applet über das Internet bzw. über das WorldWideWeb geladen wird. Lediglich Applets, die einzig und allein von der Java-Klassen-Bibliothek, genauer vom Package `java`, abhängig sind, können von den Browsern zur Ausführung gebracht werden. Applets, die nativen Code beinhalten, verlieren also ihren Sinn und Zweck eben dadurch, daß sie lediglich auf dem lokalen Rechner lauffähig sind und nicht mehr über das Netz zur Verfügung gestellt werden können.

Zwei Technologien befinden sich derzeit in der Entwicklungsphase, mit deren Hilfe die Ausführung von Java-Code beschleunigt werden soll, ohne daß auf Nativen Code zurückgegriffen werden muß. Die Plattformunabhängigkeit bleibt somit gewahrt.

Die erste Technologie ist völlig transparent zum Java-Code. Der Programmierer benötigt keinerlei Kentnisse über sie. Es handelt sich dabei um einen „Just-In-Time-Übersetzer“[13], der anstelle des Java-Interpreters den Code interpretiert. Dieser Compiler übersetzt, während der Bytecode einer Java-Funktion ausgeführt wird, diesen in einen nativen Binär-Code. Dieser Binär-Code wird in einer Art Cache gehalten, wo er das nächste Mal hervorgeholt wird, wenn die Funktion wieder zur Ausführung kommt. Diese Technologie soll in einigen Experimenten die Geschwindigekt von übersetztem C-Code erreicht haben, nachdem der erste Aufruf der Funktion nur ein wenig mehr Zeit gekostet haben soll.

Der `java2c`-Wandler ist die zweite Technologie, die hier kurz vorgestellt werden soll. Der Wandler übersetzt eine `class`-Datei in einen C-Quelltext. Der Quelltext kann dann von dem installierten C-Compiler auf dem Rechner compiliert werden. Dadurch erhält man eine native Klassenbibliothek, die immer dann benutzt wird, wenn eine Methode aus einer dieser Klassen aufgerufen wird. Allerdings geschieht dies nur auf dem lokalen Rechner. Über das Netzwerk wird nach wie vor das eigentliche Java-Äquivalent zur Verfügung gestellt. Diese Technologie ermöglicht, die Geschwindigkeit von voll optimierten C-Code zu erreichen.

3.4.4 Einbinden von Nativem Code in Java-Programme

Das Einbinden von Nativem Code soll Schritt für Schritt an dem bekannten `Hello World` Beispiel erläutert werden. Der Native Code entstammt in diesem Fall der Programmiersprache C. Das Beispiel wurde aus [34] entnommen.

Schreiben des Java-Codes: Der erste Schritt ist es natürlich, die Klasse zu schreiben, welche die nativen Methoden enthält. In unserem Fall ist dies die Klasse `HelloWorld`. In ihr befindet sich die native Funktion `displayHelloWorld` .

```
class HelloWorld
{
    public native void displayHelloWorld();

    static
    {
         System.loadLibrary("hello");
    }
}
```

Die Funktion, die später den Nativen Code enthalten wird, ist durch den Modifizierer `native` gekennzeichnet.

Etwas befremdlich mag der `static`-Block, also der Klasseninitialisator, erscheinen. Das Laden der Klasse und das Laden der Dynamically-Loadable-Library werden miteinander verbunden.

[13] Der Just-In-Time-Übersetzer wird in einem der folgenden Kapitel ausführlich behandelt.

daß irgendwelche halbfertigen Sachen nach und nach ins System geladen werden.

In einem anderen Source-File mit Namen `Main.java` wird nun eine Klasse definiert, die eine Methode enthält, welche die native Funktion in der Klasse `HelloWorld` aufruft.

```
class Main
{
    public static void main(String args[])
    {
        new HelloWorld().displayHelloWorld();
    }
}
```

Hieraus geht hervor, daß native Methoden äquivalent zu Java-Standardfunkionen aufgerufen werden. Auch die Parameterübergabe erfolgt nach völlig gleichen Gesichtspunkten. In unserem Fall werden jedoch keine Parameter übergeben.

Übersetzen des Java-Codes: Wie gewohnt werden beide Quelldateien übersetzt. Das Ergebnis sind die zwei Klassenfiles `HelloWorld.class` und `Main.class`.

Erzeugen des Includefiles: Mit Hilfe des Utility-Programms `javah` wird aus `HelloWorld.class` ein C-Includefile generiert mit Namen `HelloWorld.h`.
Ein Blick in das Includefile zeigt die für C üblichen Deklarationen. Es wird darin ein Struct[14] definiert, das die Java-Klasse `HelloWorld` repräsentiert. Desweiteren wird eine Funktion in der C typischen Schreibweise deklariert:

```
extern void HelloWorld_displayHelloWorld(struct HHelloWorld *);
```

Dies ist die Deklaration für die C-Funktion, die später in `displayHelloWorld()` implementiert wird. Der Name wird berechnet aus dem Paketnamen, aus dem Klassennamen und aus dem Namen der nativen Java-Funktion. Wenn weitere native Funktionen existieren würden, würden sie ebenfalls an dieser Stelle auftauchen.

Erzeugen des Stub-Files: Zusätzlich zum gerade erzeugten Includefile, muß ein sogenanntes Stub-File generiert werden. Dieses Stub-File enthält C-Programmcode, der die Verbindung zwischen Java-Klasse und ihrer parallelen Darstellung als C-Struktur herstellt. Die Funktion dieses Stub-File ist es, automatisch Argumente und Rückgabewerte zwischen Java und C umzuwandeln und hin und her zu transferieren. Es ist nicht notwendig, Einzelheiten über das Stub-File zu wissen, lediglich, daß es übersetzt und mit dem C-Code zusammen gebunden werden muß, damit letzterer mit Java interagieren kann.
Das Stub-File wird ebenfalls mit dem Utility `javah` aus dem Klassenfile `HelloWorld.class` generiert, allerdings unter Angabe des Schalters `-stubs`. Das Ergebnis ist eine Datei namens `HelloWorld.c`.

[14] Es handelt sich hierbei um einen Ausdruck für einen gängigen Datentyp in C.

mit dem Programmieren des C-Codes begonnen werden. Dieser wird in einem File mit Namen `HelloWorldImp.c` abgelegt und sieht in unserem Fall so aus:

```
#include <StubPreamble.h>
#include "HelloWorld.h"
#include <stdio.h>

void HelloWorld_displayHelloWorld(struct HHelloWorld *this)
{
    printf("Hello World!");
    return;
}
```

`StubPreamble.h` ist ein Includefile, das Teil des JDK ist, und das genug Informationen enthält, um C-Programme in Java einzubinden. Es muß daher von dem jeweiligen zu implementierenden C-Programm grundsätzlich mittels `include` in den Programmtext eingefügt werden.
`HelloWorld.h` ist das von `javah` generierte Include-File.
Jetzt wird auch ersichtlich, wieso die zu implementierende C-Funktion einen Parameter besitzt, obwohl `displayHelloWorld()` keinen Parameter übergibt: Durch diesen Parameter wird der `this`-Zeiger simuliert, wie er zum Beispiel aus C++ bekannt ist.

Erzeugen der Dynamically Loadable Library: An dieser Stelle muß nun die Dynamically Loadable Library generiert werden, die zur Laufzeit in das Java-Programm mittels `System.loadLibrary("hello")` eingebunden wird. Dazu werden mit dem C-überstzer die Dateien `HelloWorld.c` und `HelloWorldImp.c` in eine Dynamically Loadable Library übersetzt und in einem File mit dem Namen `hello` gespeichert. Die Dokumentation des verwendeten C-Überstzers informiert darüber, mit welchem Übersetzeraufruf dies erreicht wird.

Nach diesem Schritt kann die Java-Applikation, die jetzt eine Kombination aus Java- und C-Code ist, gestartet werden. Wenn alles richtig gemacht wurde, und sich in den Programmtexten keine Fehler eingeschlichen haben, sollte auf dem Bildschirm `Hello World!` erscheinen.

3.5 Schlußbemerkung

Die Programmiersprache Java besitzt mit Packages, Interfaces und Exceptions Features, deren Vorteile für die Programmierung sich nicht nur auf die Implementation beschränken, sondern weit in die Programmentwicklung und in das Programmdesign hineinreichen. Sobald der Programmierer sich auf diese zum Teil neue Systematik umgestellt hat, ist sie ihm eine große Hilfe bei der Entwicklung von Programmcode, der aufgrund dieser Strukturen bereits ein hohes Maß an Übersichtlichkeit und Effizienz besitzt und auch zu späteren Zeitpunkten eine weitgehend problemlose Wartbarkeit und Erweiterbarkeit garantiert.

Nativer Code öffnet Java die Welt der Betriebssysteme und Hardwarekomponenten. Er gestattet die Realisierung von zeitkritischen Abläufen durch Implementierung von plattformabhängigen Programmcode.

Seminarbeitrag von **Bernhard Haumacher**

In einem Betriebssystem unterscheidet man bei parallel ablaufenden Aktivitäten zwischen zwei verschiedenen Formen: Ein Prozeß ist eine eigenständige Aktivität, welcher ein eigener Adreßraum zugeteilt ist. Jeder Prozeß kann wiederum aus mehreren Aktivitätssträngen bestehen, welche dann alle in demselben Adreßraum ablaufen und ohne Betriebssystemunterstützung auf gemeinsame Datenstrukturen zugreifen können. Diese Aktivitätsstränge nennt man Threads.

Die Java zugrundeliegende abstrakte Maschine soll betriebssystemunabhängige und portable Programmierung ermöglichen, ohne aber das zugrundeliegende Betriebssystem ersetzen zu wollen. Java selbst kennt daher keine Adreßräume und damit auch keine Prozesse, wohl aber Threads.

Der Aufruf von Betriebssystem-Prozessen ist zwar über die Kapselklasse `java.lang.Process` möglich, soll hier aber nicht näher behandelt werden. Threads werden dagegen von Java voll unterstützt, wofür Java eigene Sprachkonstrukte zur Verfügung stellt.

4.1 Ein erster Thread

Eine Instanz der Klasse `java.lang.Thread` repräsentiert einen Thread, also einen eigenen Aktivitätsstrang. Dabei muß man aber klar zwischen dem passiven Speicherobjekt, welches bei der Instanzierung der Klasse `java.lang.Thread` angelegt wird, und dem eigentlichen Thread, der sich im Zustand der abstrakten Maschine widerspiegelt, unterscheiden. Im folgenden soll das Speicherobjekt, also die Instanz der Klasse `java.lang.Thread`, mit Threadobjekt und der neue Aktivitätsstrang mit Thread bezeichnet werden.

Die Klasse `java.lang.Thread` bietet unter anderem folgende Methoden an:

```
public                   Thread()
public                   Thread(Runnable target)
public synchronized void Thread.start()
public               void Thread.run()
```

Der Konstruktor der Klasse `java.lang.Thread` legt nur das Threadobjekt an und bereitet so den eigentlichen Threadstart vor. Die Abspaltung des neuen Aktivitätsstrangs, also die Erzeugung des neuen Threads, geschieht erst beim Aufruf der Methode `Thread.start()`. Diese Methode erzeugt den neuen Thread und kehrt sofort zurück, unabhängig von der Dauer der Berechnung, die der neue Thread ausführt. Der neu erzeugte Thread führt die Methode `Thread.run()` aus. Mit Erreichen des Endes der Methode `Thread.run()` wird der neue Thread beendet, nicht jedoch das Threadobjekt zerstört.

zen, was sich in zwei Sorten Konstruktoren wiederspiegelt:

Der direkte Aufruf des Standardkonstruktors `public Thread()` der Klasse `java.lang.Thread` macht wenig Sinn, da sich so ja keine Aktivität für den neuen Thread spezifizieren läßt. Eine von `java.lang.Thread` abgeleitete Klasse müßte die Methode `public void Thread.run()` überschreiben, um dann als eigener Thread ausführbar zu sein.

Der Konstruktor `Thread(Runnable target)` dagegen erzeugt ein neues Threadobjekt und verbindet dieses mit einem beliebigen Objekt, dessen Klasse das Interface `java.lang.Runnable` implementiert:

```
public abstract void Runnable.run()
```

Die einzige Methode, die dieses Interface erklärt, ist `Runnable.run()`. Beim Aufruf der Methode `Thread.start()` führt der neu erzeugte Thread jetzt die Methode `Runnable.run()` des an den Threadkonstruktor übergebenen Objekts `target` aus.

Eine weitere (statische) Methode der Klasse `java.lang.Thread` ist:

```
public static Thread currentThread()
```

Da jedes Objekt über den Aufruf der statischen Methode `Thread.currentThread()` der Klasse `java.lang.Thread` eine Referenz auf das Threadobjekt des aktuellen Threads erhalten kann, macht es im allgemeinen wenig Sinn, eine neue Klasse von der Klasse `java.lang.Thread` abzuleiten, nur um eine Methode als eigenen Thread auszuführen. Der Weg über das Interface `java.lang.Runnable` ist der flexiblere und daher normalerweise empfehlenswert.

Das folgende kleine Beispiel verwendet den zweiten der oben besprochenen Ansätze mit dem Interface `java.lang.Runnable`. Das Hauptprogramm `Test1` erzeugt einen neuen `Thread clock`, dem bei der Konstruktion eine Referenz auf ein Objekt der Klasse `ZeitAnsage` mitgegeben wird. Nach dem Starten des neuen Threads versetzt sich das Hauptprogramm in eine Endlosschleife, in der es immer wieder ein Lebenszeichen auf der Java-Console ausgibt und sich dann in einen zweisekündigen Tiefschlaf versetzt. Auch der neue Thread, der jetzt die Methode `ZeitAnsage.run()` ausführt, tut nichts anderes, als alle drei Sekunden die aktuelle Systemzeit an die Java-Console auszugeben.

```
import java.lang.Thread;
import java.lang.Runnable;
import java.lang.System;
import java.util.Date;

class ZeitAnsage implements Runnable {
  public void run() {
    while (true) {
      Date now = new Date();
      System.out.println(now.toLocaleString());
      try {Thread.sleep(3000); } catch (InterruptedException e) {break; }
    }
  }
}
```

```
  public static void main(String args[]) {
    Thread clock = new Thread(new ZeitAnsage());
    clock.start();

    while (true) {
      System.out.println("Ich arbeite !");
      try {Thread.sleep(2000); } catch (InterruptedException e) {break; }
    }
  }
}
```

4.1.1 Beliebige Methoden nebenläufig ausführen

In Java kann normalerweise nicht eine beliebige Methode eines beliebigen Objekts als nebenläufige Aktivität aufgerufen werden. Um eine Methode also parallel ausführeren zu können - d.h. der Aufrufer kann unmittelbar nach dem Aufruf mit seiner Berechnung fortsetzen, ohne auf die Beendigung der Berechnung in der aufgerufenen Methode warten zu müssen -, muß sie standardmäßig immer die Form

```
public void run()
```

haben. Es können also keine Parameter übergeben und keine Ergebnisse zurückgeliefert werden. Auch der Name der Methode müßte prinzipiell immer `run()` lauten, da nur die Methode `run()` im Interface `java.lang.Runnable` erklärt ist und immer nur diese Methode beim Starten eines Threads zur Ausführung kommt.

Zum Vergleich sieht die Win32 API Routine zum Erzeugen eines neuen Threads folgendermaßen aus:

```
HANDLE CreateThread(
 LPSECURITY_ATTRIBUTES  lpThreadAttributes, // address of thread security attributes
 DWORD                  dwStackSize,        // initial thread stack size, in bytes
 LPTHREAD_START_ROUTINE lpStartAddress,     // address of thread function
 LPVOID                 lpParameter,        // argument for new thread
 DWORD                  dwCreationFlags,    // creation flags
 LPDWORD                lpThreadId          // address of returned thread identifier
);
```

Hier kann eine beliebige Funktion als eigenständiger Thread gestartet werden, die einen 32 Bit Wert als Argument nimmt und einen 32 Bit Wert als Resultat liefert. Um hier beliebige Parameter an die Threadfunktion übergeben zu können, muß man sich eine Speicherstruktur, die die Parameter aufnehmen soll, reservieren, diese vorbelegen und einen Zeiger darauf an die API Routine `CreateThread()` übergeben.

Ein ähnlicher Trick kann auch in Java angewendet werden, um eine beliebige Methode eines beliebigen Objekts in einem eigenständigen Thread ausführen zu können. Das ist möglich, da jeder Methode, also auch `void Runnable.run()`, implizit die Instanz der Klasse übergeben wird, für die die Methode aufgerufen wurde.

Man definiert sich also eine sogenannte Adaptor-Klasse, die das Interface `java.lang.Runnable` implementiert. Die Funktionalität dieser Klasse besteht ausschließlich darin, das Objekt und

Methode des übergebenen Objekts mit den entsprechenden Parametern aufzurufen und deren Ergebnis für weitere Verarbeitung zwischenzuspeichern. Die Adaptor-Klasse ist sozusagen der Behälter, in dem die Parameter übergeben werden und der nach der Ausführung das Ergebnis aufnimmt. In der Methode `run()` der Adaptor-Klasse kann dann auf die bei der Konstruktion übergebenen Objektvariablen zugegriffen werden.

Die folgenden Code-Fragmente zeigen wie man prinzipiell beim nebenläufigen Aufruf einer beliebigen Methode vorgeht:

```
// eine beliebige Klasse ...
public class AnyClass {
  // ... mit einer beliebigen Methode
  public AnyResult anyMethod(AnyParam1 anyParam1, ..., AnyParamN anyParamN) {
    <nebenlaeufig auszufuehrender Code>
  }
}

// die Adaptor-Klasse
public class AnAdaptor implements Runnable {
  // Parameter als lokale Variablen
  AnyClass  _obj;
  AnyParam1 _param1;
    ...      ...
  AnyParamN _paramN;

  public AnyResult _result;

  // der Konstruktor
  public AnAdaptor(AnyClass obj, AnyParam1 param1, ..., AnyParamN paramN) {
    _obj    = obj;
    _param1 = param1;
      ...      ...
    _paramN = paramN;
  }

  public void run() {
    _result = _obj.anyMethod(_param1, ..., _paramN);
  }
}

// An einer beliebigen Codestelle:
  // obj sei von der Klasse AnyClass

  AnAdaptor adaptor = new AnAdaptor(obj, param1, ..., paramN);
  Thread    thread  = new Thread(adaptor);

  thread.start();  // Aufruf von obj.anyMethod(param1, ..., paramN);

  // Nach Beendigung des Threads kann auf
  // adaptor._result zugegriffen werden.
```

Eine notwendige Voraussetzung für die Kooperation von mehreren parallelen Prozessen ist die Synchronisierung der verschiedenen Aktivitäten, um Inkonsistenzen zu vermeiden. Ohne Synchronisierung würden schon in ganz trivialen Fällen nicht vorhersagbare Effekte auftreten, wie folgendes Beispiel zeigt.

Die Klasse `DecrementableCounter`, erklärt einen bei der Konstruktion setzbaren Zähler, der später durch `decrementIfPositive()` nur noch erniedrigt werden kann, falls sein Zustand noch positiv ist. Der Zustand kann über `getState()` abgefragt werden. Eine wichtige Klasseninvariante ist also, daß der Zustand des Zählers nie negativ werden kann.

```
class DecrementableCounter {
  int state;

  DecrementableCounter(int defaultState) {
    state = defaultState;
  }

  public int getState() {
    return state;
  }

  public void decrementIfPositive() {
    if (getState() > 0) {
      state = state - 1;
    }
  }
}
```

Würde dieses Beispiel in einer Umgebung mit mehreren nebenläufigen Aktivitäten zur Anwendung kommen, so könnte sich folgendes Szenario abspielen:

- Der aktuelle Zählerzustand beträgt im Moment `state == 1`.
- Zwei Threads passieren in der Methode `decrementIfPositive()` quasi gleichzeitig die Abfrage auf Positivität des Zählers.
- Beide erniedrigen erst jetzt den Zähler um 1.

Dies hätte zur Folge, daß nach Beendigung der Zähler den Zustand `state == -1` hätte, was klar im Widerspruch zur Klasseninvarianten stünde.

Das Problem ergibt sich hier aus der Tatsache, daß sich zwei Threads gleichzeitig in der Methode `decrementIfPositive()` aufhalten, die den Zustand des Objekts verändert. Um derartige Inkonsistenzen zu vermeiden, muß man also verhindern, daß sich gleichzeitig mehrere Threads in Methoden aufhalten, die den Zustand eines Objekts verändern. Die gleiche Situation könnte nämlich auftreten, wenn zwei Threads gleichzeitig zwei verschiedene Methoden durchlaufen, die beide den Zustand des Objekts beinflussen.

Ein Monitor ist eine Konstruktion, die verhindert, daß sich mehrere Threads gleichzeitig in einem kritischen Bereich aufhalten. Im obigen Fall würden alle Methoden, die eine Zustandsveränderung vornehmen, zusammen einen kritischen Bereich bilden.

Alle kritischen Bereiche müssen mit Aufrufen von `Monitor.enter()` und `Monitor.exit()` eingerahmt werden.

```
Monitor.enter();
  <kritischer Bereich>
Monitor.exit();
```

Der Aufruf von `Monitor.enter()` belegt den Monitor, falls dieser noch frei ist, ansonsten wird der aufrufende Thread angehalten und konzeptuell in eine Warteschlange eingereiht.

`Monitor.exit()` gibt den Monitor wieder frei und benachrichtigt die in diesem Monitor wartenden Threads. Eine Implementierung in Pseudo-Java könnte etwa folgendermaßen aussehen:

```
void Monitor.enter() {
  while (this.entered) {
    <Thread in Warteschlange dieses Monitors einreihen>
  }
  this.entered=true;
}

void Monitor.exit() {
  this.entered=false;
  <in diesem Monitor wartende Threads aufwecken>
}
```

Das Problem bei dieser Implementierung in Java selbst bestünde darin, daß hier auch der Zustand eines Objekts verändert wird, nämlich der des Monitorobjekts. Dabei kann es nach obigem Schema zu genau denselben Inkonsistenzen bei der Interaktion von mehreren Threads kommen. Der Grundbaustein der Synchronisierung, der Monitor, ist also nicht in Java selbst implementierbar. Java stellt daher ein eigenes Sprachkonstrukt zur Verfügung, um Monitore zu verwenden.

Die erste Frage nach der Erzeugung eines Monitors ist sehr einfach zu beantworten: Einen Monitor erzeugt man direkt überhaupt nicht, jedem Java-Objekt und jeder Java-Klasse ist automatisch ein eigener Monitor zugeordnet. Benötigt man dennoch ein einzelnes Monitorobjekt, so kann man einfach eine Instanz der Klasse `java.lang.Object` erzeugen.

Einen kritischen Bereich durch einen Monitor zu schützen, ist jetzt auf zwei verschiedene Arten möglich:

- Man fügt der Deklaration einer Methode, die Teil eines kritischen Bereichs ist, den Modifier `synchronized` hinzu. Das bewirkt, daß bevor ein Thread den Rumpf der Methode betreten darf, dieser zuerst den Monitor der aktuellen Instanz, für die die Methode aufgerufen wurde, erwerben muß. Falls es sich um eine statische Methode handelt, wird der Monitor der Klasse erworben.

  ```
  synchronized <resultType> <methodName> ( <parameterList> ) {
    <methodBody>
  }
  ```

kritischen Bereich mit einer `synchronized` Anweisung umschließt. Das Objekt, dessen Monitor angefordert wird, wird dabei in Klammern angegeben. Hier ist nur eine Instanz einer Klasse, nicht die Klasse selbst erlaubt.

```
synchronized( <objectReference> ) {
  <blockBody>
}
```

Das Schlüsselwort `synchronized` stellt die Basis für alle in Java realisierbaren Koordinations- und Kooperationsmechanismen dar. Es wird durch das Schlüsselwort `synchronized` aber nicht garantiert, daß der entsprechende Codeblock atomar ausgeführt wird, das heißt, daß ein Taskwechsel innerhalb eines solchen Blocks unterbleibt. Methoden, die nicht mit `synchronized` geschützt sind, können also trotzdem parallel zu einer geschützten Methode ausgeführt werden. Das Schlüsselwort `synchronized` steht also nicht in direkter Beziehung zum Schutz innerhalb des Kerns eines Betriebsystems, der atomare bzw. unterbrechungsfreie Ausführung eines Aufrufs garantiert.

4.3 Asynchroner Variablenzugriff

Wenn mehrere Threads gleichzeitig auf eine Variable zugreifen, die nur in mehreren Schritten gelesen werden kann (double, long), so könnte passieren, daß zu dem Zeitpunkt, in dem der eine die Variable gerade zur Hälfte geschrieben hat, das System zum nächsten Thread umschaltet, der jetzt die ganze Variable ließt. Das Ergebnis der Leseoperation wäre natürlich Unsinn, da zur Hälfte der alte und zur anderen Hälfte der neue Zustand gelesen würde.

Ist die Variablendeklaration mit dem Modifier `volatile` versehen, so sorgt der Compiler dafür, daß Zugriffe auf diese Variable nach außen hin immer als atomar beobachtet werden. Außerdem bewirkt der Modifier `volatile`, daß die entsprechende Variable vor jeder Operation aus dem Speicher gelesen und nach jeder Operation auch wirklich wieder in den Speicher geschrieben wird. Das ist insbesondere bei einem zugrundeliegenden Multiprozessorsystem entscheidend, da dort ja Operationen echt gleichzeitig durchgeführt werden können.

Bei der momentanen Implementierung der virtuellen Java Maschine (1.0 Beta/Linux) scheint die Benutzung des Schlüsselworts `volatile` jedoch vollkommen überflüssig zu sein, da nie während eines Variablenzugriffs zwischen den verschiedenen Threads umgeschaltet wird.

In der derzeitigen Implementierung sieht es eher so aus, als ob nur zu einem neuen Thread umgeschaltet wird, wenn der aktuelle Thread sich um einen Monitor bewirbt, der gerade besetzt ist, oder explizit der laufende Thread mit `Thread.sleep(msec)` für eine ganz bestimmte Zeit eingeschläfert wird oder aber die Threadumschaltung explizit mit `Thread.yield()` erzwungen wird.

Das wiederum bedeutet natürlich auch, daß ein Thread, der in einer Endlosschleife hängt, in welcher er keinen Monitor anfordert, oder nicht ausdrücklich mit `Thread.yield()` auf seine Vorfahrt verzichtet, das gesamte System blockiert, da nie mehr ein Threadwechsel vorgenommen wird.

4.4 Koordination

Die Grundform der Koordination ist ein Signalobjekt, welches mit zwei Operationen ähnlich einem Monitor die Benachrichtigung mehrerer Threads untereinander ermöglicht. Ein Thread

Der wartende Thread wird angehalten und in die Warteschlange des Signalobjekts eingereiht, bis eine Benachrichtigung von einem anderen Thread eintrifft. Mit `Signalobjekt.notify()` kann ein Thread die wartenden Threads in einem Signalobjekt benachrichtigen, was das Weiterlaufen der benachrichtigten Threads bewirkt.

Wie die Monitore müssen Signalobjekte ebenfalls nicht extra erzeugt werden, da jedes Java-Objekt gleichzeitig auch ein Signalobjekt darstellt, d.h. die Klasse `java.lang.Object`, die Basisklasse aller in Java definierbaren Klassen, stellt Methoden zum Warten auf Benachrichtigungen und zum Benachrichtigen zur Verfügung:

```
public final void java.lang.Object.wait()
public final void java.lang.Object.wait(long timeout)
```

Beim Aufruf einer dieser Methoden wird der aufrufende Thread angehalten und in die Liste wartender Threads des Objekts eingefügt. Der Thread kann erst wieder weiterlaufen, wenn ein anderer Thread eine Benachrichtigung ausspricht oder bei Verwendung der zweiten Form die maximale Wartezeit überschritten ist.

Beim Aufruf von `anObject.wait()` muß sich der der aufrufende Thread im Besitz des Monitors des Objekts `anObject` befinden. D.h. ein Aufruf von `anObject.wait()` darf nur in einer `synchronized` Methode von `anOject` selbst erfolgen, oder der Monitor des Objekts `anObject` muß explizit durch `synchronized(anObject)` $\{\ldots\}$ erworben worden sein.

Der Besitz des Monitors ist notwendig, um konsistente Verwaltung der wartenden Threads zu gewährleisten. Während der Wartezeit allerdings wird der Monitor freigegeben. Nach der Benachrichtigung und vor dem Weiterlaufen wird der Monitor erneut erworben. Würde der Monitor während der Wartezeit nicht freigegeben, so könnten auch nie mehr als ein Thread in einem Signalobjekt auf Benachrichtigung warten, da der wartende Thread ja immer den Monitor besitzen würde, der notwendig ist, um den Aufruf von `wait()` duchzuführen.

```
public final void java.lang.Object.notify()
public final void java.lang.Object.notifyAll()
```

`notify()` benachrichtigt genau einen, `notifyAll()` benachrichtigt alle Threads, die sich in der Warteschlange des Objekts befinden, falls die Warteschlange nicht leer ist. Befinden sich mehrere Threads in der Warteschlange, so hängt es bei `notify()` von der jeweiligen Implementierung ab, welcher Thread aus der Warteschlange entlassen wird. Diese Methode sollte also nur zur Anwendung kommen, wenn durch den Programmkontext sichergestellt werden kann, daß sich nur ein Thread in der Warteschlange befindet, da sich sonst das Verhalten des Programms von Java-Umgebung zu Java-Umgebung ändern kann. Sollen alle Threads benachrichtigt werden, muß `notifyAll()` verwendet werden.

4.4.1 Guards

Guards sind in Java selbst formulierbare Programmkonstrukte, die mit Hilfe der oben besprochenen Koordinationsmöglichkeiten das Zutreffen von Vorbedingungen für die Ausführung bestimmter Codeabschnitte sicherstellen.

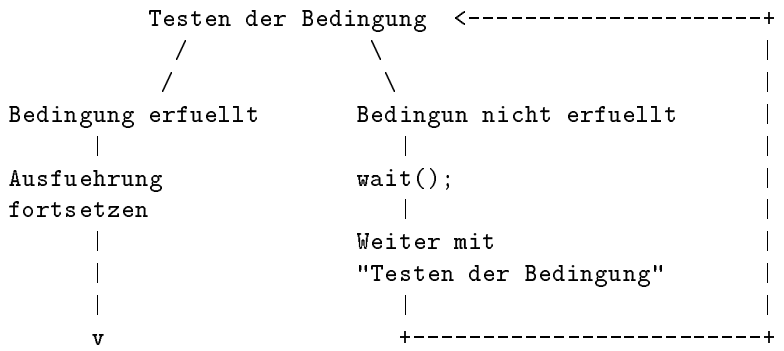
Beim Nichtzutreffen einer für einen bestimmten Codeabschnitt notwendigen Vorbedingungen sind prinzipiell mehrere Strategien denkbar:

- Verlassen der Prozedur durch Auslösen einer Ausnahmebedingung oder Zurückliefern eines Fehlercodes
- Warten, bis die Vorbedingung zutrifft.

In streng sequentiellen Programmen sind nur die ersten beiden Möglichkeiten duchführbar, wobei die erste jedoch wenig empfehlenswert ist. Da in einem streng sequentiell ablaufenden Programm eine einmal nicht zutreffende Vorbedingung bis in alle Ewigkeit unerfüllt bleiben wird (da ja keine anderen Aktivitäten nebenher ablaufen, die daran etwas ändern könnten), ist es wenig sinnvoll auf die Erfüllung der Vorbedingung zu warten.

Anders stellt sich der Sachverhalt bei parallelen Programmen dar. Hier kann es durchaus sinnvoll sein, die Erfüllung einer solchen Vorbedingung abzuwarten, wenn die Aussicht besteht, daß ein anderer Thread in Kürze die Voraussetzungen für die Erfüllung der Bedingung schaffen wird.

Prinzipiell hat ein Guard folgendes Aussehen:



Bei der Implementierung der Klasse muß jetzt noch darauf geachtet werden, daß nach jeder Änderung des internen Objektzustandes, die eventuell die Bedingung erfüllen könnte, alle in diesem Objekt wartenden Threads mittels eines Aufrufs von `notifyAll()` benachrichtigt werden.

```
Veraenderung des Objektzustandes

notifyAll()
```

Man sieht schnell ein, daß sowohl dieser Wächterabschnitt und alle anderen Codeabschnitte, die die zu testende Bedingung verändern können, mit dem `synchronized` Schlüsselwort geschützt sein müssen. Sonst könnte die Bedingung für den prüfenden Thread erfüllt sein, er würde die Ausführung fortsetzen und im selben Moment könnte ein anderer Thread durch Manipulation des Zusstands bewirken, daß die Bedingung nicht mehr erfüllt und damit die Vorbedingung für die Ausführung nicht mehr gegeben wäre.

Das folgende Beispiel zeigt die Implementierung der Klasse `LimitedBuffer`, die einen größenbeschränkten Puffer für Objekte zur Verfügung stellt. Der Puffer soll zur Kommunikation zwischen mehreren Prozessen dienen, die entweder Objekte in den Puffer hineinlegen oder wieder aus ihm entnehmen können sollen. An zwei Stellen wird die Konstruktion des Guards verwendet: Ein Thread darf aus dem Puffer nur dann ein Objekt entnehmen, wenn der Puffer nicht leer ist, und ein Thread darf in den Puffer ebenfalls nur ein Objekt einfügen, wenn er nicht voll ist.

```
 * Eine Klasse, die einen Puffer von beschraenkter Groesse darstellt.
 * Diese Klasse soll Kommunikation zwischen mehreren Threads ermoeglichen.
 * Threads koennen entweder Objekte in den Puffer schreiben oder wieder
 * aus diesem entnehmen.
 */
public class LimitedBuffer {
  /** Groesse des Puffers */
  int    _size;

  /** array, in dem die Objekte zwischengespeichert werden */
  Object _buffer[];

  /** Stelle im Puffer-array, die als naechstes ausgelesen wird */
  int _nextRead;

  /** Stelle im Puffer-array, die als naechstes beschrieben wird */
  int _nextWrite;

  /*
   * Es gelten folgende Klasseninvarianten:
   *   Der Puffer ist entweder leer, oder alle Warteplaetze
   *     _buffer[_nextRead .. (_nextWrite-1) modulo _size] sind duch
   *     gepufferte Objekte belegt.
   *   Der Puffer ist genau dann leer wenn gilt _nextRead==_nextWrite.
   *
   * Daraus folgt, dass immer ein Warteplatz im Puffer frei bleibt, da
   *   sonst nach dem letzten Einfuegen wieder _nextWrite==_nextRead
   *   gelten wuerde. Deshalb muss ein Platz mehr im Puffer-array belegt
   *   werden, als der Puffer Warteplaetze enthalten soll.
   */

  /**
   * Erzeugt einen neuen <em>LimitedBuffer</em> mit einer Kapazitaet
   * von <em>size</em> Objekten.
   */
  public LimitedBuffer(int size) {
    _size      = size+1;
    _buffer    = new Object[_size];
    _nextRead  = 0;
    _nextWrite = 0;
  }

  /** Stellt fest, ob der Puffer leer ist. */
  public synchronized boolean empty() {
    return (_nextRead == _nextWrite);
  }

  /** Stellt fest, ob der Puffer voll ist. */
  public synchronized boolean full()  {
    return ((_nextWrite+1)%_size == _nextRead);
  }

  /**
   * Liest ein Objekt aus dem Puffer aus, falls der Puffer nicht leer
```

```
     * in den Puffer eingefuegt wird.
     * @return Das aus dem Puffer gelesene Objekt
     */
    public synchronized Object read() {
      while (empty()) {
        try wait();  catch (InterruptedException e) {}
      }
      Object result = _buffer[_nextRead];
      _nextRead = (_nextRead+1)%_size;

      notifyAll();

      return result;
    }

    /**
     * Schreibt ein Objekt in den Puffer, falls dieser nicht voll ist.
     * Andernfalls blockiert der Thread und wartet, bis ein Objekt aus dem
     * Puffer entnommen wird.
     * @param obj Das in den Puffer zu schreibende Objekt
     */
    public synchronized void write(Object obj) {
      while (full()) {
        try wait(); catch (InterruptedException e) {}
      }
      _buffer[_nextWrite] = obj;
      _nextWrite = (_nextWrite+1)%_size;

      notifyAll();
    }
  }
```

Die obige Implementierung verwendet mit `synchronized` geschützte Methoden in Zusammenhang mit Wächterausdrücken der Form

```
while (empty()) {
  try wait(); catch (InterruptedException e) {}
}
```

um die Konsistenz des Objektzustandes zu erhalten und Threads anzuhalten, die aus einem leeren Puffer lesen oder in einen vollen Puffer hineinschreiben wollen.

Allerdings sind noch einige Dinge an obiger Implementierung auszusetzen:

- Nach jedem Schreiben und Lesen werden alle wartenden Threads aufgeweckt, obwohl immer garantiert nur ein einziger fortsetzen kann. Nach einem Schreibzugriff z.B. kann höchstens eine wartende Leseanforderung ausgeführt werden.

  Würde man etwa `notifyAll()` einfach durch `notify()` ersetzen, so wäre der Effekt bei mehreren wartenden Anforderungen abhängig von der zugrundeliegenden Java-Implementierung, da in der Spezifikation keine Aussage darüber getroffen wird, welcher Thread bei mehreren wartenden durch `notify()` aufgeweckt wird.

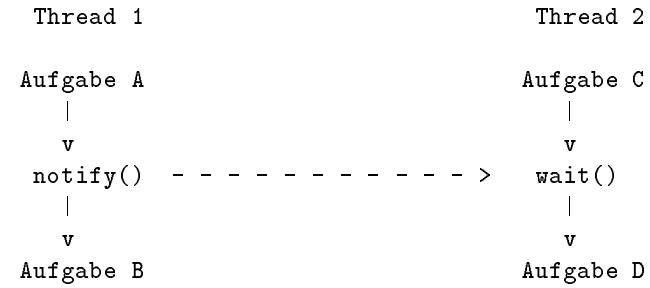
die Möglichkeit, als nächstes fortzusetzen, jedoch wird auch hier nicht garantiert, daß jeder irgendwann einmal zum Zuge kommt. Die Reihenfolge wird gänzlich dem Zufall, bzw. dem Java-Task-Sheduler überlassen.

Um eine garantierte Bearbeitungsreihenfolge der Anfragen zu erreichen, z.B. „am längsten wartende Anfrage zuerst“ oder „Anfrage mit höchster Priorität zuerst“ müßte eine eigene Threadwarteschlange implementiert werden.

4.4.2 Erweiterte Koordinationsmechanismen

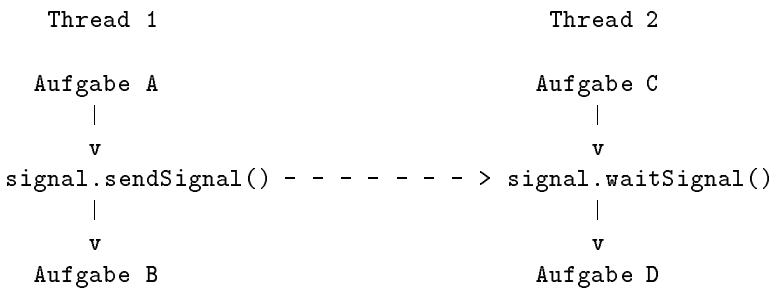
In den obigen Abschnitten wurden die Java-eigenen Koordinationsmechanismen (Monitore und primitive Signalobjektfunktionen) ausschließlich dazu eingesetzt, Inkonsistenzen bei Zustandsänderungen zu vermeiden und Vorbedingungen sicherzustellen.

Koordination von zwei Aktivitäten kann aber durchaus komplexer formulierte Zusammenhänge beschreiben. Die Bedingung „Thread 1 muß Aufgabe A vollständig abgearbeitet haben, bevor Thread 2 mit Aufgabe D beginnt“ läßt sich nicht durch einfache Verwendung von `wait()` und `notify()` erzwingen:



Denn wenn Thread 1 die Aufgabe A abgearbeitet hat, bevor Thread 2 den Aufruf von `wait` erreicht hat, bleibt `notify()` ohne Effekt, da keine Threads in der Warteschlange stehen. Der Aufruf von `notify()` wird nämlich nicht gespeichert, so daß Thread 2 nach Erledigung von Aufgabe C für alle Zeiten auf eine Benachrichtigung warten würde, die schon längst gegeben und wieder vergessen wurde.

An dieser Stelle ist die Verwendung eines eigenständigen Koordinationsobjekts angezeigt, das unter Verwendung der Vorgehensweisen der vorangegangenen Abschnitte, sehr einfach in Java implementierbar ist.



Die folgende Klasse `Signal` implementiert ein eigenständiges Signalobjekt, das mehrere eingehende Signale zwischenspeichern kann. Threads, die auf ein noch nicht gesendetes Signal warten,

ge, mit der Signale an wartende Threads verteilt werden festgelegt: Immer der Thread, der sich zuerst in die Warteschlange eingereiht hat, wird auch als erster mit einem Signal bedient. Leider müssen beim Senden eines Signals immer alle Threads aufgeweckt werden aber nur derjenige, der am Kopf der Schlange wartet kann weiterarbeiten.

```
import java.lang.Thread;
import Queue;            // nicht im "Java-Lieferumfang" enthalten

public class Signal {
  /** Anzahl der zur Verfuegung stehenden Signale */
  int     _signals;

  /** Schlange der wartenden Threads */
  Queue   _queue;

  /** Klasseninvariante:
    * _signals==0 or _queue.empty()
    * <=> Waere die Warteschlange nicht leer und ein Signal laege vor,
    *     so haette ein Thread schon laengst weiterlaufen koennen.
    */

  /** Erzeugt ein neues Signalobjekt, in dem schon 'signals' Signale
    * zur Verfuegung stehen.
    */
  public Signal(int signals) {
    _signals = signals;
    _queue   = new Queue();
  }

  /** Fuehrt eine Signalisierung durch. Steht ein Thread in der
    * Warteschlange, so wird dieser freigegeben, ansonsten wird das
    * Signal aufbewahrt.
    */
  public synchronized void sendSignal() {
    _signals++;
    if (! _queue.empty()) notifyAll();
  }

  /** Wartet auf ein Signal. Falls schon ein Signal eingegangen ist,
    * wird dieses sofort konsumiert, ansonsten wird der Thread in
    * die Liste der wartenden Threads eingereiht.
    */
  public synchronized void waitSignal() {
    if (_signals>0) {
      _signals--;
    } else {
      Thread running = Thread.currentThread();
      _queue.push(running);

      do {
        try
          wait();
        catch (InterruptedException e) {}
      } while (_queue.top() != running);
```

```
        _queue.pop();
      }
    }
  }
```

Müssen wirklich bei jedem Signal alle Threads benachrichtigt werden?

- Die Methode `waitSignal()` muß durch einen Monitor geschützt sein, da sie Zustandsänderungen durchführt.
- Gleichzeitig muß aber der Monitor wieder freigegeben werden, sobald der Thread beginnt auf ein Signal zu warten, sonst könnte immer nur ein Thread gleichzeitig auf ein Signal warten.
- Dies erledigt die Methode `wait()` automatisch, aber auch nur für den Monitor des Objekts, für das `wait()` aufgerufen wurde.
- Alle Threads müssen letztendlich im Signalobjekt selbst warten, da sonst keine Möglichkeit besteht, den Monitor, der beim Betreten der `synchronized` Methode erworben wurde, zwischenzeitlich wieder freizugeben.
- Das wiederum hat zur Folge, daß bei jedem eingehenden Signal immer alle Threads aufgeweckt werden müssen, sie warten ja alle im gleichen Objekt.

  Ein Ausweg wäre die Benachrichtigung eines Threads über einen Interrupt, ausgelöst über die Methode `void Thread.interrupt()`, die aber in der augenblicklichen Version von Java nicht implementiert ist.

4.4.3 Dead-Locks

Ein Dead-Lock tritt dann auf, wenn mehrere Threads gegenseitig darauf warten, daß jeweils ein anderer ein bestimmtes Signal gibt. Da aber alle auf einen anderen warten, kann keiner das erwartete Signal auslösen, und das Programm blockiert.

Die einfachste Situation, bei der unter ungünstigen Bedingungen ein Dead-Lock auftreten kann, sieht folgendermaßen aus: Thread1 versucht zuerst den Monitor des Objekts `SO1` zu erwerben und danach den Monitor von `SO2`. Thread2 geht gerade umgekehrt vor. Bei der Abarbeitungsreihenfolge

- Thread1 erwirbt den Monitor von SO1
- Thread2 erwirbt den Monitor von SO2
- Thread1 wartet auf die Freigabe von SO2
- Thread2 wartet auf die Freigabe von SO1

tritt der Dead-Lock auf. Keiner der beiden Threads kann fortsetzen, da beide auf den jeweils anderen warten.

```
synchronized(SO1) {
  synchronized(SO2) {
    .
    .
    .
  }
}
```

```
synchronized(SO2) {
  synchronized(SO1) {
    .
    .
    .
  }
}
```

In Java ist kein Mechanismus integriert, der das Auftreten von Dead-Locks zur Laufzeit automatisch erkennt, obwohl das prinzipiell möglich wäre.

Man kann aber auf ganz einfache Weise verhindern, daß in solchen Situationen Dead-Locks auftreten, indem man das Belegen von Resourcen - nichts anderes ist ja der Erwerb eines Monitors - ordnet. Diese Vorgehensweise ist auch als Resource-Ordering bekannt.

In obigem Beispiel müßte also die Reihenfolge des Monitorerwerbs in beiden Codeabschnitten die gleiche sein, nur so kann die oben geschilderte Situation eines Dead-Locks nicht auftreten.

4.5 Zusammenfassung

Java bietet prinzipiell alle Möglichkeiten, um sehr einfach auf hohem Niveau mit nebenläufigen Threads zu programmieren. Die Implementierung der virtuellen Java-Maschine in der vorliegenden Version (1.0 Beta/Linux) zeigt allerdings nicht die Eigenschaften, die man von einer multithreaded Umgebung erwartet. Der „Eigenbau“ von komplexeren Koordinations- und Kommunikationsobjekten ist möglich, aufgrund der mangelnden vollständigen Kontrolle über die Java-eigenen Monitore, oder wegen des Fehlens der Möglichkeit, Programmabschnitte als nur atomar ausführbar zu kennzeichnen, nicht immer besonders effizient.

Java bietet eine Reihe von vorgefertigten Bibliotheken an:

java.applet	Ermöglicht das Ausführen von Java-Programmen im WWW.
java.awt	Enthält Graphik, Kontrollelemente, Dialoge, Fenster und Layoutmanager
java.io	File-I/O, Stream-I/O, ...
java.lang	Strings, nummerische Werte, Threads, mathematische Funktionen ...
java.net	Klassen für Netzwerkkommunikation
java.util	Datenstrukturen (Vektor, Aufzählung, ...) sowie andere nützliche Klassen

In all diesen Bibliotheken wird man allerdings vergeblich nach Konzepten suchen, die sich mit der Ausgabe auf Druckern beschäftigen. Weder die Streams noch das AWT bieten eine Lösung, obwohl sie von den Grundkonzepten her dafür geeignet wären.

Daran ist erkennbar, daß die Bibliotheken von Java immer noch weiter verbessert werden können. Im Gegensatz zu den Sprachkonzepten, die sehr ausgereift erscheinen.

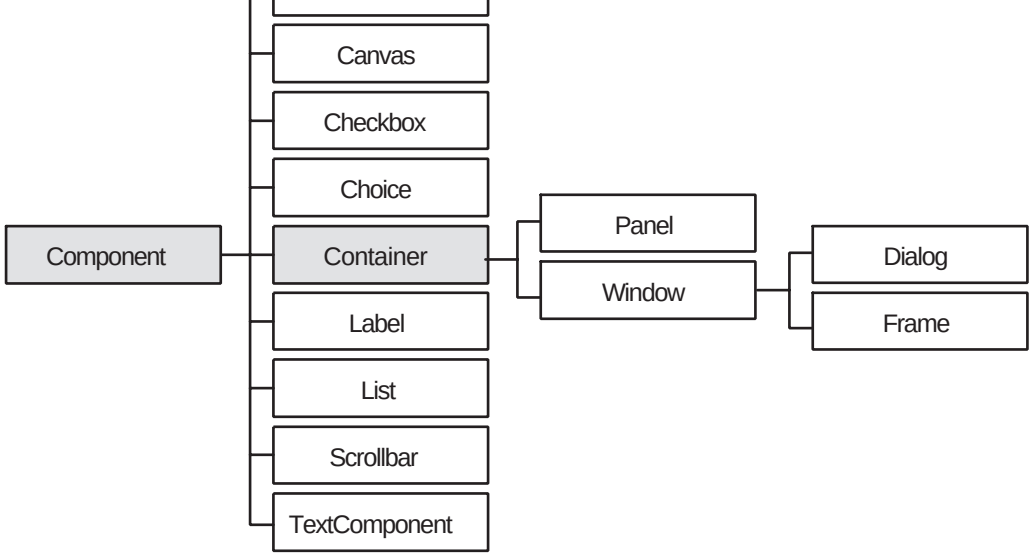
Im folgenden werden die Bibliotheken java.awt, java.io und java.net genauer behandelt.

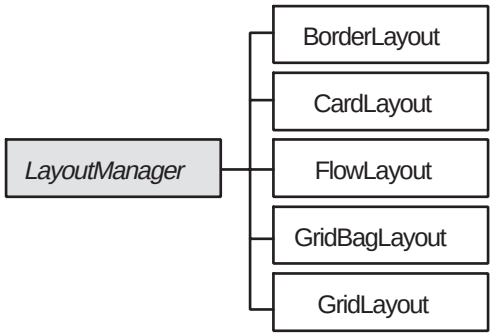
5.1 Das AWT

Das Abstract Windowing Toolkit (AWT) definiert eine Reihe von Klassen, die in zwei Bereiche gegliedert werden können:

- Fensterbausteine und deren Steuerung (Kontrollelemente, Dialoge, Layoutmanager, ...)
- Graphische Zeichenroutinen (Kreise, Polygone, Fonts, Bilder, ...)

Die nachfolgende Abbildung zeigt die wichtigsten Klassen des AWT (ohne die Klassen der Graphik). Grau hinterlegte Klassen sind abstrakte Klassen, kursiv gesetzte Klassennamen stellen Interfaces dar.





5.1.1 Component und Container

Jede GUI-Komponente ist im AWT von der abstrakten Klasse *Component* abgeleitet. Diese Klasse beschreibt den Umgang mit Ereignissen und mit GUI-Komponenten (z.B.: Verschieben, Vergrößern, usw.). Ein spezielles *Component* ist der *Container*. Dieser hat zusätzlich die Eigenschaft selbst mehrere *Components* verwalten zu können.

Das AWT bietet sämtliche Kontrollelemente eines modernen Fenstersystems an. Dazu gehören Buttons, Checkboxen, Listboxen, Eingabefelder, Textausgaben, ...

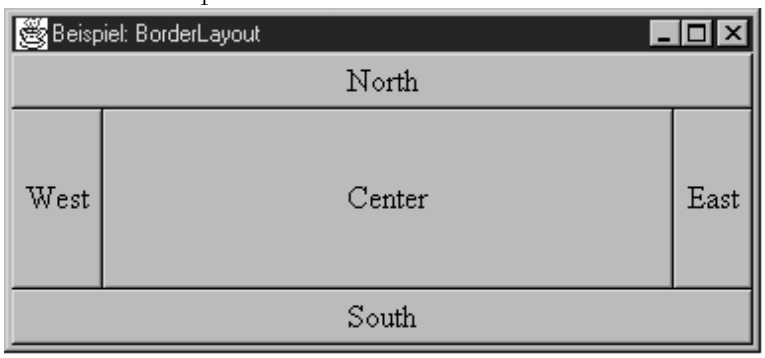
Diese Elemente werden mittels der *Container* zusammengefaßt. *Container* können z.B. Dialoge oder Fenster sein. Es können aber auch mehrere Elemente zunächst zu einer Einheit (*Panel*) zusammengefaßt werden, die dann mit anderen solchen Einheiten in ein Fenster oder einen Dialog eingesetzt werden.

Vergleicht man den Aufbau des AWT mit dem von Motif, erkennt man einige Gemeinsamkeiten. Beide Klassenhierarchien kennen eine Basis-Komponente (*Component* bzw. *Core*). Von dieser Basiskomponente werden sowohl die einfachen Kontrollelemente (in Motif *XmPrimitive*) als auch die *Container* (in Motif *Composite*) abgeleitet. Java und Motif kennen auch einen Manager, der dafür verantwortlich ist, wie die Kontrollelemente eines *Containers* angezeigt werden (Positionierung, Größe, Verhalten bei Größenveränderung des *Containers*, ...). In Java heißen diese Manager Layoutmanager, Motif kennt sie als Ableitungen von der Klasse *XmManager*.

Java bietet das Interface *LayoutManager* an. Klassen, die dieses Interface implementieren, dienen als Layoutmanager dazu anzugeben, wie in einem *Container* die *Components* plaziert werden. Java bietet neben dem Interface auch einige Klassen, die bereits vorgefertigte Layoutmanager darstellen und für die meisten Zwecke ausreichen. Daneben bietet Java aber auch die Möglichkeit eigene Manager zu definieren. Die folgenden Layout-Manager werden von Java angeboten:

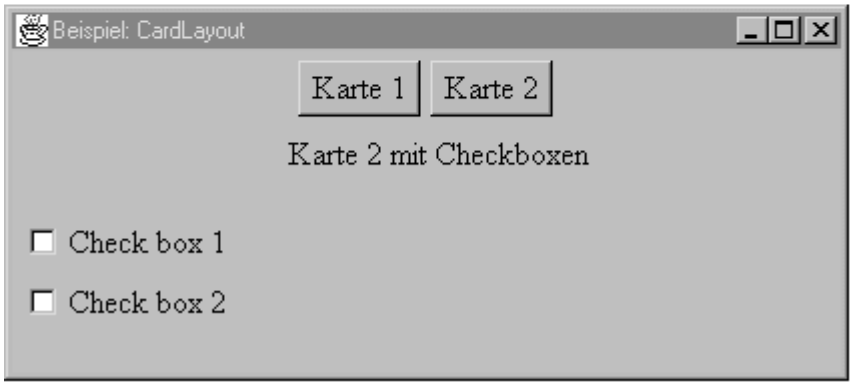
- BorderLayout

  Komponenten können über Angaben in Himmelsrichtungen (*South*, *East*, *North*, *West* und *Center*) in dem *Container* plaziert werden.



- CardLayout

  Die dem *Container* zugeordneten Komponenten werden nur einzeln angezeigt. Mit Hilfe der Methoden des Layoutmanagers können bestimmte „Karten“ im Anzeigefeld des *Containers* sichtbar gemacht werden. Als Komponenten des *Containers* dienen in der Regel wiederum *Container*, durch deren Komponenten dann der Inhalt der einzelnen Karten bestimmt wird. Dieses Layout ist vergleichbar mit „Registerdialogen“, bei denen der Anwender zwischen den einzelnen Seiten eines Dialoges mittels einer Liste oder Karteikartenreitern wechseln kann. *CardLayout* impliziert aber keine Kontrollelemente, die dem Benutzer das Wechseln der Seiten ermöglichen.

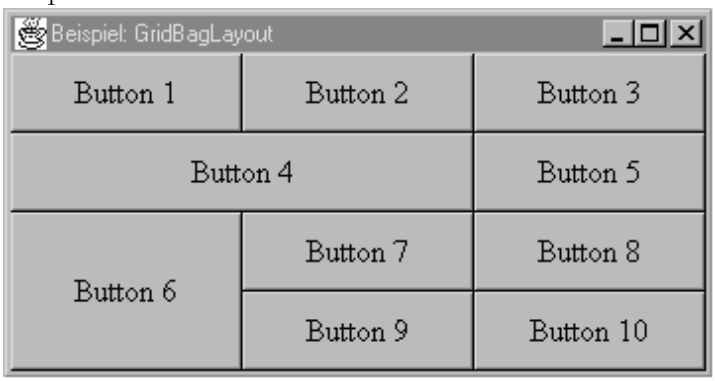


- FlowLayout

  Die Komponenten werden in Zeilen angeordnet. Der Manager fügt Komponenten solange einer Zeile hinzu (von links nach rechts) bis diese voll ist. Dann beginnt er mit der nächsten Zeile.

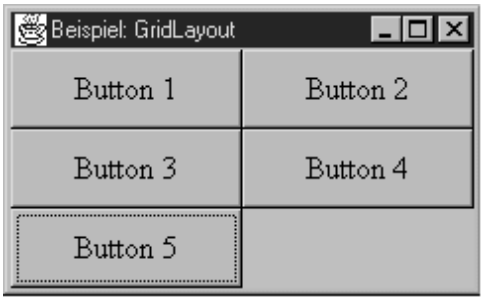
- GridBagLayout

  Der *Container* wird in Spalten und Zeilen eingeteilt. Komponenten müssen direkt an eine Zellen-Position gesetzt werden. Sie können sich auch über mehrere Spalten und/oder Zeilen ausdehnen. Die Spaltenbreiten und Zeilenhöhen können variieren.



- GridLayout

nenten werden aber der Reihe nach in die Zellen gesetzt (von links nach rechts und von oben nach unten). Die Breite bzw. die Höhe der einzelnen Zellen ist immer gleich.



Motif bietet ähnliche Manager an. Der Vorteil dieser Manager liegt darin, daß die Darstellung unabhangig von der Größe des *Containers* ist und damit auch unabhängig von der darstellenden Hardware (Pixelauflösung einer Graphikkarte, usw.). Auch Zeichensatzgrößen stellen so keine Probleme mehr da. Fenstersysteme, die dieses Konzept nicht verwenden (wie z.B. Windows) legen die Verantwortung der Plazierung der Komponenten in die Hand des Programmierers. Dieser muß beispielsweise bei größenveränderlichen Dialogen selbst entsprechende Aktionen zum Vergrößern der Komponenten vorsehen.

5.1.3 Die verschiedenen Container-Klassen

Java bietet die folgenden *Container* an:

- Panel

  Ein *Panel* wird immer als Komponente eines anderen *Containers* verwendet. Es dient der Strukturierung von Kontrollelementen und kann die ihm zugeordneten Komponenten durch einen eigenen Layoutmanager verwalten (Voreinstellung ist *FlowLayout*). Im Gegensatz zu den Klassen, die von *Window* abgeleitet werden, stellt *Panel* kein eigenes Fenster im Sinne der Fensterverwaltung des Betriebssystems dar.

- Window

  Diese Klasse stellt ein „echtes“ Fenster (im Sinne der Fensterverwaltung) dar. Dieses Fenster kann z.B. auch Ereignisse hervorrufen wie `WINDOW_DESTROY`, `WINDOW_MOVE`, usw. Jedoch hat es keinen Rahmen und kein Menü. Diese müssen ihm hinzugefügt werden.

- Dialog

  Ein spezielles Fenster, das die typischen Dialogeigenschaften hat. Es kann modal sein (Benutzeraktionen in anderen Fenstern sind nicht möglich), eine Titelzeile haben, ...

  Der Dialog benutzt als Voreinstellung den Layout-Manager *BorderLayout*.

- Frame

  Dies ist ein typisches Fenster, das normalerweise das Fenster einer Anwendung darstellt. Es kann eine Titelzeile haben, ein Menü, ein Icon und das Aussehen des Mauszeigers bestimmen, wenn dieser sich innerhalb des Fenster befindet. Diese Klasse implementiert einen *MenuContainer* (siehe 5.1.5), wodurch das Anzeigen und Verwalten eines Menüs ermöglicht wird.

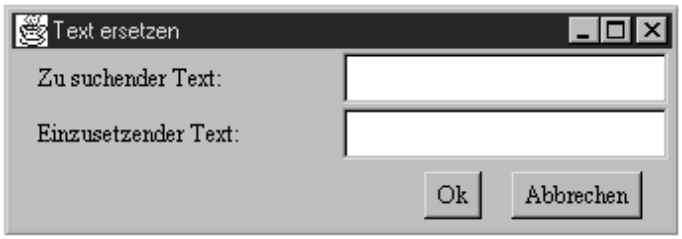
Ein spezieller Dialog, der zur Abfrage einer zu öffnenden oder zu speichernden Datei dient.

5.1.4 Verwendung von Kontrollelementen

Einzelne *Components* werden einem *Container* mittels der Methode *add* zugeordnet. Dieser Methode des *Containers* wird die Komponente übergeben. Um die Komponente plazieren zu können, wird der *Container* den Layoutmanager aufrufen.

In der Regel reicht es nicht, in einem Fenster nur einen Layoutmanager zu verwenden. Vielmehr muß man hierarchisch vorgehen und zunächst die Kontrollelemente in sogenannten *Panels* gruppieren. Diese Gruppen können eigene Layoutmanager verwenden. Die einzelnen Gruppen werden dann (eventuell nach weiterer Gruppierung) dem Fenster mittels *add* zugeordnet. Das Fenster wendet dann seinen eigenen Layoutmanager an, um die Gruppen zu plazieren.

Als Beispiel sei hier ein Dialog gegeben, der das Ersetzen einer Zeichenkette durch eine andere in einem größeren Text abfragen soll. Die nachfolgende Abbildung zeigt den Dialog.



Der Dialog wird zunächst in zwei Bereiche aufgeteilt. Der oberere Teil soll die Eingabezeilen enthalten, der untere die Buttons. Hierfür werden zwei *Panels* verwendet: *panelButton* und *panelEdit*. Beide *Panels* können in dem Dialog mittels eines *BorderLayouts* plaziert werden. Dies geschieht für *panelButton* mit der Angabe *South* und für *panelEdit* mit der Angabe *Center*. Die *Panels* sind nun so plaziert, daß die Buttons unter den Eingabezeilen erscheinen.

Die einzelnen Buttons und Eingabezeilen müssen noch in ihren Bereichen angeordnet werden. Eine Button-Reihe kann durch Verwendung des LayoutManagers *FlowLayout* realisiert werden. Für die Eingabefelder und deren Beschreibungstexte ist *GridLayout* eine einfache Alternative zu *GridBagLayout*, bei dem aber zusätzlich die Größenveränderlichkeit des Dialoges wesentlich sinnvoller gehandhabt wird. Bei Vergrößerung des Dialoges wird *GridLayout* die Eingabefelder und die Beschreibungstexte immer gleich behandeln. Dadurch werden die Eingabefelder immer höher und zwischen Beschreibungen und Eingabefeldern entsteht eine unschöne Lücke. *GridBagLayout* gibt dem Programmierer Eingriffsmöglichkeiten in das Verhalten bei Größenänderung, so daß eine angenehmere Darstellung bewahrt wird.

Der Programmtext könnte etwa so aussehen:

```
import java.awt;

public class ReplaceDialog extends Dialog {

Panel      panelEdit;
Panel      panelButton;
TextField  editSearch;
TextField  editReplace;
```

```
Button     buttonCancel;
Label      labelSearch;
Label      labelReplace;

       public ReplaceDialog(Frame parent) {
              super(parent, "Text ersetzen", true);

              setLayout(new BorderLayout());

              panelEdit=new Panel();
              panelButton=new Panel();

              add(''Center'', panelEdit);
              add(''South'', panelButton);

              panelEdit.setLayout(new GridLayout(2, 2));
              panelButton.setLayout(new FlowLayout(FlowLayout.RIGHT, 15, 5));

              labelSearch=new Label("Zu suchender Text:");
              labelReplace=new Label("Einzusetzender Text:");
              editSearch=new TextField();
              editReplace=new TextField();
              panelEdit.add(labelSearch);
              panelEdit.add(editSearch);
              panelEdit.add(labelReplace);
              panelEdit.add(editReplace);

              buttonOK=new Button(''OK'');
              buttonCancel=new Button(''Abbrechen'');
              panel2.add(buttonOk);
              panel2.add(buttonCancel);
       }

// weitere Methoden, z.B. zum Reagieren auf Ereignisse, ...

}
```

Die Erstellung der Kontrollelemente und deren Plazierung wird im Konstruktor des Dialoges vorgenommen. Dies ist in der Regel der sinnvollste Ort, ist aber nicht so vorgeschrieben.

Der Konstruktor der Klasse *Dialog* verlangt die Übergabe des Vaterfensters, eines Textes, der in der Titelzeile des Dialoges dargestellt wird, und der Angabe, ob der Dialog modal oder nicht modal sein soll. Im Beispiel steht bereits fest, daß der Dialog den Titel „Text ersetzen“ trägt und modal ist.

Nach dem Konstruktoraufruf wird der Layoutmanager für den Dialog festgelegt und die beiden *Panels* erstellt, die dann dem Dialog als Komponenten zugeordnet werden und selbst ihre Layoutmanager festlegen. Das *GridLayout* des *panelEdit* benötigt die Angaben, wieviele Komponenten in einer Zeile und in einer Spalte angeordnet werden sollen. In diesem Fall werden zwei

out soll die Buttons rechtsbündig anordnen und zwischen ihnen 15 Pixel in der Horizontalen Platz lassen, damit die Buttons optisch voneinander getrennt sind. Falls die Buttons nicht in eine Reihe passen sollten (z.B. wenn der Dialog verkleinert wurde), soll außerdem 5 Pixel in der Vertikalen als Freiraum bleiben.

Abschließend werden die Eingabefelder, Beschreibungen und Buttons den entsprechenden *Panels* zugeordnet.

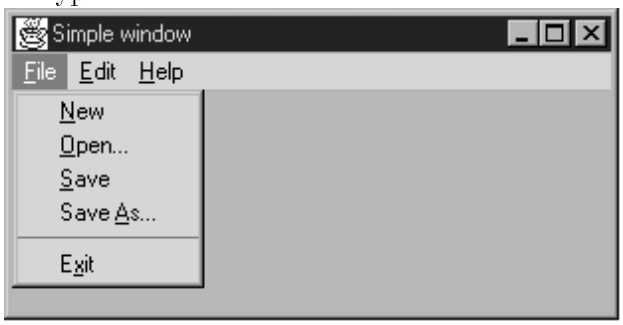
5.1.5 Menüs

Ähnlich wie Komponenten einem *Container* hinzugefügt werden, werden auch Menüeinträge (*MenuComponent*) einem *Menu* oder einem *MenuBar* hinzugefügt. Diese beiden Klassen implementieren das Interface *MenuContainer*, das Methoden zur Verwaltung und Verwendung von Menüeinträgen bereitstellt. Ein *MenuBar* stellt die übliche Menüzeile dar, ein *Menu* ist dagegen ein Pulldownmenü, das bei einer Auswahl eines Menüpunktes in einer Menüzeile aufgeklappt wird.

Ein *MenuItem* ist ein Text, der im Menü dargestellt werden soll. Ein *CheckboxMenuItem* bietet zusätzlich eine Statusanzeige in Form einer Checkbox an, mit deren Hilfe der Menüpunkt darstellen kann, ob eine Option ein- oder ausgeschaltet ist.

Ein *Frame* implementiert ebenfalls einen *MenuContainer*. Dies ermöglicht es diesem Fenster ein Menü darzustellen.

Als Beispiel kann hier ein typischer Aufbau eines Menüs dienen:




```
import java.awt;

public class MyFrameWindow extends Frame {

       public MyFrameWindow() {
              super("Beispiel fr Mens");

              MenuBar mb = new MenuBar();

              fileMenu = new Menu(''\&File'');
              fileMenu.add(new MenuItem(''\&New''));
              fileMenu.add(new MenuItem(''\&Open...''));
              fileMenu.add(new MenuItem(''\&Save''));
              fileMenu.add(new MenuItem(''Save \&As...''));
              fileMenu.addSeparator();
```

```
            mb.add(fileMenu);

            editMenu = new Menu(''\&Edit'');
            editMenu.add(new MenuItem(''\&Undo''));
            editMenu.addSeparator();
            editMenu.add(new MenuItem(''Cu\&t''));
            editMenu.add(new MenuItem(''\&Copy''));
            editMenu.add(new MenuItem(''\&Paste''));
            mb.add(editMenu);

            helpMenu = new Menu(''\&Help'');
            helpMenu.add(new MenuItem(''\&About...''));
            mb.add(helpMenu);

            setMenuBar(mb);
        }

// Weitere Methoden

}
```

Der Aufbau des Menüs geschieht in der Regel im Konstuktor des Fensters. Zuerst wird der Titel des Fensters dem Konstuktor von *Frame* übergeben. Dann wird eine Menüzeile (*mb*) erstellt und die einzelnen Pulldownmenüs werden erstellt und mit Menüpunkten gefüllt.

Zum Schluß wird dem *Frame* mitgeteilt, daß das soeben erstellte Menü, ausgehend von der Menüzeile *mb*, als Menü in dem Fenster des *Frames* verwendet werden soll.

5.1.6 Ereignisse

Die Kontrollelemente erzeugen Ereignisse, die signalisieren, daß der Benutzer eine bestimmte Aktion mit dem Element vorgenommen hat. Auch Menüs erzeugen solche Ereignisse als Reaktion auf die Menüauswahl des Benutzers. Daneben können auch Fenster diverse Ereignisse hervorrufen, die beispielsweise angeben, daß das Fenster verschoben oder verkleinert wurde.

Das AWT kapselt alle Ereignisse mittels der Klasse *Event*. Diese Klasse definiert eine Reihe vordefinierter Ereignisse (Taste gedrückt, Maus bewegt, usw.) als Konstanten. Mittels der Konstante `ACTION_EVENT` können aber auch eigene Ereignisse sowie die meisten Ereignisse, die Menüs und Kontrollelemente erzeugen, verwaltet werden. Neben der Art des Ereignisses kennt die Klasse *Event* auch das Objekt, das das Ereignis erzeugt hat, und den Zeitpunkt, zu dem dies passierte. Weiterhin werden die Mausposition und die gedrückte Taste in einem *Event*-Objekt gespeichert.

Ein Ereignis wird durch ein von *Component* abgeleitetes Objekt erzeugt und dann an eine andere Komponente geschickt (*postEvent*). Die Klasse *Component* definiert eine Methode *handleEvent*, die bestimmte Ereignisse (Tastendruck und Mausaktionen) an vordefinierte Ereignisbehandlungsfunktionen weiterleitet: *mouseDown*, *mouseDrag*, *mouseEnter*, *mouseExit*, *mouseMove*, *mouseUp*, *keyDown*, *keyUp* und *action*. Wird ein Ereignis nicht bearbeitet, so wird es an die Methode *handleEvent* einer anderen Komponente (in der Regel das Vaterfenster) weitergeleitet.

chende Komponente die Methode *handleEvent* oder eine der vordefinierten Behandlungsfunktionen überschreiben. In einer eigenen *handleEvent*-Methode sollte sichergestellt werden, daß für nicht bearbeitete Ereignisse die Methode der Basisklasse aufgerufen wird. Nur so können die vordefinierten Ereignisbehandlungsfunktionen aufgerufen werden oder die Ereignisse an andere Komponenten weitergegeben werden. In den vordefinierten Ereignisbehandlungsfunktionen muß über den Rückgabewert der jeweiligen Methode angegeben werden, ob das Ereignis bearbeitet wurde oder nicht.

In manchen Fenstersystemen wird auch der Wunsch nach dem Neuzeichnen eines Fensterinhaltes über eine solche Ereignisbehandlung mitgeteilt. In Java jedoch nicht. Hier wird dazu die Methode *repaint* derjenigen Komponente aufgerufen, die neu gezeichnet werden soll. Dies resultiert dann in einen Aufruf von *paint* oder *update* der gleichen Komponente. Diese Methoden müssen überschrieben werden, wenn eine Komponente bestimmte Ausgaben machen soll (siehe 5.1.7).

5.1.7 Graphik

Das AWT bietet einige grundlegende Methoden, um graphische Ausgaben in einem Fenster zu machen. Wie die meisten Fenstersysteme wird hierzu ein Objekt verwendet, in dem verzeichnet wird, mit welchen Attributen Ausgaben auf dem Bildschirm geschehen sollen: aktuelle Farbe, aktueller Font, aktueller Zeichenbereich (Clipping), ...

Dieses Objekt ist in der Klasse *Graphics* gekapselt, die weiterhin Methoden zum Zeichnen verschiedener geometrischer Objekte, zur Ausgabe von Text und zur Darstellung von Bildern anbietet.

Einige dieser Methoden werden zusätzlich durch weitere Klassen unterstützt, die den Aufbau der darzustellenden Objekte bzw. Attribute erleichtern: *Image*, *Polygon*, *Font* und *Color*.

Das Zeichnen ist denkbar einfach: Man ermittelt ein *Graphics*-Objekt, indem man das Fenster, in das die Ausgaben stattfinden sollen, nach diesem fragt (*getGraphics*). Danach kann man die Methoden des *Graphics*-Objektes nutzen, um die Ausgaben zu tätigen.

In manchen Fällen kann man auch andere Objekte nach einem *Graphics*-Objekt fragen. Die Klasse *Image* bietet diese Möglichkeit. Dadurch kann man auch Ausgaben in ein Bild machen, so daß diese erst dann sichtbar werden, wenn das Bild selbst angezeigt wird.

Da in einem Fenstersystem Teile eines Fensters durch andere zeitweise verdeckt werden können und deshalb beim Sichtbarwerden solcher verdeckter Bereiche der Inhalt neu angezeigt werden muß, werden Ausgaben in Fenstern in der Regel in speziellen Methoden (*paint* oder *update*) vorgenommen. Diese Methoden sind Bestandteil jedes *Component*-Objekts und werden immer dann aufgerufen, wenn das AWT feststellt, daß die Anzeige auf dem Bildschirm erneuert werden muß. *paint* wird verwendet, wenn das AWT den Inhalt des Fensters komplett neu angezeigt haben will, *update* wird aufgerufen, wenn Teile des Fensters verdeckt waren und nur diese Teile auch wieder angezeigt werden sollen. Wird *update* nicht überschrieben, so ruft die Basisklasse in *update* die Methode *paint* auf.

Zur graphischen Ausgabe wird immer ein *Graphics*-Objekt benötigt. Deshalb liefert das AWT bei dem Aufruf von *paint* und *update* ein entsprechendes Objekt mit und die Ausgabe in das Fenster kann direkt über dieses Objekt geschehen, ohne das eine Aufruf von *getGraphics* notwendig ist.

Abschließend soll ein Beispiel für die Verwendung von Graphik gezeigt werden. Es soll eine Klasse realisiert werden, die einen Button darstellt, der statt eines Textes ein Bild anzeigt. Die nachfolgende Abbildung zeigt einen solchen Button innerhalb eines Applets.

Das Beispiel zeigt auch die Verwendung der Ereignisbehandlungsfunktionen *mouseDown* und *mouseUp*.

Der Konstruktor beschränkt sich auf das Festlegen der Größe des Buttons und der Speicherung des übergebenen Bildes. Weiterhin wird die Variable *buttonIsPressed* initialisiert. Diese soll angeben, ob eine Maustaste gedrückt ist. Der Konstruktor geht davon aus, daß dies zum Zeitpunkt der Erstellung nicht der Fall ist. Um die Variable entsprechend der Benutzeraktion zu verändern, werden die beiden Ereignisbehandlungsfunktionen *mouseDown* und *mouseUp* überschrieben, so daß die Variable entsprechend gesetzt werden kann. Diese Funktionen teilen dem AWT auch mit, daß der Button neu angezeigt werden muß, da dieser dem Anwender den „Drück"-Effekt vermitteln soll. Dies geschieht durch den Aufruf der Methode *repaint*.

Die eigentliche Anzeige wird in der Methode *paint* gemacht. Diese ermittelt zunächst die Größe des Fensters, die gebraucht wird, um ein 3D-Rechteck als Umrandung des Buttons anzuzeigen. Das 3D-Rechteck ist ein Recheck, das den Eindruck einer angehobenen oder abgesenkten Fläche vermittelt (wie bei Buttons üblich). Hierfür wird der obere und der linke Rand in einer anderen Farbe als der untere und der rechte Rand gezeichnet. Die zu verwendenden Farben werden mit einem Aufruf von *setColor* festgelegt. Diese Methode legt die aktuelle Zeichenfarbe für fast alle Zeichenfunktionen fest. Das 3D-Rechteck benutzt die aktuelle Zeichenfarbe für den oberen und den linken Rand des Rechtecks, wenn die eingeschlossene Fläche angehoben erscheinen soll, und für den unteren und den rechten Rand, wenn die Fläche abgesenkt erscheinen soll. Der jeweils andere Teil wird in einer dunkleren Farbe gezeichnet. Das AWT bietet in der Klasse *Color* die Möglichkeit mit den Methoden *brighter* und *darker* eine Farbe zu ermitteln, die heller bzw. dunkler als die Ausgangsfarbe ist (eine dunklere Farbe entspricht 70% der Ausgangsfarbe).

Ein „3D-Look" wird am besten erreicht, wenn als Zeichenfarbe Weiß verwendet wird.

Nachdem die aktuelle Zeichenfarbe gesetzt wurde, wird das 3D-Rechteck gezeichnet. Da alle Zeichenfunktionen des AWT immer nur ein Pixel breite Linien zeichnen, werden zwei Aufrufe von *draw3DRect* gemacht. Den beiden Aufrufen werden jeweils etwas unterschiedliche Rechtecke angegeben, so daß die Breite der Linien des entstehenden Rechtecks zwei Pixel beträgt.

Neben den Koordinaten des Ursprungs und den Ausdehnungen des Rechtecks wird zusätzlich angegeben, ob das Rechteck eine angehobene oder eine abgesenkte Fläche hervorrufen soll. In dem Beispiel soll der Button abgesenkt sein, wenn eine Maustaste im Bereich des Buttons gedrückt ist. Deshalb wird für den letzten Parameter von *draw3DRect* die Variable *isButtonPressed* übergeben.

für eigene GUI-Elemente zur Verfügung gestellt.

```
import java.awt.*;

public class ImgButton extends Canvas {

        private boolean buttonIsPressed;
        private Image buttonImage;

        public ImgButton(Image buttonImage, int width, int height)  {
                this.buttonImage=buttonImage;
                mouseButtonIsPressed=false;

                resize(width, height);
        }

        public void paint(Graphics grfx) {
                Dimension sizeComp=size();

                grfx.setColor(Color.white);

                grfx.draw3DRect(0, 0, sizeComp.width-1, sizeComp.height-1,
                                                        buttonIsPressed);
                grfx.draw3DRect(1, 1, sizeComp.width-3, sizeComp.height-3,
                                                        buttonIsPressed);

                if (buttonIsPressed)
                        grfx.drawImage(buttonImage, 4, 4, this);
                else
                        grfx.drawImage(buttonImage, 2, 2, this);
        }

        public boolean mouseDown(Event evt, int x, int y) {
                buttonIsPressed=true;
                repaint();

                return true;
        }

        public boolean mouseUp(Event evt, int x, int y) {
                buttonIsPressed=false;
                repaint();

                return true;
        }
}
```

Das java.io Package gliedert sich in zwei Bereiche:

- Nicht Stream-orientierte Klassen
- Stream-orientierte Klassen

Die Funktionen dieses Packages unterliegen dem Sicherheitskonzept von Java, so daß bestimmte Methoden nur in entsprechenden Fällen das gewünschte Ergebnis liefern.

5.2.1 Nicht Stream-orientierte Klassen

Java bietet mit der Klasse *File* ein Objekt, das mit Dateien oder Verzeichnissen umgehen kann. Dieser Umgang umfaßt das Auflisten von Dateien eines Verzeichnisses, das Löschen und Umbenennen und das Ermitteln von Dateiattributen. Das Lesen und Schreiben von Daten aus bzw. in eine Datei kann mittels der Klasse *RandomAccessFile* geschehen. Diese erlaubt über die Methode *seek* auch das Setzen der aktuellen Lese- bzw. Schreibposition.

5.2.2 Stream-orientierte Klassen

In vielen Fällen werden Dateien aber sequentiell gelesen und geschrieben. Dann kann man auf die Stream-orientierten Klassen von Java zurückgreifen. Neben einer Datei als Quelle bzw. Ziel bieten diese Klassen auch andere Quellen bzw. Ziele: Pipes, Strings, Byte-Arrays, den Bildschirm, usw.

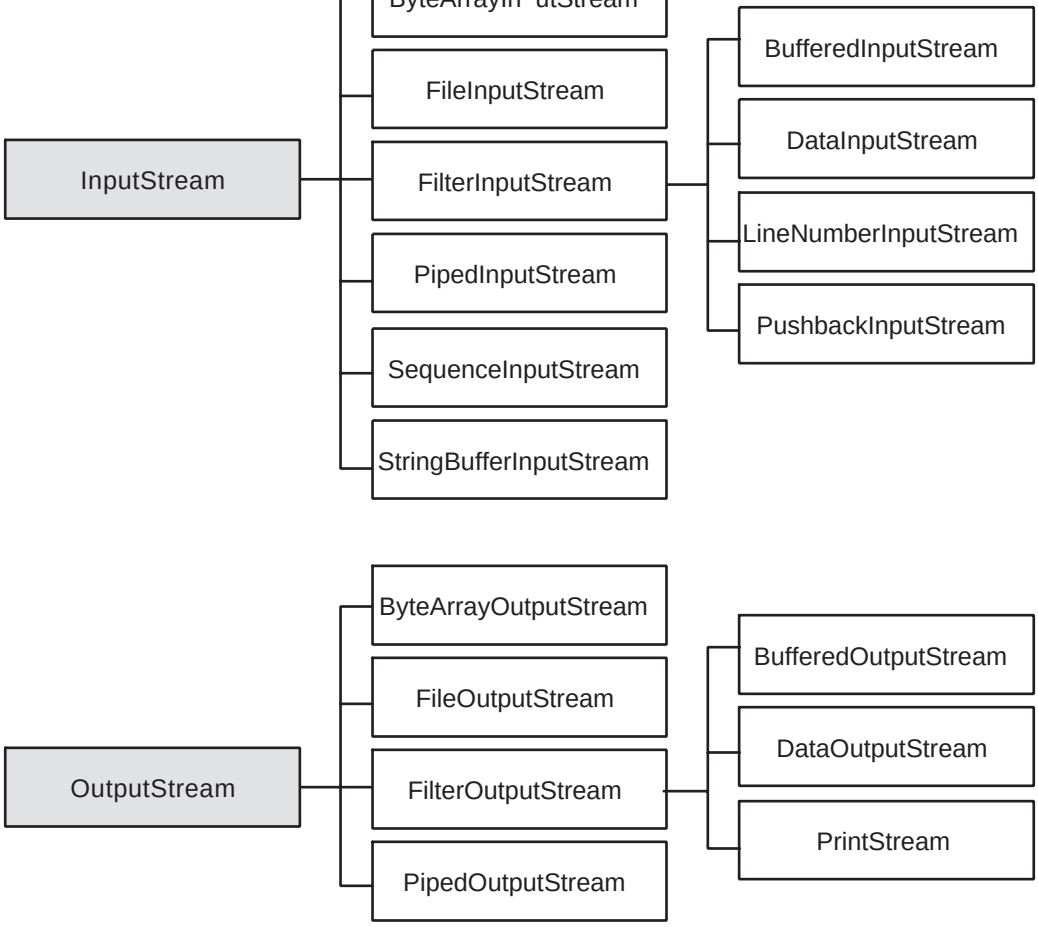
Grundsätzlich lassen sich Stream in zwei Varianten einteilen:

- Eingabestreams
- Ausgabestreams

Konsequenterweise bietet Java deshalb zwei grundlegende Klassen für die Arbeit mit Streams: *InputStream* und *OutputStream*. Diese Klassen sind abstrakt und definieren (teilweise abstrakte) Methoden zum Umgang mit Streams. Zu diesen Methoden gehört das Lesen und Schreiben von Bytes sowie das Schließen (das Öffnen geschieht in der Regel über den Konstruktor) und das „Leeren“ des Streams.

Unterklassen von *InputStream* und *OutputStream* bieten die Verwendung von Streams für die unterschiedlichsten Zwecke an. In der Regel gibt es zu jedem Eingabe-Stream auch einen entsprechenden Ausgabestream.

Die nachfolgende Abbildung zeigt die Klassenhierachie der Streams.



Die Streams können entsprechend ihrer Quelle bzw. ihres Ziels gegliedert werden:

- Files

  Mit den Klassen *FileInputStream* und *FileOutputStream* wird ein sequentielles File-I/O angeboten. Mit Hilfe der Methoden *write* bzw. *read* kann man einzelne oder mehrere Bytes oder einen Integer in eine Datei schreiben bzw. aus einer Datei lesen.

- Byte-Arrays

  Die Klassen *ByteArrayInputStream* und *ByteArrayOutputStream* erlauben das Lesen und Schreiben innerhalb eines Arrays von Byte-Werten. Auch hier stehen *read* und *write* zur Verfügung. Jedoch nur für den Zugriff auf einzelne oder mehrere Bytes.

- Pipes

  Pipes dienen zur Kommunikation zwischen Threads. Hierfür benutzt einer der Threads einen *PipedOutputStream*, der andere einen *PipedInputStream*. Beide müssen über die Methode *connect* dieser Klassen oder über einen der Konstruktoren miteinander verbunden werden. Dann können einzelne oder mehrere Bytes zwischen den Threads ausgetauscht werden (*read* bzw. *write*).

- Strings

  Die Klasse *StringBufferInputStream* dient zum Lesen von Bytes aus einem String. Der zu verwendende String wird dem Stream-Objekt im Konstruktor übergeben. Mittels *read*

das Simulieren eines sequentiellen Lesevorgangs, der normalerweise von einer Datei oder über eine Pipe geschieht.

- Streams

  Einer der großen Vorteile des Streamkonzepts ist es auf einen Stream einen anderen aufsetzen zu können. Solche Aufsätze werden als Filter bezeichnet. Ein Filter kann ankommende Daten in ein anderes Format umwandeln, einen Lesepuffer realisieren, usw. Die Klassen *FilterInputStream* und *FilterOutputStream* stellen Basisklassen solcher Filter dar. Im java.io Package sind eine Reihe Filter bereits definiert, die für die meisten Aufgaben verwendet werden können. Im Einzelnen sind dies:

  - BufferedInputStream, BufferedOutputStream

    Verwendet einen Pufferspeicher, um ein effizentes Lesen bzw. Schreiben des Streams zu gewährleisten.

  - DataInputStream, DataOutputStream

    Diese Klassen erlauben das Lesen bzw. Schreiben von einfachen Datentypen wie Byte, Char, Double, Float, Int, ... Wichtig ist, daß die Daten in einer maschinenunabhängigen Art gelesen und geschrieben werden. Datentypen wie unsigned Byte, die Java nicht kennt, können hiermit verarbeitet werden. Diese Klassen eignen sich vor allen Dingen für die Verwendung beim Lesen und Schreiben von Dateien. Dazu wird ein *DataInputStream* bzw. ein *DataOutputStream* auf einen *FileInputStream* bzw. einen *FileOutputStream* aufgesetzt.

  - LineNumberInputStream

    Ein Filter, der die Fähigkeit hat, die aktuelle Zeile zu ermitteln (*getLineNumber*).

  - PrintStream

    Ein *FilterOutputStream*, der die Java-Datentypen schreiben kann. Diese Klasse implementiert die überladenen Methoden *print* und *println*, um die folgenden Datentypen schreiben zu können: *char*, *int*, *long*, *float*, *double*, *boolean* und *String*. Die Klasse *System* verwendet einen solchen Stream, um textuelle Ausgaben auf den Bildschirm zu machen (*System.out*).

  - PushbackInputStream

    Diese Klasse erlaubt es, mittels der Methode *read* Daten zu lesen und mittels *unread* wieder zum Lesen freizugeben. Ein durch *unread* der Klasse übergebenes Zeichen wird durch den nächsten Aufruf von *read* wieder gelesen.

- Sequenzen von Streams

  Die Klasse *SequenceInputStream* erlaubt das Hintereinanderschalten mehrerer Eingabestreams. Dem Konstruktor werden zwei oder mehr *InputStreams* übergeben, die dann der Reihe nach gelesen werden können. Dabei wird zunächst der erste Stream von vorne bis hinten mittels *read* ausgelesen, dann der zweite usw.

Natürlich kann der Programmierer auch eigene Streams implementieren. In der Regel werden dies Filter sein, die den Austausch von speziellen Objekten ermöglichen. So z.B. der Aufbau einer 3D-Szene. Über einen File-Stream wäre es auch denkbar, dieses Objekt in einer Datei abzulegen, eine Pipe würde es zwischen Threads austauschbar machen. Und mittels einer Netzwerk-Verbindung (s.u.) können sogar zwei oder mehr Clients auf die Daten eines Servers zugreifen, der solche

von Java: Verteilte Anwendungen auf einfachste Art realisieren zu können.

Im nächsten Kapitel ist ein Beispiel zu einer Client-Server-Kommunikation angegeben, die mittels zweier Streams einen String austauscht.

5.3 Das Package java.net

Eine der großen Stärken von Java ist Fähigkeit, ohne großen Aufwand Netzwerk-Kommunikation zu ermöglichen. Die Basis bildet dabei eine Socket-Kommunikation mittels Datagrammen, wie sie von TCP/IP her bekannt ist.

Die Klassen *DatagramPacket* und *DatagramSocket* dienen dazu, solche Datagramme direkt zu verschicken und zu empfangen.

Auf einem etwas höheren Level wird eine Kommunikation zweier Sockets über Streams erreicht. Dies geschieht über die Klassen *Socket* bzw. *ServerSocket*. Mit den Methodes *getInputStream* und *getOutputStream* der Klasse *Socket* werden die Streams ermittelt, die für die Kommunikation benutzt werden. Die Methode *accept* der Klasse *ServerSocket* wartet auf eine Client-Anforderung und liefert einen Socket zurück, der den Client-Socket darstellt.

Noch etwas höher findet man die Klassen *URL* und *URLConnection*. Mit diesen kann man eine Verbindung zu einem Objekt im Internet herstellen. *URL* gibt dafür die Internet-Location an, an der sich das Objekt befindet. *URLConnection* ist eine abstrakte Klasse, die eine Verbindung zu diesem Objekt darstellt. Über diese Verbindung lassen sich Streams erstellen, mit deren Hilfe die Daten des Objektes gelesen bzw. geschrieben werden können. Auch diverse andere Attribute des Objektes lassen sich ermitteln.

URLConnections unterliegen jedoch dem strengen Sicherheitskonzept von Java. Deshalb kann die Funktionalität dieses Packages nur in bestimmten Fällen vollständig genutzt werden.

Nachfolgend sei noch ein Beispiel gegeben, das eine Client-Server-Kommunikation mittels Streams realisiert. Das Beispiel ist in zwei Teile gegliedert. Teil I stellt den Server dar , Teil II den Client.

Der Server ist als Applikation realisiert. Er wird beim Start einen Socket erstellen und an diesem ständig auf eine Client-Anforderung warten (*Server.run*). Meldet sich ein Client beim Server, so wird der Socket des Clients ermittelt und eine Verbindung (realisiert in der Klasse *Connection*) erstellt. Diese wird selbst zwei Streams erstellen, die zur Kommunikation mit dem Client dienen (*in* und *out*). Als Streams wird für das Empfangen von Daten ein *DataInputStream* und für das Senden ein *PrintStream* benutzt. In der Methode *Connection.run* wird ein String vom Client empfangen und wieder zurückgeschickt.

Der Client ist ein Applet und kann deshalb in einem WWW-Browser ablaufen. Beim Start des Applets wird sowohl ein Socket als auch zwei Streams erstellt. Wie beim Server dienen die Streams zur Kommunikation. Auch hier kommen wieder ein *PrintStream* und ein *DataInputStream* zum Einsatz. Das Applet soll so aufgebaut sein, daß es vom Benutzer einen Text entgegen nimmt und diesen an den Server schickt. Vom Server gesendete Strings werden auf dem Bilschirm ausgegeben. Die Eingabe von Text wird in der Methode *Client.action* erkannt. *action* ist eine Ereignisbehandlungsfunktion und wird durch *handleEvent* aufgerufen, wenn z.B. ein Text in einem *TextField* eingegeben wurde (*TextField* ist eines der vom AWT definierten Kontrollelemente). In dem Beispiel wird der eingegebene Text direkt an den Ausgabestream übergeben und dadurch zum Server geschickt.

Bildschirm ausgegeben (*StreamListener.run*). Ein Objekt dieser Klasse erzeugt der Client gleich beim Start des Applets.

Interessant ist die Verwendung der Streams. Beide Seiten der Kommunikation (Server und Client) ermitteln jeweils zwei Streams (mit *getInputStream* bzw. *getOutputStream*, die zum Senden bzw. Empfangen von Daten dienen. Der Client ermittelt die Streams durch seinen eigenen Socket, der Server durch den von der Methode *accept* ermittelten Socket des Clients. Die Methode *accept* wird aufgerufen, um auf eine Anforderung des Clients zu warten.

In diesem einfachen Beispiel werden Strings zur Kommunikation verwendet. Deshalb müssen die Streams, die die Sockets liefern als Quellen bzw. Ziele von Filtern dienen, die Strings verschicken bzw. empfangen können. Wollte man nur Bytes verschicken, so könnte man die von den Sockets gelieferten Streams direkt verwenden. Komplexere Klassen sollten eigene Filter bereitstellen, um das Senden und Empfangen von Objekten dieser Klassen zu ermöglichen.

Teil I: Der Server.

```
import java.io.*;
import java.net.*;

public class Server extends Thread {
    // Ein Socket wird an einem Port erstellt. In diesem Fall wird
    // der Port 6789 angenommen. Ein Client, der mit diesem Server
    // kommunizieren will, mu sich an diesen Port wenden, um mit
    // seinen eigenen Socket mit diesem zu kommunizieren.
        public final static int DEFAULT_PORT=6789;
        protected ServerSocket listenSocket;

    // Falls ein Fehler auftritt, wird eine Meldung ausgegeben und
    // das Programm beendet.
        public static void fail(Exception e, String msg) {
            System.err.println(msg+": "+e);
            System.exit(1);
        }

    // Erstellen des Server-Sockets und starten des Wartevorgangs.
        public Server() {
            try { listenSocket=new ServerSocket(DEFAULT_PORT); }
            catch (IOException e) fail(e, "Exception creating server socket");
            System.out.println("Server: listening on port "+DEFAULT_PORT);
            this.start();
        }

    // Der Wartevorgang: Hier wird gewartet, bis ein Client sich beim
    // Server meldet. Geschieht dies, wird eine Verbindung aufgebaut.
    // Die Methode listenSocket liefert den Socket des Clients.
        public void run() {
            try {
                while (true) {
```

```
                    Connection c=new Connection(clientSocket);
                }
            }
            catch (IOException e)
                  fail(e, "Exception while listening for connection.");
        }

    // Starten der Applikation
        public static void main(String[] args) {
            new Server();
        }

}

// Die Kommunikation mit dem Client
class Connection extends Thread {
        protected Socket client;
        protected DataInputStream in;
        protected PrintStream out;

    // Erstellen der Streams und starten des Serverdienstes.
    // Hierfr wird der Socket des Clients nach den Streams
    // gefragt.
        public Connection(Socket clientSocket) {
            client=clientSocket;
            try {
                in=new DataInputStream(client.getInputStream());
                out=new PrintStream(client.getOutputStream());
            }
            catch (IOException e) {
                try client.close(); catch(IOException e2);
                System.err.println("Exception while getting "
                                   +"socket streams: "+e);
                return;
            }
            this.start();
        }

    // Der Serverdienst: Eine ankommende Textzeile wird wieder an
    // den Client zurckgeschickt.
        public void run() {
            String line;

            try {
                for(;;) {
                    line=in.readLine();
                    out.println(line);
                }
```

```
            catch (IOException e);
            finally try client.close(); catch (IOException e2);
        }
}
```

Teil II: Der Client.

```
import java.awt.*;
import java.applet.*;
import java.io.*;
import java.net.*;

public class Client extends Applet {
    public static final int PORT=6789;
    Socket s;
    DataInputStream in;
    PrintStream out;
    TextField inputField;
    TextArea outputArea;
    StreamListener listener;

// Erstellen des Sockets, der Streams, der Bildschirmmaske und
// Anstoen des Lauschvorgangs am Server.
    public void init() {
        try {
            s=new Socket(this.getCodeBase().getHost(), PORT);
            in=new DataInputStream(s.getInputStream());
            out=new PrintStream(s.getOutputStream());

            inputField=new TextField();
            outputArea=new TextArea();
            outputArea.setEditable(false);
            setLayout(new BorderLayout());
            add("North", inputField);
            add("Center", outputArea);

            listener=new StreamListener(in, outputArea);

            showStatus("Connected to "
                               +s.getInetAddress().getHostName()
                               +":"+s.getPort());
        }
        catch (IOException e) showStatus(e.toString());
    }

// Wenn der Benutzer einen Text eingegeben hat, dann wird dieser Text
```

```
    public boolean action(Event e, Object what) {
        if (e.target==inputField) {
            out.println((String)e.arg);
            inputField.setText("");
            return true;
        }
        return false;
    }
}


class StreamListener extends Thread {
    DataInputStream in;
    TextArea output;

    public StreamListener(DataInputStream in, TextArea output) {
        this.in=in;
        this.output=output;
        this.start();
    }

// Lauschvorgang: Alles was der Server zum Client schickt, wird
// auf dem Bildschirm ausgegeben.
    public void run() {
        String line;

        try {
            for(;;) {
                line=in.readLine();
                if (line==null) break;
                output.setText(line);
            }
        }
        catch (IOException e) output.setText(e.toString());
        finally output.setText("Connection closed by server.");
    }
}
```

5.4 Ausblick

Die Bibliotheken von Java sind keineswegs vollständig und es sind zahlreiche Erweiterungen denkbar. Sehr stark wird man im Moment das Fehlen von Druckerausgaben vermissen. Dies wird den Einsatz von Java in größeren Anwendungen noch verhindern. Andere Erweiterungen könnten auch 3D-Ausgaberoutinen umfassen und vor allen Dingen eine Schnittstelle zu Datenbanksystemen.

Das Sun sich dieser „Mankos“ bewußt ist, zeigen viele Pressemitteilungen. Anfang Mai gab Sun bekannt, daß ein API für 2D-Graphik entwickelt wird. Gleichzeitig wurde Interesse an einem

Kurze Zeit später wurden APIs für Multimedia-Fähigkeiten, 3D-Graphik, verteilte Datenbanken, Unterstützung von Sicherheitsmechanismen (Kodierung, Digitale Unterschrift, Authentifikation), usw. angekündigt.

Nicht zuletzt hat Sun auch Pläne das bestehende JDK weiter auszubauen.

Abschließend muß man noch sagen, daß es kaum eine Programmiersprache gibt, die vom „Werk“ aus mit solchen starken Bibliotheken ausgeliefert wird ohne gleichzeitig nur für spezielle Anwendungen konzipiert zu sein.

Java Bytecode

Hans-Ulrich Schlieben

8. Juli 1996

6 Java Bytecode

6.1 Einleitung

Ein Vergleich der Java-Virtual-Machine mit real existierenden CPU's (Central Processing Unit) kann nicht auf der Basis von Mips oder Specs geführt werden. Das eigentliche Ziel von Java ist es auf möglichst vielen realen CPU's zu arbeiten. Der folgende Vergleich der Architektur, der Datentypen und der Befehle soll aufzeigen, was die Unterschiede zu einer realen CPU ausmachen und folglich vom Interpreterentwickler zu implementieren ist.

Die Java-Virtual-Machine ist eine reine Stackmaschine. Dies bedeutet, daß Lade- und Speicheroperationen immer den Stack als Quelle oder Ziel benutzen. Die eigentlichen Operationen wie z.B. die Addition bezieht ihre Operanden vom Stack und legt das Ergebnis wieder auf dem Stack ab. Um die Daten zu Speichern stehen jeder Klasse oder Methode unter anderem lokale Variablen zur Verfügung. Jede Klasse besitzt eine Ausführungsumgebung, die es ihr durch dynamisches Linken ermöglicht andere Klassen aufzurufen oder die Kontrolle an die aufrufende Klasse zurückgeben.

Interessant am Java-Bytecode ist, daß es eine Art „Styleguide“ gibt. Wie es z.B. bei der Fensteroberfläche des Apple Macintosh verboten ist eine Laufleiste am oberen Rand des Fensters anzuordnen, so ist es im Java-Code verboten zuerst eine Zahl auf den Stack zu legen und Sie dann als Adresse zu mißbrauchen. Die Java-Virtual-Machine würde diese zwei Anweisungen im Prinzip ausführen. Es gibt aber Programme, mit denen Java-Code auf solche Fehler hin untersucht werden kann, da Java-Code typisiert ist. Es kann anhand des Codes bestimmt werden, welche Speicher- oder Stackwörter welche Datentypen enthalten. Erst diese Zusicherungen machen den Java-Code einigermaßen sicher, sodaß der Anwender sich trauen darf ein wildfremdes Programm auf seiner Maschine zu starten.

Die Reglementierung bei Java geht weit über den eigentlichen Bytecode hinaus. So ist z.B. das Format der Klassendateien exakt festgelegt. Um das Binden der Klassen flexibel zu halten wird erst zur Laufzeit gebunden. Benutzt im Java-Code eine Klasse eine andere, so steht in der Klassendatei im sogenannten Constantpool im Klartext sowohl der Datei- als auch der Methodenname. Es ist die Aufgabe des Interpreters die Datei zu laden, die Ablaufumgebung der Methode zu generieren und diese schließlich zu starten. Die langsame Verbindung der Methoden über den Constantpool muß nur beim ersten Zugriff erfolgen. Ist die Verbindung erst einmal hergestellt, so wird der Befehl im Code durch eine schnellere Variante ersetzt, die die Methode direkt referenziert.

Die Geschwindigkeit darf nicht ganz außer acht gelassen werden, da Java größtenteils für interaktiv bedienbare Programme gedacht ist. Momentan ist Java etwa 10 mal langsamer als übersetzter C-Code. Es wird aber bereits an einer richtigen Hardware-Java-CPU entwickelt.

6.2.1 Datentypen

Die virtuellen Maschinendatentypen enthalten die Basisdatentypen von Java:

`byte`	1-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`short`	2-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`int`	4-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`long`	8-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`float`	4-Byte IEEE 754 Fließkommazahl einfacher Genauigkeit
`double`	8-Byte IEEE 754 Fließkommazahl doppelter Genauigkeit
`char`	2-Byte vorzeichenloses Unicodezeichen
`object`	4-Byte Referenz auf ein Java Objekt
`returnAddress`	4-Bytes benutzt bei den Befehlen jsr, ret, jsr_w, ret_w

Tabelle 1: Datentypen der Javamaschine

Die interne Darstellung eines Objektes, auf die `object` zeigt, ist implementationsabhängig. In der Implementation von Sun zeigt `object` auf ein Paar von Zeigern, die auf die Daten und die Methodentabelle verweisen. Der Adreßraum ist momentan auf 32 Bit begrenzt. Es ist aber möglich, eine virtuelle Javamaschine zu entwickeln, die den Code automatisch auf 64 Bit erweitert.

Der Datentyp `boolean` wird, wenn er einzeln auftritt, auf `int` abgebildet. Im Gegensatz dazu wird ein `array of boolean` als `array of byte` bearbeitet.

6.2.2 Register

Zu jeder Zeit führt die virtuelle Javamaschine den Code einer Javamethode aus. Der Programmzähler `pc` zeigt immer auf den nächsten auszuführenden Befehl. Jede Methode hält Speicherplatz für lokale Variablen, den Operandenstack und die Ausführungsumgebungsstruktur bereit, die über die Register `vars`, `optop` und `frame` referenziert werden. Dieser Speicher kann auf einmal alloziert werden, da der Bedarf für die lokalen Variablen und den Stack schon zur Übersetzungszeit und die Größe der Umgebungsstruktur dem Interpreter bekannt ist. Alle Register sind 32 Bit groß.

6.2.3 Operand Stack

Alle Maschinenbefehle arbeiten auf dem Operandenstack. Sie holen ihre Operanden vom Stack und legen die Ergebnisse wieder auf dem Stack ab. Die Stackorganisation wurde gewählt, um auch CPU's mit kleiner Registeranzahl oder irregulärer Registerstruktur zu unterstützen. Der Stack ist 32 Bit breit. Auch kleinere Datentypen wie z.B. `char` werden 32 Bit breit auf dem Stack abgelegt. Die Typen `long` und `double` benötigen zwei Plätze auf dem Stack. Die Reihenfolge (high, low) ist implementationsabhängig. In Java müssen auch auf dem Stack die jeweiligen Typen genau beachtet werden. So ist es z.B. verboten zwei `int` auf dem Stack abzulegen und sie dann als `long` zu verarbeiten. In Java ist zu jeder Zeit der Typ jedes Elementes des Stacks bekannt. Weiterhin wird verlangt, daß der Typ jedes Elements an einer Programmstelle immer gleich ist. Dies verbietet es z.B. einen String in einer Schleife auf den Stack zu legen.

```
char text[]= "Es ist verboten einen String auf den Stack zu legen";
int i;

for(i= 0; text[i] != 0; i++)
  PushCharOnStack(text[i]);
```

Innerhalb der Schleife würde sich bei jedem Schleifendurchlauf der Typstatus ändern. Bei jedem Aufruf von `PushCharOnStack` wäre der Stack unterschiedlich belegt. Diese Einschränkungen ermöglichen es aber schon zur Übersetzungszeit die Stackgröße festzustellen. Außerdem erspart man sich die dauernde Überprüfung auf Über- oder Unterlauf des Stacks zur Laufzeit.

6.2.4 Garbage Collected Heap

Der Java Heap ist der Laufzeitdatenbereich in welchem Klassen (Objekte) Speicherplatz belegen. Die Sprache Java gibt dem Programmierer nicht die Möglichkeit den Speicherplatz explizit wieder freizugeben. Obwohl Java als Garbage Collected Sprache gedacht ist, ist der eigentliche Algorithmus nicht festgeschrieben.

6.2.5 Beschränkungen

Der `constant pool` einer Klasse hat ein Maximum von 65535 Einträgen. Die Anzahl von Argumentwörtern bei einem Methodenaufruf ist auf 255 begrenzt.

Momentan sind noch die Zeilennummerntabelle, die `code exception table` und die lokalen Variablen auf 65535 Einträge beschränkt. Dies soll ab der Version 1.0beta2 nicht mehr der Fall sein.

6.3 Das Format der Klassendateien

Jede Klassendatei (`.class`) enthält die compilierte Version einer Java Klasse oder eines Java Interfaces.

Die Datei besteht aus 8-Bit Bytes. Alle 16 und 32 Bitwerte werden mit den höherwertigen Bytes zuerst abgespeichert. Aufeinanderfolgende Felder sind zusammenhängend und nicht auf gerade Adressen ausgerichtet. Die im folgenden angegebenen Strukturen enthalten Arrays variabler Größe mit variabel großen Elementen und werden Tabellen genannt.

Die Typen `u1`, `u2` und `u4` bezeichnen eins, zwei und vier Byte große vorzeichenlose Wörter.

6.3.1 Format

Die folgende Pseudostruktur gibt die oberste Ebene der `.class`-Datei an:

```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
```

```
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

`magic`

Dieses Feld muß den Wert `0xCAFEBABE` enthalten, um die Datei als Javadatei zu markieren.

`minor_version, major_version`

Versionsnummer des Javacompilers mit dem die Datei erzeugt wurde. Es wird nur Code eines Compilers mit gleicher Hauptversionsnummer ausgeführt.

`constant_pool`

Die Tabelle `constant_pool` enthält verschiedene Stringkonstanten, Klassennamen und andere Konstanten, die aus der Datei referenziert werden. Das voranstehende Feld `count` gibt die Anzahl der Einträge an. Textfelder werden wie in Java üblich in Utf8 codiert.

`access_flags`

Im Feld `access_flags` ist bitweise codiert, ob die Klasse Public, Final und ein Interface ist.

`this_class`

Dieses Feld ist ein Index in den Konstantenpool.

`super_class`

Ist `super_class` gleich Null, dann hat die definierte Klasse keine Superklasse und ist vom Typ `java.lang.Object`. Ansonsten enthält `constant_pool[super_class]` eine Referenz auf die Superklasse.

`interfaces`

Jeder Wert dieser Tabelle gibt einen Index in den Konstantenpool an. Jeder bezeichnete Eintrag muß ein Interface sein.

Hier stehen die Informationen zu den von der Klasse definierten Variablen. Definiert werden jeweils die Zugriffsrechte, der Name und die Signatur.

`methods`

Für die definierten Methoden werden jeweils Zugriffsrechte, Name, Signatur und Attribute definiert. In den Attributen steht z.B. auch der eigentliche Code der Methode.

`attributes`

Bisher gibt es hier nur das Attribut `SourceFile`, welches den Namen der Javadatei angibt.

6.4 Der virtuelle Maschinenbefehlssatz von Java

Im folgenden wird nicht weiter erwähnt, daß Operanden immer vom Stack entfernt werden, soweit nicht anders angegeben.

6.4.1 Lade Konstanten auf den Stack

Die Typen `byte` und `short` können mit den Befehlen `bipush` und `sipush` direkt aus dem Bytecode generiert werden. Diese „Abkürzung“ gibt es für die komplexeren Typen mit den `const`-Befehlen nur für wenige häufig benötigte Werte. Alle anderen Konstanten müssen mit `ldc` aus dem Konstantenpool geholt werden.

bipush, byte1 Der vorzeichenbehaftete 8-Bit Wert byte1 wird als Integer auf den Stack geladen.

sipush, byte1, byte2 Der aus byte1 und byte2 zusammengesetzte vorzeichenbehaftete 16-Bit Wert wird als Integer auf den Stack geladen.

ldc1, indexbyte1 Der vorzeichenlose 8-Bit Index zeigt in den Konstantenpool. Das dortige Element wird auf den Stack geladen.

ldc2, indexbyte1, indexbyte2 Der vorzeichenlose 16-Bit Index zeigt in den Konstantenpool. Das dortige Element wird auf den Stack geladen.

ldc2w, indexbyte1, indexbyte2 Der vorzeichenlose 16-Bit Index zeigt in den Konstantenpool. Das dortige Element wird auf den Stack geladen. Dieses Element ist eine Doppelwortkonstante, also vom Typ double oder long.

aconst_null Es wird eine Null-Referenz auf den Stack geladen.

iconst_m1 Legt den Integerwert -1 auf den Stack.

iconst_<n> Legt einen der Integerwerte $0, 1, 2, 3, 4, 5$ auf den Stack.

lconst_<l> Legt einen der Long-Integerwerte $0, 1$ auf den Stack.

fconst_<f> Legt einen der Floatwerte $0, 1, 2$ auf den Stack.

dconst_<d> Legt einen der Doublewerte $0, 1$ auf den Stack.

Mit den `load`-Befehlen werden lokale Variablen auf den Stack geladen. Für alle Befehle gibt es zusätzlich Bytecodes, die die ersten vier Variablen direkt ohne den Parameter `vindex` adressieren.

Die Bytecodes `iload` und `fload` z.B. tun technisch dasselbe. Es gibt nur deshalb unterschiedliche Bytecodes, um jederzeit den Typ bestimmen zu können.

iload, vindex Der Wert der lokalen Integervariablen an der Stelle vindex des momentanen Javaframes wird auf den Stack geladen.

iload_<n> Der Wert der lokalen Integervariablen $0, 1, 2$ oder 3 des momentanen Javaframes wird auf den Stack geladen.

lload, vindex Der Wert der lokalen Longvariablen vindex des momentanen Javaframes wird auf den Stack geladen. Es werden zwei Wörter auf den Stack geladen.

lload_<n> Der Wert der lokalen Longvariablen $0, 1, 2$ oder 3 des momentanen Javaframes wird auf den Stack geladen.

fload, vindex Der Wert der lokalen Floatvariablen vindex des momentanen Javaframes wird auf den Stack geladen.

fload_<n> Der Wert der lokalen Floatvariablen $0, 1, 2$ oder 3 des momentanen Javaframes wird auf den Stack geladen.

dload, vindex Der Wert der lokalen Doublevariablen vindex des momentanen Javaframes wird auf den Stack geladen. Es werden zwei Wörter auf den Stack geladen.

dload_<n> Der Wert der lokalen Doublevariablen $0, 1, 2$ oder 3 des momentanen Javaframes wird auf den Stack geladen.

aload, vindex Der Wert der lokalen Objektreferenzvariablen vindex des momentanen Javaframes wird auf den Stack geladen.

aload_<n> Der Wert der lokalen Objektreferenzvariablen $0, 1, 2$ oder 3 des momentanen Javaframes wird auf den Stack geladen.

6.4.3 Lade Variablen vom Stack

Die Variable wird vom Stack in eine wie auch bei den `load`-Befehlen adressierte und typisierte lokale Variable gespeichert.

Der Stack wird um die Anzahl der geladenen Wörter verringert.

istore, vindex Die lokale Integervariable an der Stelle vindex des momentanen Javaframes wird mit dem Wert vom Stack beschrieben.

istore_<n> Die lokale Integervariable $0, 1, 2$ oder 3 des momentanen Javaframes wird mit dem Wert vom Stack beschrieben.

lstore, vindex Die lokale Longvariable an der Stelle vindex des momentanen Javaframes wird mit zwei Wörtern vom Stack beschrieben.

Wörtern vom Stack beschrieben.

fstore, vindex Die lokale Floatvariable an der Stelle vindex des momentanen Javaframes wird mit dem Wert vom Stack beschrieben.

fstore_<n> Die lokale Floatvariable $0, 1, 2$ oder 3 des momentanen Javaframes wird mit dem Wert vom Stack beschrieben.

dstore, vindex Die lokale Doublevariable an der Stelle vindex des momentanen Javaframes wird mit zwei Wörtern vom Stack beschrieben.

dstore_<n> Die lokale Doublevariable $0, 1, 2$ oder 3 des momentanen Javaframes wird mit zwei Wörtern vom Stack beschrieben.

astore, vindex Die lokale Objektreferenzvariable an der Stelle vindex des momentanen Javaframes wird mit dem Wert vom Stack beschrieben.

astore_<n> Die lokale Objektreferenzvariable $0, 1, 2$ oder 3 des momentanen Javaframes wird mit dem Wert vom Stack beschrieben.

6.4.4 Behandlung von Feldern

Mit den verschiedenen `newarray`-Befehlen werden Felder alloziert. Die Länge eines Feldes kann man mit `arrylength` erhalten. Mit den entsprechenden `aload`-Codes kann der Inhalt eines Feldelementes auf den Stack geladen werden. Mit `astore` kann ein Feldelement beschrieben werden.

newarray, atype Es wird ein Array vom Typ atype erzeugt. Die Größe des Feldes wird vom Stack geholt. Die Referenz des Feldes wird auf dem Stack abgelegt.

anewarray, indexbyte1, indexbyte2 Es wird ein Objektarray erzeugt. Indexbyte1 und 2 bilden einen Index auf eine Klasse im Konstantenpool, die den Typ des Feldes angibt. Die Größe des Feldes wird vom Stack geholt. Die Referenz des Feldes wird auf dem Stack abgelegt.

multianewarray, indexbyte1, indexbyte2, dimensions Es wird ein mehrdimensionales Objektarray erzeugt. Indexbyte1 und 2 bilden einen Index auf eine Klasse im Konstantenpool, die den Typ angibt. Die Größen der Dimensionen werden vom Stack geholt. Die Referenz des Feldes wird auf dem Stack abgelegt.

arraylength Erwartet auf dem Stack eine Referenz auf einen Array. Liefert auf dem Stack die Größe des Feldes zurück.

iaload; laload; faload; daload; aaload; baload; caload; saload Erwartet auf dem Stack eine Feldreferenz auf ein Feld vom jeweiligen Typ (integer, long, float, double, object, byte, char, short) und den Index. Liefert auf dem Stack das entsprechende Feldelement zurück.

iastore; lastore; fastore; dastore; aastore; bastore; castore; sastore Erwartet auf dem Stack eine Feldreferenz auf ein Feld vom jeweiligen Typ (integer, long, float, double, object, byte, char, short), den Index und den Wert. Der Wert vom Stack wird in das entsprechende Feld an der Stelle Index geschrieben.

Mit diesen untypisierten Befehlen kann der Stack manipuliert werden. Der Javaübersetzer ist dafür verantwortlich, daß z.B. nicht der Befehl `dup` ausgeführt werden kann, wenn sich eine Longvariable auf dem Stack befindet. Mit `pop` wird der Stack verkleinert. Die `dup`-Befehle duplizieren Wörter auf dem Stapel. Die beiden obersten Wörter auf dem Stack werden mit `swap` vertauscht.

pop Entferne ein Wort vom Stack.
Stack: ..., a → ...

pop2 Entferne zwei Wörter vom Stack.
Stack: ..., b, a → ...

dup Dupliziere das oberste Wort auf dem Stack.
Stack: ..., a → ..., a, a

dup2 Dupliziere die beiden obersten Wörter auf dem Stack.
Stack: ..., b, a → ..., b, a, b, a

dup_x1 Dupliziere das oberste Wort auf dem Stack und verschiebe es zwei Stellen nach unten.
Stack: ..., b, a → ..., a, b, a

dup2_x1 Dupliziere die beiden obersten Wörter auf dem Stack und verschiebe sie zwei Stellen nach unten.
Stack: ..., c, b, a → ..., b, a, c, b, a

dup_x2 Dupliziere das oberste Wort auf dem Stack und verschiebe es drei Stellen nach unten.
Stack: ..., c, b, a → ..., a, c, b, a

dup2_x2 Dupliziere die beiden obersten Wörter auf dem Stack und verschiebe sie drei Stellen nach unten.
Stack: ..., d, c, b, a → ..., b, a, d, c, b, a

swap Vertausche die beiden obersten Wörter auf dem Stack.
Stack: ..., b, a → ..., a, b

6.4.6 Arithmetikbefehle

Die Arithmetikbefehle erwarten ihre Operanden vom Stack und hinterlassen das Ergebnis an deren Stelle auf dem Stack. Die Typen `integer`, `long`, `float` und `double` sind zulässig. Als Operationen stehen die Addition, Subtraktion, Multiplikation, Division, Restbestimmung und die Negation zur Verfügung.

iadd; ladd; fadd; dadd Erwarten jeweils zwei Operanden vom entsprechenden Typ auf dem Stack, addieren diese und legen das Ergebnis auf dem Stack ab.

isub; lsub; fsub; dsub Erwarten jeweils zwei Operanden vom entsprechenden Typ auf dem Stack, subtrahieren den zweiten vom ersten Operanden und legen das Ergebnis auf dem Stack ab.

imul; lmul; fmul; dmul Erwarten jeweils zwei Operanden vom entsprechenden Typ auf dem Stack, multiplizieren diese und legen das Ergebnis auf dem Stack ab.

Stack, dividieren den ersten durch den zweiten Operanden und legen das Ergebnis auf dem Stack ab.

irem; lrem; frem; drem Erwarten jeweils zwei Operanden vom entsprechenden Typ auf dem Stack, dividieren den ersten durch den zweiten Operanden und legen den verbleibenden Rest auf dem Stack ab.

ineg; lneg; fneg; dneg Erwarten jeweils einen Operanden vom entsprechenden Typ auf dem Stack, negieren diesen und legen das Ergebnis auf dem Stack ab.

6.4.7 Logische Befehle

Es gibt die logischen Operationen Und, Oder, Exklusivoder und bitweises Schieben für die Typen `integer` und `long`.

ishl Erwartet auf dem Stack einen Wert und einen Faktor vom Typ Integer. Der Wert wird um die unteren 5 Bits von Faktor nach links geschoben und auf dem Stack abgelegt.

ishr Erwartet auf dem Stack einen Wert und einen Faktor vom Typ Integer. Der Wert wird um die unteren 5 Bits von Faktor nach rechts geschoben und auf dem Stack abgelegt. Beim Verschieben wird das Vorzeichen beachtet.

iushr Erwartet auf dem Stack einen Wert und einen Faktor vom Typ Integer. Der Wert wird um die unteren 5 Bits von Faktor nach rechts geschoben und auf dem Stack abgelegt. Beim Verschieben wird mit Null aufgefüllt.

lshl Erwartet auf dem Stack einen Wert vom Typ Long und einen Faktor vom Typ Integer. Der Wert wird um die unteren 6 Bits von Faktor nach links geschoben und auf dem Stack abgelegt.

lshr Erwartet auf dem Stack einen Wert vom Typ Long und einen Faktor vom Typ Integer. Der Wert wird um die unteren 6 Bits von Faktor nach rechts geschoben und auf dem Stack abgelegt. Beim Verschieben wird das Vorzeichen beachtet.

lushr Erwartet auf dem Stack einen Wert vom Typ Long und einen Faktor vom Typ Integer. Der Wert wird um die unteren 6 Bits von Faktor nach rechts geschoben und auf dem Stack abgelegt. Beim Verschieben wird mit Null aufgefüllt.

iand; land Erwartet zwei Operanden auf dem Stack. Auf dem Stack wird das Ergebnis der bitweisen Undverknüpfung der beiden Operanden abgelegt.

ior; lor Erwartet zwei Operanden auf dem Stack. Auf dem Stack wird das Ergebnis der bitweisen Oderverknüpfung der beiden Operanden abgelegt.

ixor; lxor Erwartet zwei Operanden auf dem Stack. Auf dem Stack wird das Ergebnis der bitweisen Exklusiv-Oderverknüpfung der beiden Operanden abgelegt.

Mit diesen Befehlen kann eine Typumwandlung zwischen den Typen `integer`, `long`, `float` und `double` ausgeführt werden. Zusätzlich gibt es drei Befehle um den Typ `integer` in die Typen `byte`, `char` und `short` umzuwandeln.

i2l; i2f; i2d; l2i; l2f; l2d; f2i; f2l; f2d; d2i; d2l; d2f Liest einen Wert mit entsprechendem Typ vom Stack und legt den umgewandelten Wert vom neuen Typ auf dem Stack ab.

int2byte Liest einen Integer vom Stack. Schneidet bis auf 8 Bit alles ab. Erweitert diesen vorzeichenbehafteten Wert wieder auf 32 Bit und legt das Ergebnis auf den Stack.

int2char Liest einen Integer vom Stack. Schneidet bis auf 16 Bit alles ab. Erweitert diesen vorzeichenbehafteten Wert wieder auf 32 Bit und legt das Ergebnis auf den Stack.

int2short Liest einen Integer vom Stack. Schneidet bis auf 16 Bit alles ab. Erweitert diesen vorzeichenlosen Wert wieder auf 32 Bit und legt das Ergebnis auf den Stack.

6.4.9 Verzweigungsbefehle

Bei den bedingten Verzweigungsbefehlen werden zwei Zahlen vom Typ `integer` miteinander oder eine Zahl mit der Zahl 0 verglichen. Stimmt der Vergleich, so wird zu einem 16-Bit Offset verzweigt. Die Befehle `lcmp`, `fcmp` und `dcmp` vergleichen eine Zahl vom entsprechenden Typ mit 0 und legen den Wert -1, 0 oder 1 vom Typ `integer` auf den Stack, wenn die Zahl kleiner, gleich oder größer 0 ist. Mit `if_acmpeq` und `if_acmpne` können Adressen verglichen werden. Ein unbedingter Sprung zu einem 16- oder 32-Bit Offset wird mit `goto` erzwungen. Ein Unterprogramm kann mit `jsr` aufgerufen und mit `ret` verlassen werden.

ifeq; iflt; ifle; ifne; ifgt; ifge Hinter dem jeweiligen Opcode folgen im Code noch die beiden Bytes branchbyte1 und branchbyte2. Auf dem Stack wird ein Integer erwartet. Ist der Wert je nach Befehl gleich, kleiner, kleinergleich, ungleich, größer oder größergleich null, so wird aus den beiden Branchbytes ein 16-Bit-Offset gebildet und an diese Stelle verzweigt.

ifnull; ifnonnull Hinter dem jeweiligen Opcode folgen im Code noch die beiden Bytes branchbyte1 und branchbyte2. Auf dem Stack wird eine Objektreferenz erwartet. Ist sie je nach Befehl gleich oder ungleich NULL, so wird aus den beiden Branchbytes ein 16-Bit-Offset gebildet und an diese Stelle verzweigt.

if_cmpeq; if_cmpne; if_cmplt; if_cmpgt; if_cmple; if_cmpge Hinter dem jeweiligen Opcode folgen im Code noch die beiden Bytes branchbyte1 und branchbyte2. Auf dem Stack werden zwei Integer erwartet. Ist je nach Befehl der erste gleich, ungleich, kleiner, größer, kleinergleich oder größergleich dem zweiten Wert, so wird aus den beiden Branchbytes ein 16-Bit-Offset gebildet und an diese Stelle verzweigt.

lcmp Erwartet zwei Werte vom Typ Long auf dem Stack. Je nachdem, ob der erste Wert kleiner, gleich oder größer dem zweiten ist wird als Ergebnis eine $-1, 0$ oder 1 vom Typ Integer auf dem Stack abgelegt.

fcmpl; dcmpl Erwartet zwei Werte vom Typ Float oder Double auf dem Stack. Je nachdem, ob der erste Wert kleiner, gleich oder größer dem zweiten ist wird als Ergebnis eine $-1, 0$ oder 1 vom Typ Integer auf dem Stack abgelegt. Ist einer der beiden Werte keine gültige Zahl, so wird eine -1 auf den Stack gelegt.

ob der erste Wert kleiner, gleich oder größer dem zweiten ist wird als Ergebnis eine $-1, 0$ oder 1 vom Typ Integer auf dem Stack abgelegt. Ist einer der beiden Werte keine gültige Zahl, so wird eine 1 auf den Stack gelegt.

if_acmpeq; if_acmpne Hinter dem jeweiligen Opcode folgen im Code noch die beiden Bytes branchbyte1 und branchbyte2. Auf dem Stack werden zwei Objektreferenzen erwartet. Sind sie je nach Befehl gleich oder ungleich, so wird aus den beiden Branchbytes ein 16-Bit-Offset gebildet und an diese Stelle verzweigt.

goto, branchbyte1, branchbyte2 Die Programmausführung wird an der Stelle des durch branchbyte1 und branchbyte2 gebildeten 16-Bit-Offsets fortgeführt.

goto_w, branchbyte1, branchbyte2, branchbyte3, branchbyte4 Die Programmausführung wird an der Stelle des durch die Branchbytes gebildeten 32-Bit-Offsets fortgeführt.

jsr, branchbyte1, branchbyte2 Die diesem Befehl folgende Adresse wird auf dem Stack abgelegt. Die Programmausführung wird an der Stelle des durch branchbyte1 und branchbyte2 gebildeten 16-Bit-Offsets fortgeführt.

jsr_w, branchbyte1, branchbyte2, branchbyte3, branchbyte4 Die diesem Befehl folgende Adresse wird auf dem Stack abgelegt. Die Programmausführung wird an der Stelle des durch die Branchbytes gebildeten 32-Bit-Offsets fortgeführt.

ret, vindex Die Programmausführung wird an der Adresse fortgeführt, die in der durch vindex bezeichneten lokalen Variablen enthalten ist.

ret_w, vindex1, vindex2 Aus vindex1 und vindex2 wird ein 16-Bit-Index gebildet. Die Programmausführung wird an der Adresse fortgeführt, die in der durch den Index bezeichneten lokalen Variablen enthalten ist.

6.4.10 Funktionsrücksprungbefehle

Mit den `return`-Befehlen kann von einer Funktion zurückgekehrt werden. Der optionale Rückgabewert kann vom Typ `integer`, `long`, `float`, `double` oder `address` sein. Der Interpreter kann mit `breakpoint` angehalten werden. Die Kontrolle wird an einen Unterbrechungsbehandler übergeben.

ireturn; lreturn; freturn; dreturn; areturn Es wird ein Wert vom entsprechenden Typ auf dem Stack erwartet. Dieser wird auf den Stack der Funktion gelegt, die diese Funktion aufgerufen hat. Der Operandenstack wird gelöscht. Der Interpreter setzt das Programm in der diese Funktion aufrufenden Funktion fort.

return Der Operandenstack wird gelöscht. Der Interpreter setzt das Programm in der diese Funktion aufrufenden Funktion fort.

breakpoint Hält das Programm an und übergibt die Kontrolle an den Unterbrechungsbehandler.

tableswitch; 0-3 bytes pad; default1-4; low1-4; high1-4; offset1..n Die Instruktion ist von variabler Länge. Damit ab default alle Einträge auf geraden Adressen stehen, werden bis zu drei Bytes pad eingefügt. Auf dem Stack wird ein Wert vom Typ Integer erwartet. Ist der Wert kleiner low oder größer high, so wird mit dem default-offset verzweigt. Ansonsten wird mit dem Offset an der Stelle Wert-low+1 verzweigt.

lookupswitch; 0-3 bytes pad; default1-4; npairs1-4; offset-pairs1..n Die Instruktion ist von variabler Länge. Damit ab default alle Einträge auf geraden Adressen stehen, werden bis zu drei Bytes pad eingefügt. Auf dem Stack wird ein Wert vom Typ Integer erwartet. Default1-4 enthält einen Offset, zu dem verzweigt wird, wenn kein passender Eintrag in der Tabelle vorhanden ist. Npairs gibt an wieviele Einträge die Tabelle enthält. Jeder Eintrag besteht aus einem Vergleichswert und einem Offset. Stimmt der Wert vom Stack mit dem Vergleichswert überein, so wird mit dem entsprechenden Offset verzweigt.

6.4.12 Manipulation von Objektfeldern

Das Lesen und Schreiben von Elementen von statischen und dynamischen Strukturen ist mit `getfield`, `putfield`, `getstatic` und `putstatic` möglich.

putfield; indexbyte1; indexbyte2 Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. Der Index zeigt auf eine Referenz zu einer Klasse und einem Feldnamen. Die Referenzen werden aufgelöst, woraus sich der Feldoffset und die Feldbreite für das auf dem Stack referenzierte Objekt ergeben. Auf dem Stack wird eine Objektreferenz und ein Wert erwartet. Das Feld des Objektes wird mit Wert besetzt. Dieser Befehl kann mit 32- und 64-Bit-Feldern umgehen.

getfield; indexbyte1; indexbyte2 Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. Der Index zeigt auf eine Referenz zu einer Klasse und einem Feldnamen. Die Referenzen werden aufgelöst, woraus sich der Feldoffset und die Feldbreite für die auf dem Stack erwartete Objektreferenz ergeben. Das Feld des Objektes wird gelesen und auf dem Stack abgelegt. Dieser Befehl kann mit 32- und 64-Bit-Feldern umgehen.

putstatic; indexbyte1; indexbyte2 Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. Der Index zeigt auf eine Referenz zu einer statischen Klasse. Auf dem Stack wird ein Wert erwartet. Das Feld wird mit Wert besetzt. Dieser Befehl kann mit 32- und 64-Bit-Feldern umgehen.

getstatic; indexbyte1; indexbyte2 Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. Der Index zeigt auf eine Referenz zu einer statischen Klasse. Auf dem Stack wird der Wert des Feldes abgelegt. Dieser Befehl kann mit 32- und 64-Bit-Feldern umgehen.

6.4.13 Methodenaufruf

Die verschiedenen Typen von Methoden können mit den `invoke`-Befehlen aufgerufen werden. Es wird das komplette Methodenenvironment erzeugt und die Methode aufgerufen.

stantenpool der Klasse gebildet. An dieser Stelle ist eine komplette Methodensignatur zu finden. Auf dem Stack werden eine Objektreferenz und die Argumente für die Methode erwartet. Aus der Objektreferenz wird ein Zeiger auf die Methodentabelle des Objektes berechnet. In der Tabelle wird nach der Methode gesucht die zur Signatur paßt. Das Ergebnis ist ein Index in die Methodentabelle der entsprechnenden Klasse. Die Objektreferenz und die Argumente werden auf den Stack der Methode geladen. Die neue Methode wird aufgerufen.

invokenonvirtual; indexbyte1; indexbyte2 Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. An dieser Stelle ist eine komplette Methodensignatur und eine Klasse zu finden. Auf dem Stack werden eine Objektreferenz und die Argumente für die Methode erwartet. In der Tabelle der Klasse wird nach der Methode gesucht die zur Signatur paßt. Das Ergebnis ist ein Index in die Methodentabelle der entsprechnenden Klasse. Die Objektreferenz und die Argumente werden auf den Stack der Methode geladen. Die neue Methode wird aufgerufen.

invokestatic; indexbyte1; indexbyte2 Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. An dieser Stelle ist eine komplette Methodensignatur und eine Klasse zu finden. Auf dem Stack werden die Argumente für die Methode erwartet. In der Tabelle der Klasse wird nach der Methode gesucht die zur Signatur paßt. Das Ergebnis ist ein Index in die Methodentabelle der Klasse. Die Argumente werden auf den Stack der Methode geladen. Die neue Methode wird aufgerufen.

invokeinterface; indexbyte1; indexbyte2; nargs; reserved Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. An dieser Stelle ist eine komplette Methodensignatur zu finden. Auf dem Stack werden eine Objektreferenz und die $nargs - 1$ Argumente erwartet. Aus der Objektreferenz wird ein Zeiger auf die Methodentabelle des Objektes berechnet. In der Tabelle wird nach der Methode gesucht die zur Signatur paßt. Das Ergebnis ist ein Index in die Methodentabelle der entsprechnenden Klasse. Die Objektreferenz und die Argumente werden auf den Stack der Methode geladen. Die neue Methode wird aufgerufen. Im Gegensatz zu invokevirtual wird die Argumentanzahl im Code angegeben.

6.4.14 Verschiedene Befehle

nop Tut nichts.

iinc; vindex; const Erhöht die durch vindex referenzierte lokale Variable um den Wert const.

wide; vindex2 Dieser Bytecode muß einem der Befehle iload, lload, dload, aload, istore, lstore, fstore, dstore, astore oder iinc vorausgehen. Mit Hilfe dieses Befehls kann der sonst nur 8-Bit lange Index auf einen 16-Bit langen Index erweitert werden.

athrow Erwartet auf dem Stack eine Objektreferenz. Die referenzierte Klasse muß eine Unterklasse von Throwable sein. Die entsprechende Ausnahmebehandlung wird ausgeführt.

new; indexbyte1; indexbyte2 Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. Der Index zeigt auf einen Klassennamen der zu einem Klassenzeiger aufgelöst wird. Dann wird eine neue Instanz dieser Klasse erzeugt. Die neue Referenz wird auf den Stack gelegt.

pool der Klasse gebildet. Der Index zeigt auf einen Klassennamen der zu einem Klassenzeiger aufgelöst wird. Auf dem Stack wird eine Objektreferenz erwartet, die durch den Befehl nicht entfernt wird. Es wird überprüft, ob die Objektreferenz vom Typ des Klassenzeigers oder einer seiner Überklassen ist. Ist dies nicht der Fall, so wird eine ClassCastException ausgeführt.

instanceof; indexbyte1; indexbyte2 Aus den Indexbytes wird ein Index in den Konstantenpool der Klasse gebildet. Der Index zeigt auf einen Klassennamen der zu einem Klassenzeiger aufgelöst wird. Auf dem Stack wird eine Objektreferenz erwartet. Es wird überprüft, ob die Objektreferenz vom Typ des Klassenzeigers oder einer seiner Super-Klassen ist. Ist dies der Fall, so wird eine 1 sonst eine 0 auf den Stack gelegt.

monitorenter Erwartet auf dem Stack eine Objektreferenz. Soweit der Monitor des Objektes frei ist, wird mit dem nächsten Befehl fortgefahren und der Monitor gesperrt. Ansonsten wird gewartet, bis der Monitor wieder frei ist.

monitorexit Erwartet auf dem Stack eine Objektreferenz. Der Monitor des Objektes wird wieder freigegeben. Ist dies die letzte Freigabe dieses Objektes können andere Threads den Monitor betreten.

6.5 Vergleich von Java-Virtualcode mit realen CPU's

Die Java-Virtual-Engine mit realen CPU's zu vergleichen ist nicht ganz einfach. Von der Architektur her ist die Java-CPU am ehesten mit dem Novix-4000 zu vergleichen. Diese 1985 gebaute CPU ist dazu gedacht, ausschließlich Code der Programmiersprache Forth abzuarbeiten. Der Registersatz enthält keine Universalregister sondern besteht hauptsächlich aus zwei Stapelzeigern, einem Programmzeiger und einem Arbeitsregister. Da diese CPU aber keine nennenswerten Marktanteile erringen konnte, ist der Vergleich eher uninteressant.

Die meistverkauften CPU's im Personalcomputersektor sind mit Abstand die der 80x86 Reihe. Die Architektur der 80x86-CPU ist mit ca. 15 Jahren schon sehr alt. Diese CPU ist ein klassischer Vertreter der CISC-Architektur (Complex Instruction Set Computer).

Aus der Klasse der RISC-CPU's (Reduced Instruction Set Computer) wie z.B. Alpha, MPC 601 (PowerPC), PA7000, R4400 oder Sparc habe ich den Alpha zum Vergleich ausgewählt.

6.5.1 Register

80x86 Die 80x86-CPU hat die 8 32-Bit Register EAX, EBX, ECX, EDX, EBP, ESI, EDI und ESP. Alle Register enthalten einen einzeln adressierbaren 16-Bit-Teil. Diese 16-Bit-Register heißen AX, BX, CX, DX, BP, SI, DI und SP. Die 16-Bit-Register AX, BX, CX und DX können auch jeweils als zwei 8-Bit-Register AL, AH, BL, BH, CL, CH und DL, DH angesprochen werden. Die sechs 16-Bit breiten Segmentregister lauten CS, DS, ES, FS, GS und SS. Das Statusregister EFLAGS und der Programmzeiger EIP sind 32-Bit breit. Im 80x86 (80x87) sind die 8 Fließkommaregister 80-Bit breit. Diese Register sind ringförmig als Stapel organisiert. Im 16-Bit breiten Statusregister der Fließkommaeinheit ist ein Stapelzeiger enthalten der angibt, welches der physikalischen Register gerade mit ST0 bis ST7 bezeichnet wird.

nicht zur Adressierung zählen, da alle Befehle ein vielfaches von 4 Bytes lang sind. Die 32 Integer-Register sind 64-Bit breit und werden mit R0 bis R31 bezeichnet. R31 nimmt eine Sonderstellung ein, da dieses Register immer 0 enthält. Mit F0 bis F31 werden die 32 Fließkommaregister bezeichnet. Sie sind 64-Bit breit. F31 ist wie R31 immer 0.

Java Alle Register sind momentan 32-Bit breit, wobei die Entwickler eine Erweiterung auf 64-Bit vorgesehen haben. Die Java-Maschine besitzt einen Programmzeiger pc und einen Stapelzeiger optop. Das Register frame zeigt auf die Ausführungsumgebung der momentan auszuführenden Methode. Das Register vars zeigt auf den Anfang der lokalen Variablen.

Alle drei CPU's haben einen Programmzähler. Die 80x86 besitzt mit ESP wie die Java-CPU mit optop einen Stapelzeiger im Gegensatz zum Alpha-Prozessor der gar keinen Stapelzeiger besitzt. Das vars- oder auch das frame-Register kann mit dem EBP-Register des 80x86 verglichen werden, da relativ zu beiden adressiert werden kann.

6.5.2 Datentypen

`byte`	1-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`short`	2-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`int`	4-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement

Tabelle 2: Datenformate der 80x86-CPU

Die 80x86 unterstützt die in Tabelle 2 gezeigten Datentypen.

`long`	8-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`bcd`	10-Byte BCD-Zahl mit 18 Ziffern
`float`	4-Byte IEEE 754 Fließkommazahl einfacher Genauigkeit
`double`	8-Byte IEEE 754 Fließkommazahl doppelter Genauigkeit
`extended`	10-Byte Fließkommazahl

Tabelle 3: Zusätzliche Datenformate des 80x86/80x87-Prozessors

Die Fließkommaeinheit, die ab dem 80486 im Chip integriert ist, unterstützt zusätzlich die in Tabelle 3 aufgelisteten Datentypen.

`byte`	1-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`word`	2-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`longword`	4-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`quadword`	8-Byte vorzeichenbehaftete Ganzzahl 2-erkomplement
`float`	4-Byte IEEE 754 Fließkommazahl einfacher Genauigkeit
`double`	8-Byte IEEE 754 Fließkommazahl doppelter Genauigkeit

Tabelle 4: Datenformate der Alpha-CPU

Die vom Alpha-Prozessor unterstützten Datentypen werden in Tabelle 4 aufgeführt.

Die Datentypen von Java sind in Tabelle 1 in Abschnitt 6.2.1 aufgeführt.

Typkonvertierungsbefehlen. Die kleinste Einheit die der Prozessor in den Speicher schreiben kann, ist ein 64-Bit Wort. In dieser Hinsicht ähnelt er der Java-CPU, die keine „physikalische“ Einheit unter 32-Bit behandelt. Der 80x86-Prozessor kann sogar noch einzelne Bytes bearbeiten oder in den Speicher schreiben.

6.5.3 Adressierungsarten

Der 80x86-Prozessor beherrscht umittelbare, indirekte, absolute, und relative Adressierungsmodi. Zusätzlich können noch Register- und Konstantenoffsets angegeben werden. Es gibt Adressierungsarten mit zusätzlichem Postincrement oder Predecrement. Die Adressierungsarten sind auf bestimmte Register beschränkt. So können z.B. nicht alle Register einen Offset bei einer indirekten Adressierung bilden. Bei den wenigsten Befehle sind alle Adressierungsarten erlaubt. So müßen die Daten oft den Umweg über den Akkumulator EAX zurücklegen, bis sie beim gewünschten Ziel ankommen.

Alle 32 Universalregister der Alpha-CPU sind gleichwertig. Der Alphaprozessor beherrscht die unmittelbare, die indirekte und die relative Adressierung. Die unmittelbare Adressierung ist auf 16-Bit beschränkt. Bei der indirekten Adressierung kann ein 16-Bit Offset angegeben werden. Da das Register R31 immer Null enthält, kann man zumindest einen Adreßraum von 16-Bit absolut adressieren.

Die Java-CPU beherrscht die unmittelbare, die indirekte und die relative Adressierung. Die unmittelbare Adressierung ist auf 16-Bit beschränkt. Die relative Adressierung zum Register pc kann bis zu 32-Bit umfassen. Die indirekte Adressierung über den Stapelzeiger beeinflußt diesen mit Predecrement oder Postincrement. Die relative Adressierung zum Register vars kann bis zu 65536 Wörter ansprechen. Die indirekte Adressierung dient dazu, Daten aus Feldern, Objekten oder Methoden anzusprechen.

Die 80x86-CPU ist gekennzeichnet durch sehr viele Adressierungsmöglichkeiten, die aber nicht symmetrisch sind. Hingegen sind die Adressierungsmodi der Alpha-CPU symmetrisch. Bei der Java-Maschine sind die Adressierungsarten immer fest einzelnen Registern zugeordnet.

6.5.4 Befehle

Die Java-Maschine ist eine reine Stackmaschine. Es gibt nur Lade- und Speicherbefehle mit denen der Speicher angesprochen werden kann. Die eigentlichen Operationen wie z.B. Additionen oder Vergleiche werden nur auf dem Stack ausgefürt. Eine Registermaschine ist der Alpha-Prozessor. Hier werden alle Operanden für Additionen usw. aus Registern geholt und die Ergebnisse in Registern abgelegt. Die 80x86 Architektur hat wenige nicht symmetrische Register, um sowohl Parameter als auch Ergebnisse von Operationen zu speichern. Zusätzlich ist es möglich direkt den Speicher mit den verschiedenen Adressierungsmodi zu referenzieren. Allerdings können auch hier nicht alle Adressierungsmodi von allen Befehlen benutzt werden.

Die meisten Befehle der Java-CPU belegen im Speicher nur ein Byte. Bis auf die beiden Tabellenbefehle sind alle Befehle maximal 5 Bytes lang. Befehle der 80x86-CPU haben eine Länge von einem bis zu maximal 18 Bytes. Alle Befehle des Alpha-Prozessors sind 4 Bytes lang.

Beim 80x86 können für die Datentypen `byte`, `short` und `int` alle oben beschriebenen Adressierungsmodi eingesetzt werden. Die Datentypen der Fließkommaeinheit lassen sich nur mit sehr wenigen Adressierungsmodi bewegen. Aber auch bei den Ladebefehlen sind Asymmetrien anzutreffen. So lassen sich z.B. in einem Befehl nicht gleichzeitig die indirekte, indizierte Adressierung als Quelle und Ziel einsetzen. Es gibt aber auch Ladebefehle, die Speicherblöcke variabler Länge verschieben.

Der Alpha-Prozessor hat bei den Ladebefehlen als Quelle oder Ziel oder beidem immer ein Register. Hingegen ist bei der Java-CPU immer der Stack Quelle oder Ziel des Ladebefehls. Weder dem Alpha-Prozessor noch der Java-Engine ist es möglich, direkt von Speicher zu Speicher Daten zu laden.

Beim Java-Prozessor sind die Ladebefehle typgebunden. Dadurch weiss die Maschine immer, welche Typen der Stack enthält. Da typfalsche Operationen automatisch überprüft werden können, ist es leichter, mögliche Fehler in Programmen aufzufinden. Weder der Alpha-, noch der 80x86-Prozessor kümmern sich um den Typ eines Elementes.

Bei der unmittelbaren Adressierung gibt es beim Java-Prozessor nicht nur den drei Bytes langen Befehl `sipush`, der im Code eine 16-Bit-Konstante enthält, sondern auch sieben weitere Opcodes von einem Byte Länge, die direkt das Laden der Konstanten -1 bis 5 ermöglichen.

Arithmetikbefehle

Bei den Arithmetikbefehlen stellt die Alpha-CPU für die Typen `longword` und `quadword` die Operationen addieren, subtrahieren, multiplizieren und vergleichen zur Verfügung. Für die Fließkommatypen gibt es zusätzlich die Division.

Als Operationen stellt die Java-CPU Addition, Subtraktion, Multiplikation, Division, Restbestimmung, Negation und Vergleich für alle Typen bereit.

Zusätzlich kann die 80x86-Fließkommaeinheit noch die Funktionen Sinus, Cosinus, Tangens, Quadratwurzel und Logarithmus berechnen. Trotz dieses enormen Hardwarefunktionsumfanges ist die 80x86-CPU beim Berechnen der transzendenten Funktionen der Alpha-CPU in der Geschwindigkeit unterlegen.

Bei allen CPU's gibt es Befehle um Fließkommatypen in Ganzzahltypen und umgekehrt zu wandeln.

Logische Befehle

Alle drei CPU's stellen Befehle für die logischen Operationen Und, Oder, Exklusivoder und bitweises Schieben für die Ganzzahltypen bereit.

Sprungbefehle

Es gibt bei allen drei Prozessoren bedingte Verzweigungsbefehle, die relativ zur Position des Programmzeigers verzweigen. Bei der 80x86-, Java- und Alpha-CPU beträgt der Offset 8-, 16- und 21-Bit. Lange unbedingte Sprünge und Unterprogrammaufrufe können bei der Alpha-CPU nur indirekt adressiert werden. Bei der Java-CPU können diese Sprünge nur relativ ausgeführt

möglich diese Sprünge absolut und indirekt zu adressieren.

Bei den Unterprogrammaufrufen werden die Rücksprungadressen bei allen CPU's unterschiedlich behandelt. So speichert die 80x86 die Rücksprungadresse auf dem Stapel und holt sie bei der Rückkehr wieder von dort zurück. Der Alpha-Prozessor benutzt eines der 32 Register um die Adresse zu speichern. Hier muß ein Stapel per Software nachgebildet werden. Der Java-Prozessor speichert beim Unterprogrammaufruf die Adresse auf dem Stapel und holt aber die Rücksprungadresse aus einer lokalen Variablen.

Seminarbeitrag von **Oliver Reiff**

7.1 Was heißt interpretieren?

Dieser Abschnitt gibt einen ersten Einblick in das Thema Interpretieren allgemein, also unabhängig von der Programmiersprache Java. Es zeigt Unterschiede und Gemeinsamkeiten in der Realisierung des allgemeinen Schemas anhand verschiedener Beispiele.

7.1.1 Prinzip der Interpretierer

Bis Anfang der 70er Jahre stellte sich die Frage nach Interpretierern nicht oder nur ganz vereinzelt. Der Hauptnachteil der interpretierten Programme, nämlich die vergleichsweise geringe Effizienz, zwang damals die Software–Entwickler, Assembler oder eine Übersetzer–Hochsprache zu verwenden.

So ist es nicht verwunderlich, daß auch die ersten Interpretierer eigentlich Maschinenprogramme waren, die Maschinensprache interpretierten. Erst auf den zweiten Blick wird klar, wie man Maschinenprogramme interpretieren kann: anstelle von Assemblerzeilen schrieb der Programmierer Datenzeilen in Assembler, in die komplexere Befehle hineincodiert waren. Diese wurden vom Hauptprogramm sequentiell gelesen und abgearbeitet. So kam der Begriff des Interpretierens auf, der im Gegensatz zum Compilieren bzw. Assemblieren stand.

Beim Compilieren gibt der Programmierer einen Quellcode vor, den das Werkzeug Übersetzer direkt in Maschinensprache übersetzt. Natürlich benutzen Übersetzer interne Zwischensprachen, um effizienten Code zu erzeugen oder um den Übersetzer in mehrere unabhängige Stücke zu gliedern. Diese Zwischensprachen sind nach außen hin aber nicht sichtbar und meistens auch nicht vernünftig lesbar, da sie aus attributierten Grammatiken und Baumstrukturen bestehen. Aus Sicht des Programmierers werden also alle Anweisungen in einem Durchgang und in einem Stück aus dem Quellcode erzeugt, und das Programm ist sofort einsetzbar.

Im Gegensatz dazu ist der Interpretierer ein Werkzeug, das eine Anweisung nach der anderen in drei Schritten bearbeitet:

- Lesen
- Decodieren
- Ausführen

```
Solange (noch Anweisungen da)
{
  Lies Anweisung
  Übersetze Anweisung
}
Optimiere
Schreibe Programm
```

```
Solange (noch Anweisungen da)
{
  Lies nächste Anweisung
  Erkenne nächste Anweisung
  Führe nächste Anweisung aus
}
Beende Programm
```

Das generelle Prinzip, nach dem Interpretierer vorgehen, steht somit fest – von der Definition einer Anweisung einmal abgesehen. Genau diese Defintion bestimmt aber viele charakteristische Parameter des Interpretierers.

Die Grenzen zwischen Compilieren und Interpretieren sind dennoch nicht fest definiert, sondern fließend. Dies wird schon dadurch deutlich, daß ein Prozessor nichts anderes ist, als ein Interpretierer für Maschinensprache.

7.1.2 Warum interpretieren?

Wenn interpretierte Programme langsamer sind als compilierte, warum soll man dann überhaupt interpretieren? Geschichtlich gesehen ist einer der ersten Gründe, der für interpretierte Programme sprach, deren Länge. Damals gab es nur eine wichtigere Größe, die Programme mehr beeinflußte als deren Geschwindigkeit: der Bedarf an Hauptspeicher.

Lesbarer Quellcode ist kürzer als Maschinensprache, da meist eine 1:n–Umsetzung stattfindet, d.h. eine Anweisung im Quelltext entspricht einer ganzen Reihe von Maschineninstruktionen. Im Gegensatz dazu kann man bei interpretierten Sprachen oft eine 1:1–Umsetzung erreichen. Zur Länge des Programms kommt aber noch die Größe des eigentlichen Interpretierers, der parallel dazu im Hauptspeicher gehalten werden muß.

Mit dem sog. *Threaded Code* wurde eine ebenso einfache wie leistungssarke Klasse von Interpretierern entwickelt, bei der sich die Länge des Interpretierskerns kaum auf die Programmlänge auswirkt (vgl. Abschnitt 7.2.5).

Auch heute kommt der Kompaktheit von Programmen wieder eine wichtige Rolle zu. Zwar haben heutige Systeme eine gut ausgebaute Speicherhierarchie mit geradezu riesigen Adreßräumen. Dennoch gibt es wieder einen Engpaß: die Übertragung von Programmen über Netzwerke. Auch wenn die Netze in Zukunft weiter ausgebaut werden, muß auf die Übertragungszeiten verstärkt Rücksicht genommen werden. Diesen Punkt gilt es auch bei Multiprozessorsystemen zu beachten, wenn Codefolgen an einzelne Prozessorelemente verteilt werden müssen.

Die Übertragung über Netzwerke bringt noch eine weitere Anforderung mit sich, die Programme heute erfüllen müssen, um ein großes Marktsegement zu erobern: Maschinenunabhängigkeit.

Compilierte Programme sind immer auf genau eine Rechnerarchitektur zugeschnitten, die aus Prozessortyp, Betriebssystem und Peripherie besteht. Da nicht nur die Art der drei genannten Faktoren, sondern oft auch deren Versions– oder Revisionsnummer ausschlaggebend ist, kommt es hier zu einer unüberschaubaren Anzahl von Architekturen. Zwar kann man alle drei Größen heute mit einigem Aufwand simulieren, was aber wieder Zeit kostet und vor allem oft zu schlechten Kompromissen führt. Diese Simulation ist nichts anderes als ein äußerst komplexer Interpretiervorgang.

Der Aufwand zur Erstellung dieser Software für jede neue Architektur ist meist sehr hoch. Neben einem Übersetzer müssen alle zu übertragenden Programme in vielen Details an die

Betriebssystem komplett unterschiedlich realisiert sind, ist eine Umsetzung sehr langwierig.

Während dieser Aufwand bei zu compilierenden Programmen jedes Mal neu anfällt, ist er bei Interpretierern nur ein einziges Mal gegeben: der Interpretierer muß für das neue System angepaßt werden, dann laufen alle interpretierten Programme genauso wie auf anderen Architekturen ab. Um dies nicht nur in der Theorie zu gewährleisten, müssen Interpretierer besonders sorgfältig entwickelt werden.

Der Vorteil ist aber noch größer: Programme können nicht nur auf all diesen Architekturen ablaufen, was eine weite Verbreitung bei weniger Kosten ermöglicht, sie können darüber hinaus von einer Plattform auf die andere wechseln. Dieses Phänomen der Programm– bzw. Objektmigration wird in den Netzen der Zukunft eine immer größere Rolle spielen.

Ein weiterer wirtschaftlicher Vorteil bei der Nutzung von Interpretierern ist die kürzere Programmentwicklungszeit, die sich durch geringere *Turn–Around–Zeiten* erklärt. Beim Testen compilierter Programme müssen diese nach jeder Änderung komplett neu übersetzt werden. Ein Zeitanteil, der trotz Werkzeugen wie *make* oder sog. *smart recompilation (Selektive Nachübersetzung)* gerade bei großen Programmen nicht zu vernachlässigen ist. Hinzu kommt das Problem der Versionskontrolle, wenn ein Team an einem komplexen Projekt arbeitet.

In einer Entwicklungsumgebung für einer Interpretersprache genügt meist ein Tastendruck, um das Programm zu starten, zumindest wenn keine Vorverarbeitung für den Interpretierer erfolgt. Auch das Problem mit unterschiedlichen Versionen fällt wesentlich kleiner aus, da die einzelnen Module schnell zusammengestellt werden können. Eine aufwendige Übersetzung, wie sie bei komplexen Projekten die ganze Nacht dauern kann, entfällt.

Allerdings reizt diese Technik viele Programmierer verstärkt zu einer Try–and–Error–Implementierung, da sie unsichere Stellen *ja schnell mal ausprobieren* können. Mit genügend Selbstdisziplin und Programmiererfahrung ergibt sich aber genau daraus ein weiterer Vorteil beim Testen der Programme: die bessere Behandlung von Fehlern.

Üblicherweise filtern Übersetzer nur Syntax– und einige Semantikfehler aus dem Quellcode, indem sie sich weigern, diesen zu übersetzen. Zur Fehlersuche wird meist auf ein Debugging–Werkzeug verwiesen, in dem der Programmablauf nachvollzogen werden kann. Eine vernünftige Fehlerbehandlung außerhalb dieser geschützten Umgebung gibt es selten. Eventuell wird noch eine Fehlernummer ausgegeben, deren tieferer Sinn dann im Handbuch nachzulesen ist, oder die Fehlerbehandlung wird dem Betriebssystem überlassen, was selten zu besseren Ergebnissen führt.

Der Programmierer sollte also jede Fehlerquelle vorhersehen und ausprogrammieren, was zu einem gewissen Overhead an Codelänge und Ausführungszeit führt, aber vor allem sehr aufwendig werden kann, wenn er wirklich aussagekräftige Fehlermeldungen generieren will.

Interpretierer haben es da einfacher: sie müssen erst zur Laufzeit entscheiden, ob die Anweisung einen Fehler enthält oder nicht. Für syntaktische Fehler ist dies oft ein Nachteil, da man den Fehler schon vorher hätte entdecken können, was dem Programmierer unter Umständen einige Zeit erspart hätte, um zur eigentlichen Fehlerstelle zu gelangen. Deshalb sind viele Interpretierer dazu übergegangen, nur noch syntaktisch korrekten Code zu interpretieren, d.h. entweder selbst den Quellcode zuerst auf syntaktische Mängel zu prüfen und dann zu starten, oder Code in einer Zwischensprache zu akzeptieren. Da dieser Code von einem Übersetzer erzeugt wurde, sollte er korrekt sein, wenn er nicht versehentlich oder bewußt manipuliert wurde.

Diese Tatsache kommt vor allem objektorientierten Sprachen zu gute: Die Angabe einer Anweisungen wie die Addition zweier Objekte ist immer möglich — ungeachtet der Objekttypen. Erst

zweier Zahlen kann so entsprechend ihrer Typen durchgeführt werden, während die Addition eines Fensters zu einem Feld von Bankkonten wahrscheinlich zurückgewiesen wird, es sei denn sie wurde entsprechend definiert. Dies erlaubt ein semantisch nicht ganz so striktes Sprachmodell, da die Defintion der Addition während der Laufzeit immer wieder verändert werden könnte.

Desweiteren verfügen Interpretierer über zusätzliche Informationen zu gerade aufgetretenen Fehlern und können deshalb detailiertere Fehlermeldungen erzeugen. Besonders relevant ist die genaue Lokalisierung des Fehlers. Da der Interpretierer meist den Quellcode kennt, kann er die exakte Fehlerstelle, also Zeile und Spalte im Quellcode ausgeben, bzw. in einer Interpretierumgebung den Texteditor veranlassen, diese Stelle hervorzuheben.

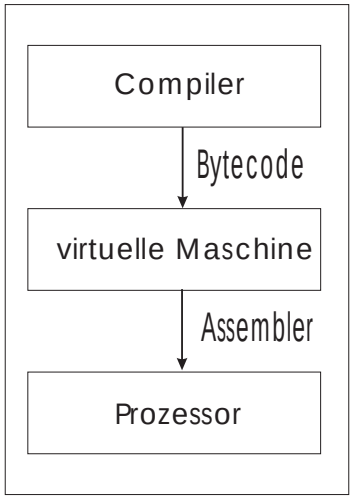
Das Debugging wird dadurch einfacher, daß innerhalb eines Interpretierers Möglichkeiten zur Visualisierung von Variablen und Kontrollfluß auf Quelltextebene eingebaut werden. Ein Übersetzer müßte jedem Maschinenprogramm diese Information zusätzlich mitgeben, die dann der Debugger wieder entsprechend auswerten muß.

Ein Gebiet, auf dem sich Interpretierer nicht durchsetzen konnten oder werden, stellen die zeitkritischen Systeme dar. Sieht man einmal von Phänomenen ab, die durch Cache und virtuelle Speicherverwaltung verursacht werden, sind interpretierte Programme stets langsamer. Abhilfe versprechen hier nur sog. *Just in Time–Übersetzer*, auf die in Abschnitt 7.6 eingegangen wird.

Ein weiterer Nachteil, den manche Programmiersprachen für Interpretierer mit sich bringen, ist die Verteilung des Quellcodes in seiner originalen Form. Dadurch können die Programme ohne Aufwand eingesehen und verändert werden.

Maschinenprogramme können auch wieder lesbar und damit veränderbar gemacht werden. Der Vorgang der Dissamblierung ist jedoch mit erheblichem Aufwand verbunden. Auch für Interpretiererprogramme kann man den Aufwand zur Dekodierung erhöhen, indem eine Zwischensprache verwendet wird. Diese Zwischensprache besteht dann aus Anweisungen für eine *virtuelle Maschine*.

7.1.3 Konzept der virtuellen Maschine



Um das Konzept der virtuellen Maschinen besser verstehen zu können, zuerst ein Beispiel aus der Prozessorarchitektur: Hier ist es üblich, zwischen *Architektur* und *Implementierung* zu unterscheiden. Beispielsweise ist PowerPC eine Architektur, während der PowerPC 601 eine Implementierung dieser Architektur darstellt. Etwas Vergleichbares gibt es auch bei Interpretierern. Diese sind Implementierungen der virtuellen Maschine. Darunter versteht man die Spezifikation ihrer Funktionsweise, d.h. im wesentlichen den Befehlsumfang. Folglich sind verschiedene Java–Interpretierer Implementierungen ein und derselben virtuellen Maschine. Oft wird aber auch die virtuelle Maschine als Interpretierer bezeichnet.

Das Prinzip der virtuellen Maschine ist selbstverständlich schachtelbar. Die unterste Ebene bildet dabei der Prozessor. Das darüberliegende Betriebssystem kann ebenso als virtuelle Maschine

Prozessorprogramme zur Verfügung stellt.

Jede dieser Schichten in einem solchen virtuellen Maschinenmodell verdeckt die darunter liegende Schicht ganz oder teilweise, so daß es oft möglich ist, eine Zwischenschicht zu entfernen, indem man die darüberliegende Schicht direkt auf der darunterliegenden programmiert. Dies ist beispielsweise dann der Fall, wenn die virtuelle Maschine mit dem Prozessor identisch ist, wie bei den geplanten Java–Prozessoren. Die andere Möglichkeit besteht in der bereits erwähnten Just in Time–Übersetzung.

Virtuelle Maschinen haben gegenüber konkreten Maschinen den Vorteil, daß sie leicht durch Software verändert und erweitert werden können. Bei den heutigen Hardware–Entwicklungskosten ein nicht zu unterschätzender Vorteil. Allerdings muß man auch hier berücksichtigen, daß es nur zu Verwirrung führt, wenn unterschiedliche Versionen der virtuellen Maschine im Umlauf sind, wie es ja bei den Architekturen im Lauf der Zeit eher zum Nachteil wurde.

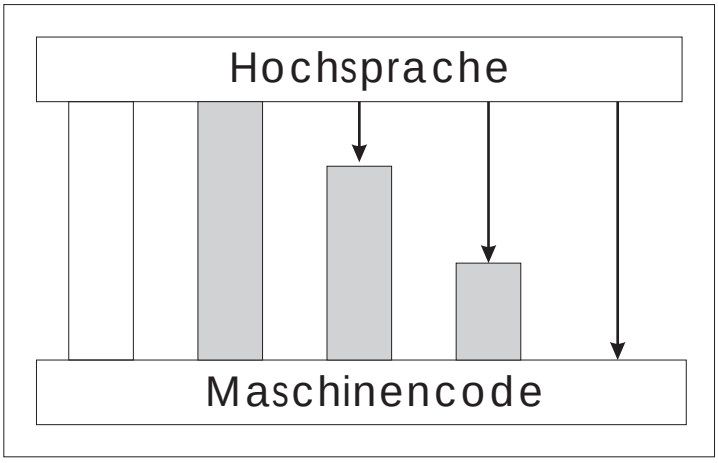
7.2 Unterschiedliche Interpretierer

In der Praxis gibt es die unterschiedlichsten Interpretierer. Von der Shell, die in so gut wie jedem Betriebssystem existiert, bis zum interpretierten Maschinencode sind alle Varianten vertreten. Anhand eines Beispiels werden die Stärken und Schwächen einzelner Vertreter verdeutlicht.

7.2.1 Spektrum unterschiedlicher Interpretierer

Wie bei der Definition des Interpretierers gesehen, hängen die meisten charakteristischen Merkmale von der Art der Anweisungen ab, die er verarbeiten kann. Bei der Maschinensprache ist eine Anweisung jeweils ein Prozessorbefehl. In typischen Interpretersprachen stellt jeweils eine Zeile des Quellcodes ein oder mehrere Befehle dar, während in einem Shell–Skript einer beliebigen Shell dagegen Anweisungen ganze Programme aufrufen. Auch eine Spracheingabe, die aus einigen Kilobyte besteht, kann eine einzige Anweisung sein. Dazwischen gibt es ein breites Spektrum an Anweisungsarten, bzw. an deren Funktionsumfang.

Auch innerhalb einer virtuellen Maschine ist der Beitrag einer einzelnen Anweisung zum gesamten Programm und damit deren Ausführungszeit sehr unterschiedlich. So gibt es in der Definition der virtuellen Java–Maschine simple Sprungbefehle ebenso wie Konstruktoren für Fensterelemente.



von links nach rechts:

- Assembler
- Threaded Code
- BASIC
- Java
- Übersetzersprache

grammiersprachen. Die Länge der Pfeile zeigt den Teil der Arbeit an, die der Übersetzer leistet. Die grauen Balken verdeutlichen den *Abstraktionsgrad* des Zwischencodes, also seinen Abstand zur ausführbaren Maschinensprache. Zur Übersetzung von Assembler ist keine Zwischensprache notwendig.

7.2.2 Die Shell

Dieser Interpretierer wird von den meisten Programmierern täglich in Anspruch genommen. Den wenigsten ist dabei klar, daß sie einen Interpretierer benutzen, der ihre Eingaben nacheinander liest, erkennt und ausführt. Im *Einzelschrittmodus* dieses Interpretierers wird dies auch nicht so leicht deutlich. Wendet man sich dagegen der Skriptprogrammierung zu, wird klar, daß die Shell genau unserer Definition eines Interpretierers genügt.

Es gibt die unterschiedlichsten Shell–Varianten auf allen Betriebssystemem. Sie alle haben einen unterschiedlichen Befehlssatz, sind also untereinander meist inkompatibel. Dennoch ist ihnen gemeinsam, daß sie Eingaben zeilenweise interpretieren und die dort angegebenen Programme ausführen. Diese *Programme* müssen nicht unbedingt eigenständige Anwendungsprogramme sein, es kann sich auch um solche Konstrukte wie IF–THEN–ELSE oder Sprungmarken handeln.

Ein entscheidender Vorteil dieser virtuellen Maschinen liegt auf der Hand: ihre offene Definition. Mit jeder neuen Anwendung des Systems wächst ihr *Sprachumfang*. Es ist kein Shell–Skript–Übersetzer nötig und auch der eigentlich Interpretierer ist recht einfach zu realisieren.

7.2.3 BASIC

Dieser *Beginner's All purposes Symbolic Intruction Code* ist wohl der typischste Interpretierer überhaupt. BASIC–Interpretierer gibt es auf fast allen Maschinen, vor allem auf denen im Heimbereich, wobei es einen kleinen kompatiblen Sprachkern gibt, der aber auf den einzelnen Betriebssystemen die unterschiedlichsten Sprachdialekte als Beigabe erhalten hat. Vor allem in der frühen Zeit der Homecomputer war BASIC sehr gefragt, was sicherlich an der einfachen Sprachsyntax begründet lag.

Ein typischer Vertreter dieser Art interpretiert direkt den Quellcode, den der Programmierer eingibt. Dabei liest und vergleicht er zeichenweise, bis er einen Befehl gefunden hat, bzw. er sicher ist, daß ein syntaktischer Fehler vorliegt.

Fortgeschrittenere Versionen erledigen eine gewisse Vorverarbeitung vor dem eigentlichen Interpretieren: das sog. *Tokenisieren*. Dabei wird den einzelnen Befehlen wie *GOTO*, die im Quellcode aus mehreren Zeichen bestehen, ein einziges Zeichen zugeordnet, eben das sog. *Token*. Nicht nur Befehle, auch Variablen– und Prozedurnamen können so behandelt werden.

Der erzeugte tokenisierte Code wird leichter interpretierbar, da man keine umständlichen Textvergleiche durchführen muß, sondern die Tokens relativ direkt interpretieren kann. Außerdem kann der Interpretierer vor dem Ausführen bereits Syntax–Fehler erkennen und den Programmierer darauf hinweisen.

Um eine noch höhere Leistung von BASIC–Systemen zu erlangen, wurden auch BASIC–Übersetzer entwickelt, die den Quellcode direkt in Maschinensprache übersetzten, um die Vorteile von BASIC bei schnellerer Programmausführung zu erhalten.

Das Prinzip von Java ist eine Weiterentwicklung des Tokenisierens: Ein Übersetzer wandelt den Quellcode in Bytecode um, der dann von einer virtuellen Maschine interpretiert wird. Jedoch wird keine 1:1–Umsetzung vorgenommen, sondern der komplexe Java–Quellcodebefehl wird in mehrere einfache Bytecodes zerlegt.

Auch hier existieren unterschiedliche Implementierungen. Grundlage ist jedoch immer dieselbe virtuelle Maschine, was größtmögliche Kompatibilität sichert. Durch die Trennung des Übersetzers vom Interpretierer können hier zwei Fehlerklassen deutlich von einander unterschieden werden: statische und dynamische Fehler.

Während statische Fehler überwiegend syntaktischer Natur sind und beim Compilieren erkannt werden, treten die dynamischen Fehler erst beim Programmablauf auf. Ein umfangreiches Laufzeitsystem sichert eine gute Fehlerbehandlung auch unter normalen Umständen, also nicht nur in einem isolierten Debugging–System.

Der genauere Aufbau eines Java–Interpretierers wird in Abschnitt 7.3 beschrieben.

7.2.5 Threaded Code

Threaded Code ist die einfachste Art des Interpretierens und wurde Anfang der 70er Jahre entwickelt. Das interessante daran ist, daß ein Threaded Code–Programm keinen externen Interpretierer benötigt, sondern diesen eingebaut hat.

Der eigentliche Interpretierer ist enorm kurz und kann von jedem Programmierer ohne Probleme in einer beliebigen Programmiersprache implementiert werden. Das hier angegebene Beispiel ist aus historischen Gründen in Assembler abgefaßt und besteht nur aus zwei Zeilen:

```
        MOVE   #DATA, R0
LOOP:   JUMP   (R0)+
```

Das Programm initialisiert zunächst das Register R0 mit dem Beginn der Daten, dann springt es bei der Marke *LOOP* an die im Register R0 angegebene Adresse und erhöht den Wert in R0, so daß dieser auf den nächsten Interpretiererbefehl zeigt. Ein zugehöriger Datenteil könnte so aussehen, wobei *DC.L* jeweils Speicher für eine Adresse oder Konstante gleicher Länge reserviert.

```
DATA:  DC.L   ADD
       DC.L   VAR_A
       DC.L   VAR_B
       DC.L   STORE
       DC.L   VAR_C
       DC.L   PRINT
       DC.L   VAR_C
       DC.L   END
```

Im ersten Wort steht die Adresse des nächsten Befehls, danach kommen seine Parameter, dann der nächste Befehl und dessen Parameter. In diesem Fall soll eine Addition ausgeführt werden, Parameter sind die Adressen der Variablen A und B. Danach wird das Ergebnis der Variablen C zugewiesen und ausgegeben.

PRINT und *END* implementiert werden. Dies geschieht für jeden Befehl einzeln, genügt aber immer folgendem Schema: alle Parameter müssen gelesen und das Register R0 entsprechend inkrementiert werden, ansonsten versucht der Interpretierer die Adresse einer nachfolgenden Variablen als Befehlsadresse zu interpretieren, was zwangsläufig scheitern muß. Nach Beendigung der Befehlsausführung wird wieder zur Marke *LOOP* gesprungen, wo der nächsten Befehl aufgerufen wird.

Während die Implementierungen von *PRINT* und *END* stark von der zugrundeliegenden Architektur abhängen, läßt sich für die Befehle *ADD* und *STORE* folgendes Schema angeben, das mit großer Wahrscheinlichkeit auf allen Maschinen ähnlich umgesetzt werden kann:

```
ADD:    MOVE    (R0)+, R1
        MOVE    (R0)+, R2
        ADD.L   R2,R1
        JUMP    LOOP

STORE:  MOVE    (R0)+,R2
        MOVE    R1,(R2)
        JUMP    LOOP
```

Um die Effizienz weiter zu steigern, kann man auch gleich den nächsten Befehl mit *JUMP (R0)+* anspringen, anstatt mit *JUMP LOOP* wieder in die Schleife zurückzukehren. Das wesentliche Strukturmerkmal des Interpretierers ist dann jedoch völlig verwischt. Weitere Varianten des Threaded Codes gehen noch weiter und ersetzen die Befehlsadressen durch Nummern, so daß deren Adresse über eine Befehlstabelle ausgelesen werden kann, was nicht ganz so effizient ist, aber die Flexibilität steigert.

Zur Codelänge ist zu sagen, daß der Kern des Interpretierers klein ist, das eigentliche Programmsegment relativ kompakt bleibt, aber die einzelnen Unterprogramme am meisten zur endgültigen Codelänge beitragen. Bei gut strukturierter Assemblerprogrammierung mit vielen übersichtlichen Unterprogrammaufrufen, ändert sich beim Übergang zu Threaded Code fast nichts.

Interessant ist auch die Tatsache, daß beispielsweise der FORTRAN IV–Übersetzer auf der PDP–11 genau mit diesem Prinzip arbeitete, was die Codegenerierung ziemlich stark vereinfachte, bei der Optimierung jedoch eher Probleme bereitete.

7.2.6 Ein Beispielprogramm

Das Beispiel aus dem letzten Abschnitt wird nun in die verschiedenen anderen Sprachen übersetzt. Dabei ist die BASIC–Umsetzung wohl am einfachsten:

```
C  = A + B
PRINT C
```

Ein tokenisierender BASIC–Interpretierer würde aus dem Programm die überflüssigen Leerzeichen streichen und jedem Befehl oder Operator ein Token zuordnen, so daß das Programm so aussieht, wobei spitze Klammern jeweils ein 1–Byte–Token umschließen:

```
<VARIABLE><C><ZUWEISUNG><VARIABLE><A><OPERATOR><PLUS><VARIABLE><B>
<PRINT><VARIABLE><C>
```

viele Leerzeichen oder Zeilenumbrüche hätte einfügen können. Besonders textuelle Anweisungen und lange Variablennamen werden dadurch gekürzt, während sich die Länge mathematischer Operatoren kaum ändert, da sie selten länger als zwei Zeichen sind.

Die Java–Variante ist wegen der Klassen– und Variablendeklaration etwas länger. Den Kern bilden aber auch hier die beiden Zeilen mit der Addition:

```
class addition {
 public static void main( String args[]) {
   int a=1, b=1;
   int c=a+b;
   System.out.println(c);
 }
}
```

Allerdings ist das nicht der Code, den es zu interpretieren gilt. Der Programmtext wird in einen Zwischencode, den Bytecode, übersetzt. Dieser sieht dann so aus.

```
 1 iconst_1
 2 istore_1
 3 iconst_1
 4 istore_2
 5 iload_1
 6 iload_2
 7 iadd
 8 istore_3
 9 getstatic #2 <Field java.lang.System.out Ljava/io/PrintStream;>
10 iload_3
11 invokevirtual #3 <Method java.io.PrintStream.println(I)V>
12 return
```

Die ersten 4 Zeilen sind die nötige Initialisierung der Variablen a und b auf 1. Die Zeilen $5-8$ leisten dann die Addition, indem zwei Operanden auf den Stack geladen und addiert werden. Zeile 9 ist nötig, um den Ausgabestrom ansprechen zu können, der dann in Zeile 11 angesprochen wird, in dem die Ausgabefunktion aufgerufen wird. Das *return* am Ende ist mit dem *END* im Threaded Code–Modell vergleichbar und bewirkt, daß das Programm beendet wird.

Der Vollständigkeit halber sei hier noch ein Vorschlag angefügt, wie man dieses Programm als Shell–Skript realisieren könnte: Da Shells normalerweise nur Zeichenketten als einzigen Datentyp kennen, müßte man ein Programm erstellen, das seine Parameter als mathematischen Ausdruck auffaßt und das Ergebnis z.B. als Zeichenkette in einer Datei ablegt. Danach wird es per *more* oder *type* auf dem Bildschirm angezeigt.

1.2.7 Abschließender Vergleich

	Shell	BASIC	Java	Threaded Code	Übersetzer–Sprache
Syntax	einfach (Programm mit jeweils unterschiedlichen Parametern)	imperative Hochsprache	objektorientierte Hochsprache	Assembler (Befehlsadressen und Daten)	beliebig
Zwischensprache	keine	tokenisiertes BASIC	Bytecode	keine	nur intern
Übersetzer	keiner	zur Beschleunigung möglich	notwendig für Transformation in Bytecode, zur Beschleunigung JIT–Übersetzer möglich	keiner	notwendig
Interpretierer	extern notwendig	extern notwendig	extern, Appletviewer oder WWW–Browser	integriert	keiner
Fehlererkennung	bleibt Programmen überlassen	eventuell statisch, immer dynamisch	statisch und dynamisch	keine	statisch, manchmal dynamisch durch Zusatzcode
Maschinennähe	Programmebene, unabhängig	fern	Quellcode fern, Bytecode nah	identisch	unterschiedlich
Geschwindigkeit	langsamer Interpretierer, gute Programme	langsam	mittel	fast optimal	optimal

Dieser Abschnitt befaßt sich ausschließlich mit Java–Interpretierern. Es beschreibt sowohl das Konzept als auch die Implementierungen bestehender Interpretierer. Es kann aber auch als eine Anleitung zur Implementierung weiterer Java–Interpretierer dienen.

7.3.1 Architektur der virtuellen Java–Maschine

Die virtuelle Maschine wurde von der Firma SUN bewußt einfach gehalten, um die Portierung auf andere Betriebssysteme möglichst einfach zu gestalten. Grundsätzlich handelt es sich um eine stack–basierte Architektur.

Wichtigste Datenstruktur ist somit der Operandenstack. Dieser Stack wird von den meisten Befehlen modifiziert, d.h. Befehle legen Daten ab, holen Daten oder berechnen neue Daten aus den auf dem Stack liegenden Parametern. Es gibt keinerlei Befehle, die den Stack umgehen können. Das Register, das die Referenz auf den Stack enthält, heißt *OPTOP* und zeigt, auf das oberste Element des Stacks.

Die virtuelle Java–Maschine kommt mit genau drei zusätzlichen Registern aus, die alle jeweils 32–Bit breit sind. Das wichtigste ist der Befehlszähler *PC*: Er zeigt immer auf den nächsten Bytecode–Befehl, der auszuführen ist.

VARS dient zur Referenzierung lokaler Variablen, zeigt also auf den Bereich, in dem jeweils die lokalen Variablen abgelegt werden. Da alle Variablen entweder einen oder zwei Speicherstellen belegen, kann man über die Nummer der Variablen diese dann adressieren. 64–Bit lange Variablen belegen einfach zwei Nummern, von denen jedoch nur die erste als Referenz benutzt wird.

FRAME ist ein Zeiger auf die Ablaufumgebung. In ihr stehen wichtige Daten zum dynamischen Binden und zur Behandlung von Fehlern und Ausnahmen, wie sie in einem späteren Abschnitt vorgestellt werden.

Die virtuelle Maschine kennt nur neun Basisdatentypen. Auch hier wurde darauf geachtet, mit möglichst wenig unterschiedlichen Typen auszukommen, um die Portierung zu erleichtern. Ganzzahlwerte stehen mit einer Breite von 1, 2, 4 und 8 Byte zur Verfügung, die jeweils im 2er–Komplement gespeichert werden. Hinzu kommen zwei unterschiedlich lange und genaue Gleitkommaformate, die dem IEEE-754 Standard genügen müssen.

Textzeichen werden im jeweils 2 Byte langen Format für Unicode Character abgelegt. Diese Größe ermöglicht internationale Kompatibilität über die engen Grenzen des ASCII–Codes hinaus, also beispielsweise das Einbinden kyrillischer oder japanischer Schriftzeichen.

Neben diesen Standarddatentypen kennt die Java–Maschine noch Objekte und Rückkehradressen. Während die 4 Byte lange Rückkehradresse nur von den Sprungbefehlen verwendet wird, und ihr Einsatzbereich deshalb sehr begrenzt ist, kommt der Objektreferenz eine deutlich höhere Wichtigkeit zu.

7.3.2 Befehlssatz der virtuellen Maschine

Zunächst scheint die Zahl von 200 zu implementierenden Befehlen für die virtuelle Maschine recht hoch. Auf den zweiten Blick sieht man jedoch, daß sie bis auf wenige Ausnahmen leicht zu implementieren sind. So existiert der Befehl *iload* mit einem Variablenindex, der die zugehörige

Variablennummer quasi fest im Befehl integriert haben.

Von diesem Typ gibt es 14 Befehlsgruppen. Sinn dieser redundanten Kurzformen für Befehle ist es lediglich, den Code möglichst klein zu halten und das Interpretieren zu beschleunigen. Bei Konstanten werden jeweils kleine Zahlen, also 0 bis 4, benutzt, da diese am häufigsten auftauchen, wenn es um nichtnumerische Programme geht. Die Variablen mit kleinem Index werden deshalb bevorzugt behandelt, weil sie rein statistisch über alle Programme am häufigsten vorkommen, da die einzelnen Methoden oft recht knapp und mit wenigen lokalen Variablen gehalten sind. Allerdings nutzen Übersetzer bisher diese Möglichkeit nicht aus, die häufig genutzten Variablen an den Anfang der lokalen Variablentabelle zu schreiben.

Der gesamte Befehlsvorrat läßt sich in folgende Klassen einteilen:

- Ablegen lokaler Variablen und Konstanten auf dem Stack
- das Speichern in lokale Variable
- Methodenaufruf
- Manipulation von Stack, Feldern und Objekten
- arithmetische und logische Operationen
- Befehle zur Konvertierung
- Sprünge, Rücksprünge und Sprünge über Tabellen
- Ausnahmebehandlung
- Monitore

Hinsichtlich der Implementierung ist zu beachten, daß die meisten Befehle problemlos auf die darunterliegende Hardware abzubilden sind, wenn die Hardware bestimmte Anforderungen erfüllt, wie beispielsweise die IEEE 754–Norm für Gleitkommarechnung. Im anderen Fall muß die komplette Gleitkommaarithmetik in Software modelliert werden, was auch bei der Ausführung zu enormen Leistungseinbrüchen führen kann.

Ausnahme bei der leichten Implementierung bilden die komplexen Sprünge über Tabellen, die allerdings auch von den Übersetzerbauern gemieden werden, so daß sie fast nie auftreten. Aber auch das Monitorkonzept ist deutlich komplexer als die anderen Befehle, wenn das Betriebssystem nicht schon ein Monitorkonzept zur Verfügung stellt.

7.3.3 Native Methoden

Das vorangegangene Abschnitt über den Befehlssatz läßt vermuten, man könne eine virtuelle Java–Maschine in einer entsprechenden Architekturumgebung zügig entwickeln, und die Entwickler von SUN hätten ihr Ziel erreicht. Das stimmt auch, was den eigentlichen Interpretierer angeht.

Aber ohne die Objekte ist auch ein guter Interpretierer nutzlos. Bereits die simple Textausgabe ist mit einer Stackmaschine schon komplex genug, wenn man aber nicht auf die Hardwareebene bzw. auf das Betriebssystem zugreifen kann, ist so etwas schlichtweg unmöglich.

zen zu können. In der Sprachbeschreibung gibt es dazu das Schlüsselwort *native*, das besagt, diese Methode ist nicht in Java, sondern einer anderen Sprache realisiert, die auch direkt auf Betriebssystem und die Hardware zugreifen kann.

Diese Methoden sind dann zwar nicht in der Sprachdefinition von Java enthalten, gehören aber unmittelbar zum Java–System dazu, da sie in den Basisklassen angegeben sind. In dem von SUN mitgelieferten Entwicklerpaket taucht allein 140 mal das Schlüsselwort *native* auf.

Für den Entwickler eines Java–Interpretierers heißt das, er muß 140 Methoden implementieren, die die unterschiedlichsten Zwecke erfüllen. Der umfangreichste Komplex ist das AWT, das *Abstract Windowing Toolkit*. Da Fenstersysteme in jedem Betriebssystem anders realisiert sind, ist dies der schwierigste und umfangreichste Teil der Interpretiererbibliothek. Handelt es sich bei dem Betriebssystem, auf das der Interpretierer portiert werden soll, nicht um ein *multithreaded* Betriebssystem, gestaltet sich auch die Implementierung von Java–Threads als schwierig.

Besonderes Augenmerk bei der Implementierung *nativer* Methoden verdient noch die Klassenbibliothek, die sich mit dem Anschluß ans Netz befaßt. In den Packages von SUN, die nicht in der Java–Bibliothek enthalten sind, aber trotzdem mitgeliefert werden, finden sich fast komplette FTP–Clients, Mail–Agents oder Telnet–Anwendungen. Auch in der Implementierung dieser Klassen steckt je nach Betriebssystem mehr Arbeit als in anderen.

7.3.4 Das Laden der Klassen

Das erste, was ein Interpretierer zu tun hat, wenn er gestartet wird, ist, das zu interpretierende Programm zu laden. Die *Technik* realisiert der Classloader.

Er versucht die angegebenen Klassen, die sich nicht im Hauptspeicher befinden, von der Festplatte oder dem Netz zu laden. Dabei durchsucht er zuerst die Pfade, die mit der Umgebungsvariablen *CLASSPATH* gesetzt wurden. Deshalb ist es wichtig, den aktuellen Pfad immer mit in den *CLASSPATH* aufzunehmen.

Interessant ist hier ein Plus an Flexibilität, das dadurch erreicht wird, daß man nachträglich noch weitere Laderoutinen einbinden kann. Das Gerüst dafür bildet die abstrakte Klasse ClassLoader aus dem java.lang–Package.

Bei einer Implementierung dieser Klasse hat die Instanz dafür zu sorgen, daß zu einem Klassennamen ein Feld von Bytes geliefert wird, daß dem Inhalt der Dateien entspricht. Die Quelle kann wiederum eine Datei sein, die sich nicht im Pfad befindet, sondern z.B. aus Sicherheitsgründen in einem anderen Verzeichnis. Es könnte aber auch vorkommen, daß man Klassendateien verschlüsselt auf der Platte ablegt, und der Klassenlader sie erst wieder decodiert.

Schließlich ist es auch denkbar, daß die Klasse erst beim Aufruf erzeugt wird, also das Übersetzen in das endgültige Bytecodeformat erst beim Aufruf erfolgt, um die Klasse ganz speziell anzupassen. Wichtig ist dabei aber immer, daß auch die Klassen das Dateiformat einhalten, die nicht aus Dateien gelesen werden.

7.3.5 Dateiformat der Klassen

Auch beim Dateiaufbau der Klassen wurde auf Einfachheit und Kompaktheit geachtet. Es beginnt mit der bezeichnenden Sequenz im Hexadezimalcode *CAFEBABE*. Nach Versions– und Revisionsnummer der virtuellen Maschine, für die der Übersetzer diese Datei erzeugt hat, folgt eine Tabelle mit Konstanten, der sog. *ConstantPool*.

Werte, die im Programm auftauchen, sondern auch die Variablen– und Methodennamen, die das Programm benutzt. Diese sind nämlich — sofern sie nicht eindeutig als *private* gekennzeichnet sind — in der Klassendatei enthalten. Dort sind auch der Klassenname und der Name der Oberklasse enthalten.

Die Namen in Form textueller Bezeichner abzulegen, hat den Vorteil, daß deren Referenzen erst zur Laufzeit aufgelöst werden müssen. Hätten die Entwerfer beispielsweise mit Offsets in der Datei gearbeitet, könnten sich bei einem erneuten Übersetzen alle Offsets ändern. Die Namen bleiben jedoch gleich.

In der Datei ist auch vermerkt, welche *Interfaces* implementiert sind. Ebenso sind alle Methoden mit ihren Parametern und Zugriffsrechten verzeichnet. Dazu kommen noch Datei–Attribute wie die Bezeichnung der Quellcodedatei.

Wichtige Bestandteile dieses Formates sind die Codesegmente, die für jede Methode getrennt gespeichert werden. Zu ihnen zählen auch die lokalen Variablen und die Methoden, die beim Auftreten einer Exception aufgerufen werden.

Zur besseren Fehlerdiagnose werden außerdem immer die zu den einzelnen Befehlen gehörenden Zeilennummern im Quellcode gespeichert. So ist eine genaue Lokalisierung des Fehlers in der Quelldatei möglich.

7.4 Laufzeitsystem

Wir kennen jetzt alle Teilbereiche, um einen einfachen Java–Interpretierer zu implementieren: Vom Laden der Klassen über die Bytecode–Befehle bis zur Implementierung grafische Benutzerschnittstellen und verteilter Anwendungen. Jetzt geht es darum, den Interpretierer benutzerfreundlich und sicher werden zu lassen.

7.4.1 Codeverifikation

Neben den Fehlern, die durch schlecht programmierten Code oder unvorhergesehene Ereignisse wie Speichermangel auftreten können, muß der Interpretierer auch feindlichen Attacken von außen standhalten. Das heißt insbesondere, daß er die Klassendateien, die er geladen hat, genauestens auf deren Korrektheit überprüft. Dieser Aspekt wird im Kapitel *Java und Security* behandelt.

7.4.2 Exceptions

In Java dynamisch auftretende Fehler können *Exceptions* verursachen. Dabei handelt es sich um Ausnahmezustände, die dann erreicht werden, wenn eine native Methode einen Fehler in die Java–Umgebung meldet, der Interpretierer eine Unregelmäßigkeit feststellt oder der Programmierer einen Fehlerfall anzeigen will.

Zum ersten Fall zählen beispielsweise die *IOExceptions*. Bei der Ein– und Ausgabe kommt es immer wieder zu Fehlern, die vorher nicht festgestellt werden können, wie der Versuch, eine nicht vorhandenen Datei zu öffnen oder einen schreibgeschützten Datenträger zu beschreiben.

Eine typische Exception, die der Interpretierer instanziiert, ist die *NullPointerException*, die dann auftritt, wenn ein Objekt benutzt werden soll, das nicht existiert, beispielsweise ein Methodenaufruf eines nicht instanziierten Objektes. Der anachronistische Begriff NullPointerException

die *ArrayOutOfBoundsExceptions* auf, die anzeigen, daß ein Feld größer dem höchsten Index adressiert wurde, bzw. ein Feld kleiner dem niedrigsten Index.

Auch der Programmierer kann dynamisch Exceptions instanziieren, wenn er beispielsweise einen Fehlerfall entdeckt, der innerhalb seiner Methode nicht zu beheben ist. Er kann außerdem Exceptions abfangen, mit zusätzlichen Informationen versehen und dann dem aufrufenden Objekt mitteilen. Wichtig für den Programmierer des Interpretierers ist, daß die Bereiche des Quellcodes, in denen Exceptions auftreten können, in der Klassendatei gespeichert sind. Die Reihenfolge, in der die Blöcke der *catch*–Anweisungen abgefragt werden, ist dabei für den Interpretierer bindend.

7.4.3 Speicherverwaltung

Sämtliche Objekte werden in Java nicht auf dem Stack abgelegt, der für gezielte Zugriffe ungeeignet ist, sondern müssen auf der Halde gespeichert werden. Unter der Halde versteht man einen Speicherbereich im Hauptspeicher. Meist ist es ein großes, zusammenhängendes und vom Betriebssystem angefordertes Stück, das der Interpretierer eigenständig verwaltet. Dies ist auf jeden Fall schneller, als jede Speicherbedarfsänderung über das Betriebssystem abzuwickeln.

Da die Sprache Java und deren virtuelle Maschine keine Zeiger vorsieht, sondern alle Zugriffe über Objektnamen oder Objektreferenzen auflöst, ist es nicht nötig, die Inhalte immer in den gleichen Speicherstellen zu halten. Speicherblöcke können beliebig verschoben werden, wenn der entsprechende interne Zeiger geändert wird. Da sämtliche Objektzugriffe nie direkt, sondern immer über diese interne Zeigertabelle geregelt werden, ist nur noch zu beachten, daß an der Speicherstelle nicht gerade ein laufendes Programm steht.

Da es in Java außerdem keine expliziten Befehle zum Freigeben von Speicher gibt, muß auch diese Aufgabe vom Interpretierer erledigt werden. Er muß also von sich aus merken, wann Objekte nicht mehr benötigt werden. Dazu ist ein interner Nutzungszähler für jedes Objekt denkbar. Ist dieser wieder auf dem Anfangswert 0, kann das Objekt entfernt werden.

Noch eleganter ist allerdings das Prinzip der *Garbage Collection*. Sie wird erst bei Speicherknappheit ausgeführt und findet durch das Verfolgen von Referenzen heraus, welche Objekte noch gebraucht werden und welche nicht. Außerdem können alle noch aktiven Speicherblöcke ganz an den Anfang des großen Bereichs geschoben werden, um den internen Verschnitt zu eliminieren. Besteht dann immer noch Speicherbedarf, kann ein größeres Segment angefordert werden oder ein zusätzliches Segment in die interne Liste der Speicherblöcke aufgenommen werden.

Da diese Speicherbereinigung recht umfangreich ist, sollte sie erst dann ausgeführt werden, wenn sie wirklich nötig ist. Man kann sie jedoch auch in regelmäßigen Abständen durchführen, weil sie dann weniger komplex ist. Ebenso bietet es sich an, eine Speicherbereinigung zu starten, wenn kein Thread aktiv ist, oder der Interpretierer im Moment sonst nicht voll ausgelastet ist. Eine Implementierung als eigener Thread ist auch eine mögliche Lösung.

Auch hier haben die Entwickler einen Schritt weiter gedacht: Zusätzlich zum automatischen Start einer Speicherbereinigung, kann diese auch per Programm ausgelöst werden, wenn beispielsweise in einer zeitkritischen Methode viel Speicher benötigt wird, und eine automatisch ausgelöste Speicherbereinigung hinderlich wäre. *Destruktoren* wie sie in C++ exklusiv angegeben werden, gibt es in Java nicht. Jedes Objekt hat aber eine Methode *finalize*, die dann ausgeführt wird, wenn das Objekt aus dem Speicher entfernt wird, also nicht genau dann, wenn es nicht mehr benötigt wird, sondern unter Umständen sehr viel später.

Unser benutzerfreundlicher Java–Interpretierer ist uns nicht genug, wir wenden uns deshalb der Optimierung zu. Dabei soll unser Interpretierer schneller werden, ohne zum Übersetzer zu werden.

7.5.1 Bereits vorhandene Optimierungen

Die Möglichkeit, Optimierungen vorzunehmen, ist bei weitem nicht so groß wie etwa bei Übersetzern. Das liegt im wesentlichen daran, daß wir uns auf einer höheren Ebene bewegen. So kann man bei der Codegenerierung vor allem die Adreßberechnung bei Feldvariablen in Schleifen enorm vereinfachen, in dem man Induktionsvariablen benutzt. In Java geht dies nicht, da immer die Nummer des Feldelementes als Parameter eines Methodenaufrufs des Objektes Array übergeben wird.

Ein zweiter Grund für die geringere Anzahl von Optimierungen ist die Tatsache, daß der Bytecode als solcher schon ziemlich gut durchdacht ist und bei der Codegenerierung nur wenige Varianten zuläßt.

Der Befehlssatz des Bytecodes ist mit 200 Befehlen vergleichsweise kompakt. Da es außerdem keine unterschiedlichen Adressierungsarten wie bei der Maschinensprache gibt, ist die Möglichkeit, eine Befehlsfolge durch eine andere auszudrücken, verschwindend klein, wenn der Übersetzer nicht extrem umständlichen Code erzeugt.

Die Entwerfer des Bytecodes haben außerdem darauf geachtet, daß möglichst kompakter Code generiert werden kann. So erhält jeder Befehl, der auf eine lokoale Variable zugreift, nur noch ein einziges Byte mit dem Index der Variablen. Damit lassen sich folglich nur 2^8 Variable ansprechen. Durch einen vorangestellten *Wide*-Befehl läßt sich die Zahl der Variablen auf 2^{16} erhöhen. Wie bereits erwähnt, gibt es Spezialbefehle für die ersten 4 Variablen. Der Bytecode bietet so eine hohe Funktionalität bei kompakten Programmen.

Neben der besseren Ausnutzung des Stacks ist dies meines Wissens der einzige Ansatz zu einer lokalen Optimierung, wie sie sich bei Maschinensprache viel häufiger ergibt. Mit besserer Ausnutzung des Stacks ist gemeint, daß Variablen, die eigentlich vom Stack genommen werden aber kurz darauf wieder darauf gelegt werden sollen, gleich auf dem Stack bleiben können.

Ein weiterer bereits umgesetzter Vorteil besteht darin, daß Eigenschaften des Stacks schon zur Zeit des Übersetzens feststehen und der Interpretierer diese nur noch aus Sicherheitsgründen überprüfen muß. Dadurch entfällt ein unnötiges Prüfen des Stackgrenzen. In folgendem Vierzeiler müßte z.B. in jeder Zeile mindestens einmal der Stack überprüft werden:

```
iload_1
iload_2
iadd
istore_3
```

Bei jedem Laden muß sichergestellt werden, daß auch wirklich Integer geladen werden und es zu keinem Stack–Überlauf kommt. Vor der Addition muß geprüft werden, ob wirklich mindestens zwei Operanden auf dem Stack liegen, und ob sie auch den richtigen Typ haben. Existenz und Typ müssen ebenfalls beim Speichern geprüft werden.

Der Interpretierer spart also je zweimaliges Testen auf Überlauf und Unterlauf. Dazu kommen sechs Typüberprüfungen, die jedesmal einen heute zeitintensiven Speicherzugriff verursachen würden, da die Objekte in jedem Fall im Speicher liegen.

Das normale Vorgehen eines Java–Interpretierers ist:

- Variable 1 lesen und ohne Prüfung auf den Stack schreiben
- Stack erhöhen
- ebenso Variable 2
- Stack erhöhen
- beide Argumente vom Stack lesen, addieren und zurückschreiben
- Stack erniedrigen
- Ergebnis in Variable 3 ablegen
- Stack erniedrigen

Neben dem notwendigen Lesen und Schreiben der Variablen kommt es hier zu 6 Speicherzugriffen auf den Stack, der im Hauptspeicher untergebracht ist. Dazu kommt das weniger aufwendige In– und Dekrementieren des Stackpointers.

Mit etwas mehr Überblick kann der Interpretierer bei diesem Beispiel erkennen, daß der Stack in dieser Befehlsfolge gänzlich überflüssig ist. Er kann also — ohne Beteiligung des Stacks — beide Variablen in Register lesen, sie addieren und dann in der Zielvariablen speichern. Dies bringt immerhin einen Faktor größer zwei in der Ausführungszeit. Da solche Programmfragmente häufig vorkommen, ist ein derartiger Test im Mittel sinnvoll.

7.5.3 Nachträgliches Verbessern des Bytecodes

Mit etwas größerem Aufwand ist es möglich, dem eigentlichen Interpretierer einen Optimierer vorschalten. Dieser erledigt dann die Aufgaben, die schon der Bytecode–Übersetzer hätte erledigen können. Da man aber netzbedingt immer wieder Bytecode erhält, der von schlechteren Übersetzern produziert wurde, kann sich auch ein solcher Schritt auszahlen.

Er ist allerdings so aufwendig, daß er sinnvollerweise einmal vom Übersetzer gründlich erledigt werden sollte, anstatt bei jedem Laden der Klasse. Dafür ist die Bandbreite bei weitem größer als bei anderen Optimierungstechniken. Schleifeninvarianten können vorgezogen werden, eine weitergehende Konstantenfaltung kann durchgeführt werden, Code, der nicht durchlaufen wird, kann entfernt werden und viele weitere Gegebenheiten können ausgenutzt werden.

Eine Optimierung kann der Übersetzer jedoch nur in sehr geringem Umfang durchführen: das Ersetzen von Methoden durch *Inline–Code*, d.h. ein Methodenaufruf eines Objekts wird durch das entsprechende Codefragment ersetzt. Dadurch fällt der Aufruf weg, aber neuer Code entsteht, so daß sich diese Vorgehensweise meist bei kleinen Methoden lohnt. Der Übersetzer kann jedoch nur Methoden der eigenen Klasse so behandeln, da sich Methoden anderer Klassen durch separate Übersetzung ändern können. Erst der Interpretierer kann Inline–Methoden anderer Klassen verwenden, da sie sich zur Laufzeit nicht mehr ändern.

Weitere Verbesserungsmöglichkeiten bei konkreten Interpretierern ergeben sich aus deren inneren Architektur und konzentrieren sich — wie bei allen anderen Programmen auch — darauf, möglichst viele Variablen in Registern zu halten, also sinnvoll mit den vorhandenen Ressourcen umzugehen. Dabei ist es besonders wichtig, den Stack als zentrale Datenstruktur effizient zu verwalten.

Sozusagen der letzte Schliff an unserem Interpretierer ist, daß wir die Optimierung derart ausweiten, daß der Interpretierer direkt Maschinensprache erzeugt, die dann abgearbeitet wird. Er wird dann vom Interpretierer zum Übersetzer.

7.6.1 Was ist ein Just in Time–Übersetzer?

Wie der Name schon sagt, handelt es sich bei Just in Time–Übersetzern um Übersetzer, die ein Programm genau zur richtigen Zeit übersetzen, nämlich dann, wenn es ausgeführt werden soll. Dadurch ergibt sich zwangsläufig eine längere Anlaufzeit bei der Ausführung, da der Programmausführung eine Übersetzungsphase vorgeschaltet ist.

Selbstverständlich sind auch Übersetzer denkbar, die Java–Programme direkt in Maschinensprache umsetzen, wenn die Programme den eigenen Rechner nicht verlassen sollen. Ein Programm erst von Java in Bytecode zu übersetzen, dann einen Interpretierer zu starten, der wiederum einen Übersetzer von Bytecode in Maschinensprache startet, und diesen Maschinencode dann eventuell noch abschnittsweise zu interpretieren, scheint ein äußerst komplizierter Weg zu sein.

Bei einer genaueren Untersuchung wird man aber feststellen, daß es jedoch eine Lösung ist, die viele Vorteile vereint. Bei der ersten Übersetzung werden statische Programmfehler beseitigt, Bytecodeprogramme sind also prinzipiell ablauf/fähig. Dabei bleiben sie jedoch Maschinenunabhängig und kompakt, was den Transfer erleichtert. Erst wenn sie auf einer bestimmten Plattform angekommen sind, wird eine Anpassung an die Architektur der konkreten Maschine vorgenommen, die alle Vorteile der üblichen Interpretation hat, zusätzlich aber noch die Programme enorm verschnellert, so daß die Geschwindigkeit mit der anderer direkt compilierter Programme Schritt halten kann.

Zudem ist die Installation eines JIT–Übersetzers beispielsweise unter MS–Windows95 denkbar einfach: es wird einfach die Interpretiererbibliothek javai.dll gegen eine neue ausgetauscht, die statt des Interpretierers jetzt den Übersetzer enthält.

7.6.2 Unterschiedliche Ansätze

Der bisher besprochene Ansatz, Klassen gleich komplett zu übersetzen, ist sicher der einfachste, hat aber einen entscheidenden Nachteil: die Phase zwischen Laden der Klasse und Beginn der Ausführung kann je nach Klassenlänge und Leistung des Übersetzers schnell eine Länge von mehreren Sekunden erreichen, was dem Benutzer negativ auf/fällt.

Daher ist es denkbar, Klassen nicht komplett zu übersetzen, sondern immer nur Teile. Eine Abstufungsmöglichkeit ist, jeweils nur die benötigten Methoden zu übersetzen. Damit fällt ein Teil der Übersetzungen weg, weil bestimmte Methoden einer Klasse gar nicht genutzt werden. Desweiteren verteilt sich die Übersetzungszeit besser, so daß sie dem Benutzer nicht so auf/fällt.

Es kann aber vorkommen, daß immer noch zuviel übersetzt wird. In etlichen Fällen enthält die Klasse Fallunterscheidungen, von denen das Programm aber immer nur einen Zweig durchläuft. In diesem Fall bietet es sich an, noch kleinere Einheiten zu bearbeiten: beispielsweise jeden Bytecodebefehl genau dann, wenn er das erste Mal ausgeführt wird.

Diese Variante erscheint jedoch vor allem im Hinblick auf die Optimierungen nicht effizient genug. Als eine sinnvolle Einheit würde ich einen *Grundblock* betrachten, also eine Sequenz von Befehlen, die mit einem Sprungbefehl endet. Ein Grundblock wird außerdem immer vollständig

Sprungbefehl vorkommt.

Eine denkbare Strategie ist es, jeweils die benötigten Grundblöcke oder bei kleineren Methoden auch ganze Methoden zu übersetzen, wenn sie das erste Mal ausgeführt werden. Werden diese Codeteile dann ein zweites Mal benutzt, könnte es sein, daß der Grundblock Teil einer Schleife oder einer häufig aufgerufenen Methode ist, so daß sich eine weitere Optimierung lohnen könnte. Natürlich kann man auch gleich optimieren oder die Schranke für den Beginn der Optimierungen höher ansetzen, so daß erst beim vierten, fünften oder sechsten Durchlauf optimiert wird.

Da auf der Prozessorebene mehr Wissen über Optimierung bereitsteht als für die Optimierung von Stack–Maschinen, kommt man hier schneller zu einem Erfolg, als bei der Optimierung des Bytecodes.

7.6.3 Wann lohnt sich Just in Time?

Bei jedem Ansatz sollte sich die Übersetzungsdauer irgendwann auszahlen. Unabhängig davon stellt sich die Frage, welche Geschwindigkeitssteigerung im Optimalfall erreicht werden kann.

Die schnellste Möglichkeit, eine Geschwindigkeitssteigerung zu erreichen, ist diejenige, jeden Bytecode durch den im Interpretierer vorgesehenen Prozeduraufruf zu ersetzen. Dadurch fällt die jeweilige Interpretierphase weg, die allerdings nur einige Prozent der Gesamtlaufzeit ausmacht.

Der nächste Schritt ist, die einzelnen Prozeduren in Maschinensprache auszurollen, was allerdings einen unheimlich langen Code produzieren würde. Schnell wird er deshalb erst, wenn der JIT–Übersetzer anfängt, diesen Code zu optimieren, was vor allem bei einfachen Bytecode–Befehlen zu schnellen Programmen führt.

Auch hier verwenden wir wieder das Beispielprogramm mit der Addition zweier Ganzzahlen. Der wesentliche Abschnitt sah im Bytecode so aus:

```
1 iload_1
2 iload_2
3 iadd
4 istore_3
5 getstatic #2 <Field java.lang.System.out Ljava/io/PrintStream;>
6 iload_3
7 invokevirtual #3 <Method java.io.PrintStream.println(I)V>
8 return
```

In Maschinensprache übersetzt kann das Programm dann so aussehen:

```
MOVE   VAR_1,R0
MOVE   VAR_2,R2
ADD    R0,R2
MOVE   #VAR_3,R3
MOVE   R2,(R3)
MOVE   #2,R0
JSR    _GETSTATIC
MOVE   #3,R1
JSR    _INVOKEVIRTUAL
RETURN
```

befehlen möglich ist. Lediglich diejenigen Prozeduraufrufe werden länger, bei denen Parameter übergeben werden müssen.

Es ist davon auszugehen, daß der Interpretierer so realisiert ist, daß er einen Befehl holt, ihn decodiert und dann das entsprechende Unterprogramm aufruft. In diesem Fall besteht der *Interpretier–Overhead* aus etwa fünf bis zehn Befehlen. Dazu kommt noch Verwaltungsarbeit wie das Testen auf Ereignisse u.ä., was aber nicht unbedingt nach jedem Befehl erledigt werden muß.

Optimierungen wie Umgehung des Stacks und gute Registerausnutzung tragen ungefähr noch einmal einen Faktor zwei bis fünf bei.

Es ist jedoch ein Irrtum, mit 10 bis 50 mal schnelleren Programmen zu rechnen. Der wesentliche Anteil der Laufzeit liegt nicht in den interpretierten Befehlen der virtuellen Maschine, sondern in den Methoden, die wir als nativ implementiert voraussetzen können. Vor allem die Objektallozierung hat einen hohen Zeitverbrauch. Selbst wenn beispielsweise die Umwandlung des Ergebnisses in eine Zeichenkette ebenfalls durch interpretierten Javacode erfolgen kann, wird dieses Programm die meiste Zeit in der Ausgaberoutine verbringen und nicht in der Interpretierphase.

7.6.4 Benchmarks

Als Testumgebung für folgende Benchmarks diente ein MS–Windows95 Betriebssystem auf einem Rechner mit Intel Pentium Prozessor (60MHz).

Sie wurden mit Borlands Just in Time-Übersetzer *AppAccelerator 1.1* durchgeführt, der laut Berichten in den Newsgroups Symantecs *Café* von den Geschwindigkeitsgewinnen sehr ähnelt. Er geht so vor, daß er jeweils ganze Klassen und deren Oberklassen komplett übersetzt, wenn eine Klasse geladen wird. Außerdem ist er direkt in den Interpretierer eingebunden. Das hat zur Konsequenz, daß er Klassen, die nicht nach seinen Richtlinien programmiert sind, an den Interpretierer weiter geben kann. Dabei handelt es sich vor allem um Klassen, die unter früheren Übersetzerversionen erstellt wurden.

Benchmark	Parameter	Zeit in Sekunden Interpretierer	Zeit in Sekunden JIT	Beschleunigung
Matrixmultiplikation 3×3	double, 1 mal	0.95	1.01	0.94
	double, 1000000 mal	64.75	6.65	9.74
	float, 1000000 mal	63.22	6.64	9.52
	long, 1000000 mal	67.34	13.84	4.87
	int, 1000000 mal	62.67	7.53	8.53
Fibonacci-Zahlen	20 mal	1.15	0.83	1.39
	1000000 mal	5.61	0.99	5.67
EspressoGrinder (Übersetzer)	10 Klassen	20.81	9.12	2.28
prmake (Dateikonvertierung)	241 Dateien	1211.65	782.16	1.55
Fenster öffnen mit Bedienelementen	1 mal	2.64	3.03	0.87
	25 mal	24.39	22.24	1.10

bar machen, wenn Codeabschnitte mehrfach durchlaufen werden. Besonders deutlich wird dies bei kurzen Programmen. Hier kann sich die Zeit zum Übersetzen oft nicht amortisieren — im Gegensatz zu numerischen Programmen mit Schleifen, die sehr oft durchlaufen werden.

In der Praxis werden sehr unterschiedliche Werte erzielt. Geschwindigkeitsverluste treten höchstens bei kleinen Programmen auf, die keine Schleifen enthalten, weshalb auch ihre Ausführungszeit kurz ist. In solchen Fällen macht sich die Verzögerung kaum bemerkbar. Wenn es sich nicht gerade um reine Mathematikpakete handelt, die eine Beschleunigung bis zum Faktor 10 erfahren können, laufen Programme etwa doppelt bis dreimal so schnell ab wie mit dem Interpretierer.

Schnellere Werte erreichen im Moment nur die *Crosscompiler*, die Java-Quellcode in C uebersetzen. Die so erreichten C-Programme sind etwa sechs- bis achtmal so schnell wie interpretierte Java-Programme, müssen aber vor der Benutzung erneut übersetzt werden, was im Fall von JIT–Übersetzern automatisch erfolgt.

Seminarbeitrag von **Patrick Ohly**

Ganz unabhängig von Java stellt das Ausführen von fremden Programmen auf dem eigenen Rechner oder Account ein großes Problem dar, weil man meist die jeweiligen Autoren nicht kennt oder ihnen nicht vertrauen kann. Die Gefahren, die dabei entstehen, sind unter anderem folgende:

- Zerstörung oder Veränderung von Daten
- Ausspähung von Informationen
- Mißbrauch des Rechners, z.B. zum „Hacken" durch Aufbau von neuen Netzwerkverbindungen

Bei Java ist diese Gefährdung prinzipiell besonders groß. Die Programme werden in Form von Bytecode, also ohne die Möglichkeit, die Funktionsweise anhand des Quellcodes zu überprüfen, über Netzwerke heruntergeladen. Aufgrund der fehlenden oder kaum verbreiteten Authentifizierungsmöglichkeiten im Internet ist also die eigentliche Herkunft eines Programms sehr unsicher.

Daß Java mit seiner Einbindung in Web-Browser und der bereits eingebauten Unterstützung für Netzwerke überwiegend in ans Netz angeschloßenen Computern verwendet wird, erhöht die Gefahr der beiden letzten Punkte bzw. macht sie überhaupt erst möglich.

8.1 Gegenmaßnahmen

Um den eben genannten Gefahren zu begegnen, wird in Java ein Sicherheitssystem verwendet, das aus vier Schichten besteht [22]. Jede weitergehende Schicht baut auf der erfolgreichen Überprüfung eines Programms durch die vorherigen auf.

Sowohl der Java-Interpreter als auch der Übersetzer von Sun stehen im Quellcode zur Verfügung. Daher sind Überprüfungen auf Programmierfehler oder Sicherheitslücken durch Dritte möglich und werden bereits durchgeführt.

8.1.1 Schicht 1: Java-Sprache/Übersetzer

Die erste Schicht stellt die Sprache Java mit dem dazugehörigen Übersetzer selbst dar. Die Sicherheit eines Programmes wird durch Maßnahmen zur Vermeidung von Programmierfehlern und Verbot von nicht kontrollierbaren Sprachkonstrukten erhöht:

möglich, da es keine Zeigerarithmetik gibt.

- Die Sprachdefinition ist strikt: Typgrößen und Ausführungsreihenfolge sind genau spezifiziert. Daher sind Java-Programme nicht vom verwendeten Übersetzer abhängig.
- Garbage Collection hilft bei der Vermeidung zweier häufig auftretender Programmierfehler in C/C++ Programmen, dem Nichtfreigeben von nicht mehr benötigtem Speicher und dem Freigeben nicht mehr belegten Speichers.
- Starke Typisierung, die es bereits dem Übersetzer erlaubt, viele Programmierfehler zu erkennen:
  - Typumwandlung von Klasse auf abgeleitete Klasse nur mit Laufzeitcheck
  - Zugriff auf Methoden/Variablen nur bei den Objekten des richtigen Typs
  - strikte Abschirmung verschiedener Objekte, etwa durch private Elemente
  - Die Umwandlung von Integer in ein Objekt und umgekehrt ist verboten.

8.1.2 Schicht 2: Bytecode Überprüfung

Die in der Java-Sprachdefinition vorgegebenen Regeln schützen den Anwender nicht bei der Ausführung von Bytecode, da der ja auch von einem „feindlichen" Java-Übersetzer oder aus einer ganz anderen Sprache heraus erzeugt sein könnte.

Außerdem greift Bytecode ja auch auf Methoden anderer Module zu, deren Argumente sich geändert haben könnten oder die gar nicht mehr aufrufbar sein müssen.

Abhilfe schafft die zweite Schicht, in der beim auszuführenden Programm vor dem Starten des Programms das Dateiformat und der enthaltene Bytecode überprüft werden. In vier Phasen wird durch diese Vorausberechnung sichergestellt, daß zur Laufzeit folgende Punkte als gegeben vorausgesetzt werden können:

- Jede Methode benötigt genau die angegebene Menge Stack.
- Alle Registerzugriffe sind gültig.
- Parameter aller Bytecode-Instruktionen sind korrekt.
- Datenkonvertierungen sind erlaubt.

Phase 1: Formatüberprüfung
Im zu ladenden Bytecode werden folgende Punkte überprüft, daß

- am Anfang die Magic Number kommt,
- alle Attribute und die gesamte Klasse die richtige Länge haben, d.h. die Datei komplett übertragen wurde,
- der Konstanten-Pool keine unbekannten Informationen enthält.

Phase 2: Bytecode-unabhängige Checks
Jetzt werden inhaltliche Zusammenhänge überprüft, jedoch noch ohne den Bytecode zu betrachten. Einige der überprüften Bedingungen sind:

- Jede Klasse (außer `Object`) hat genau eine Vaterklasse.
- Element- und Methoden-Referenzen im Konstanten-Pool haben erlaubte Namen-, Klassen- und Typsignatur. Es erfolgt nur eine formale Prüfung, noch nicht die Überprüfung der Existenz der Elemente oder der Klasse selbst.

Phase 3: Bytecode-Verifier für jede Methode
Jede Methode wird als eigenständiger Codeteil mit einem einzigen Einstiegspunkt unabhängig vom Rest des Codes betrachtet. Beim Einstieg wird der Stack als leer angenommen. Außerdem ist der maximale Stackbedarf der Methode im Code angegeben. Daher sind eine untere und eine obere Grenze für den Stack innerhalb dieser Methode gegeben. Durch einen einfachen Theorem-Beweiser werden nun für jede Instruktion folgende Punkte sicherstellt, egal, auf welchem Weg die Instruktion erreicht wurde:

- Der aktuelle Stack hat immer dieselbe Größe hat und enthält dieselben Typen.
- Kein Register mit falschem Typ oder nichtinitialisiertem Wert wird gelesen.
- Methoden und Opcodes haben passende Argumente.
- Elemente werden nur mit Werten des richtigen Typs modifiziert.

Der Bytecode-Verifier
Die Implementierung von Phase 3 soll nun ausführlicher dargestellt werden [63]. Im ersten Schritt wird die zu untersuchende Methode in einzelne Instruktionen zerlegt, deren Anfänge gespeichert werden. Danach werden die Instruktionen der Reihe nach durchgegangen, in eine Struktur umgewandelt und dabei überprüft:

- Controll-Flow-Instruktionen müssen zum Beginn einer anderen Instruktion innerhalb der Methode springen.
- Registerzugriffe dürfen nur auf die Register erfolgen, die die Methode laut Deklaration für sich in Anspruch genommen hat.
- Referenzen auf den Konstanten-Pool müssen auf den richtigen Typ zugreifen.
- Der Code darf nicht in der Mitte einer Instruktion enden.
- Der Bereich, für den ein Exception-Handler zuständig ist, muß bei einer Instruktion anfangen und enden, wobei der Endpunkt hinter dem Startpunkt liegen muß. Der Beginn des Codes des Exception-Handlers muß eine Instruktion sein.

In der Struktur zu jeder Instruktion wird der Zustand des Stacks und der Register vor Ausführung der Instruktion festgehalten. Dazu gehören die Länge des Stacks und alle dort abgelegten Datentypen. Jedes Register enthält einen bestimmten Typ oder ist ungültig.

Diese Informationen werden durch eine Datenfluß-Analyse gesammelt. Diese wird initialisiert, indem nur die erste Instruktionen als „changed" markiert wird. Für die erste Instruktion werden der Stack auf leer und alle Register mit Argumenten der Methode auf den Typ des jeweiligen Arguments gesetzt. Andere Register sind ungültig.

Danach wird die in Abbildung 1 graphisch dargestellte Schleife gestartet:

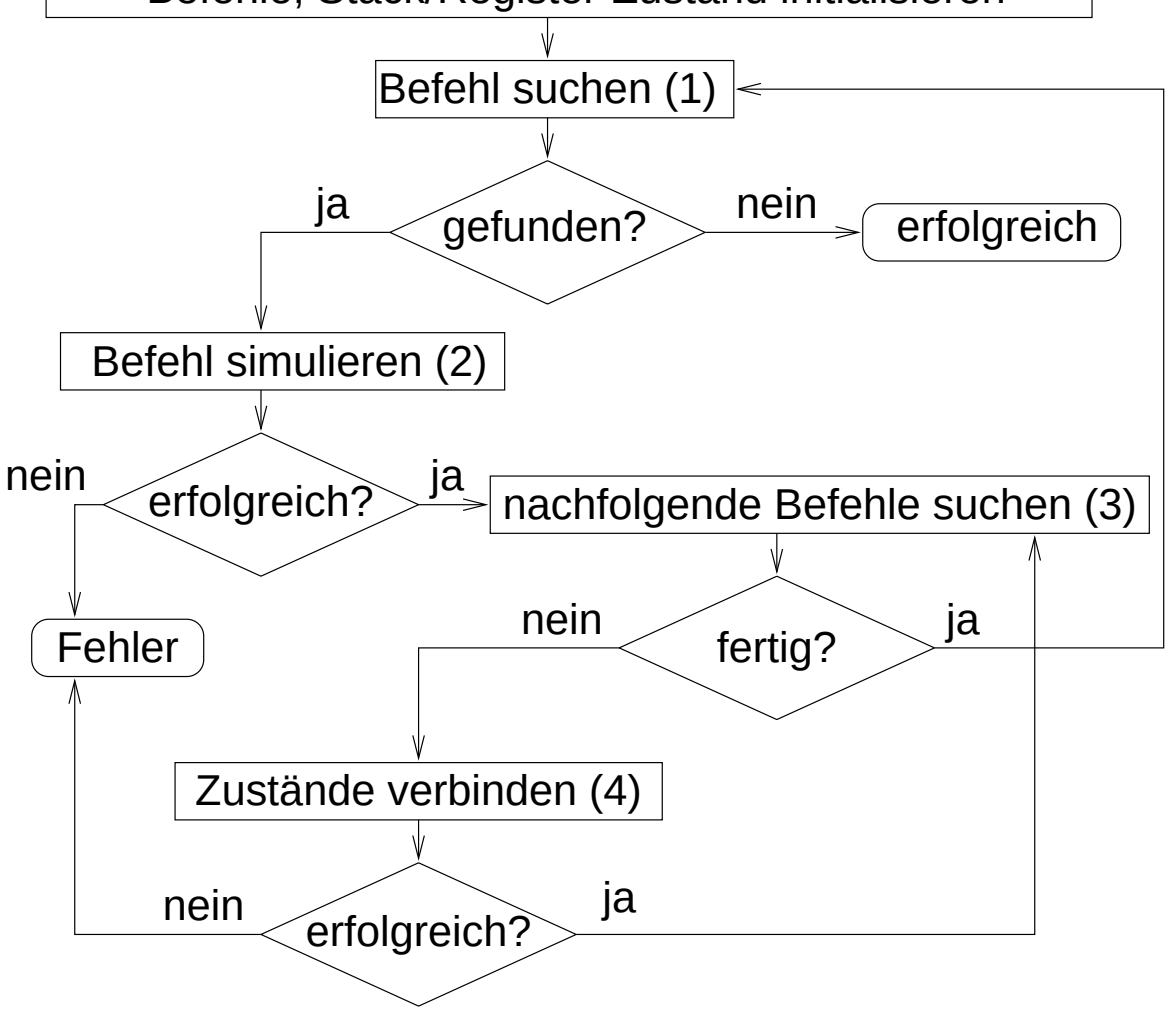


Abbildung 1: Datenflußanalyse

1. Suche eine Instruktion mit gesetztem „changed"-Bit. Wenn keine mehr gefunden wird, dann wurde die Methode erfolgreich überprüft, andernfalls wird ihr „changed"-Bit gelöscht.

2. Emuliere den Effekt der Instruktion auf den Stack und die Register:

   - Wenn Werte des Stacks verwendet werden, dann stelle sicher, daß genügend vorhanden sind und den richtigen Typ habe.
   - Wenn ein Register gelesen wird, dann muß es einen Wert des richtigen Typs enthalten.
   - Wenn Werte auf den Stack geschoben werden, dann überprüfe die obere Stackgrenze, füge die Typen der Werte zum Stackzustand hinzu und erhöhe seine Größe.
   - Wenn ein Register beschrieben wird, dann speichere seinen neuen Typ.

   Wenn irgendeine dieser Bedingungen nicht erfüllt ist, dann schlägt der Test fehl.

3. Suche alle nachfolgenden Instruktionen. Das können folgende sein:

   (a) die nächste Instruktion, falls die aktuelle kein unbedingtes `goto`, `return` oder ein `throw` ist; Fehler, wenn aus der Methode „herausgefallen" werden könnte.

(c) alle Exception-Handler für diese Instruktion.

4. Verbinde den geänderten Zustand mit dem aller nachfolgenden Instruktionen. Im Fall eines Exception-Handlers wird der Stackzustand zuvor auf einen Stack mit einem einzigen Objekt vom Typ der Exception gesetzt, wie es in der Deklaration des Exception-Handlers angegeben ist.

   - Wenn die Instruktion noch nicht betrachtet wurde, dann speichere den geänderten Zustand in ihrer Struktur ab und setze sie auf „changed".
   - Andernfalls verbinde die beiden Zustände wie unten beschrieben und setzte „changed", falls der Zustand der Instruktion dabei geändert wurde.

5. Gehe zu Schritt 1.

Beim Verbinden zweier Zuständer werden alle Typen auf dem Stack verglichen. Ist die Größe unterschiedlich oder haben zwei Einträge unterschiedliche Typen, die auch nicht auf eine gemeinsame Vaterklasse reduziert werden können, dann ist dies ein Fehler. Bei unterschiedlichen Typen mit gemeinsamer Vaterklasse wird der Typ in den der Vaterklasse abgeändert.

Bei unterschiedlichen Registerinhalten wird ein Register auf ungültig gesetzt, falls sie nicht auch auf einen gemeinsamen Vatertyp geändert werden können.

Traten bei der Überprüfung keine Fehler auf, so hat der Code Phase 3 erfolgreich passiert.

Einige Datentypen und Instruktionen verkomplizieren den Test. Auf ihre Sonderbehandlung wird nun eingegangen:

Long Integers und Doubles: Long Integers und Doubles belegen zwei hintereinander liegende Register. Werden sie in ein Register geschrieben, so wird für das darauffolgende markiert, daß es eine obere Hälfte enthält und somit nicht mehr alleine gelesen werden darf. Wird es überschrieben, so wird die untere Hälfte ungültig.

Auf dem Stack werden Longs und Doubles als einzelne Einträge mit der richtigen Größe behandelt.

Konstruktoren and neu erzeugte Objekte: Nach dem Anlegen des Speichers für ein neues Objekt wird der Konstruktor dafür aufgerufen. Das Objekt wird vom Verifier solange als „nicht initialisiert" betrachtet, bis der Konstruktor der aktuellen Klasse oder einer Vaterklasse aufgerufen wurde. Bis dahin darf das Objekt nicht gelesen werden.

Exception Handler: Der Verifier überprüft einige durch die Java-Sprachdefinition auferlegte Einschränkungen von Exception-Handlern *nicht*, da sie keine Gefahr für die Laufzeitumgebung darstellen.

Try/Finally: Der Code, der bei `finally` ausgeführt werden soll, kann mit verschiedenen Registerinhalten aufgerufen werden. Diese Register würden nach dem bisherigen Verfahren nach dem Verbinden der Zustände ungültig werden und im Code nach dem Rücksprung nicht mehr verwendet werden können. Abhilfe schafft eine weitere Sonderbehandlung der Art und Weise, wie Code aufgerufen wird.

Phase 4: Ergänzt Phase 3 zur Laufzeit um Nachladen
Aus Effizienzgründen wurden Tests bei Phase 3 vermieden, die ein Nachladen von Code erzwungen hätten. Beispiel: Die Überprüfung einer Zuweisung des Rückgabewerts einer Methode aus

Wenn aber der Variablentyp anders ist, dann muß bereits in 3 nachgeladen werden, um zu überprüfen, ob es sich um eine davon abgeleitete Klasse handelt.

Phase 4 findet erst zur Laufzeit statt. Wenn eine Instruktion zum ersten Mal ausgeführt wird, die eine andere Klasse referenziert, dann wird diese Klasse nachgeladen – falls nötig – und die Gültigkeit des Zugriffs durch die gerade ausgeführte Klasse überprüft. Beim Zugriff auf Elemente und Methoden werden außerdem deren Existenz, Signatur und die Gültigkeit des Zugriffs durch die gerade ausgeführte Methode geprüft.

Um diese Tests nicht bei wiederholter Ausführung derselben Instruktion nochmal durchführen zu müssen, werden geprüfte Opcodes durch `Quick`-Varianten ersetzt. Diese Opcodes dürfen nicht bereits in gerade geladenem Code stehen. Bei `Quick`-Opcodes ist somit sicher, daß obige Tests bereits einmal durchgeführt wurden und sie sofort ausgeführt werden können.

8.1.3 Schicht 3: `ClassLoader`

Nach der Code-Überprüfung wird der nachgeladene Code durch den `ClassLoader` in das Laufzeitsystem eingefügt. Dieses kann als eine Menge von Klassen angesehen werden, die in verschiedene Namensräume unterteilt sind. Es existieren ein Namensraum für eingebaute Klassen und für jeden Ursprung im Netz ein weiterer. Der nachgeladene Code wird dem zu seiner Quelle gehörenden Namensraum zugeordnet.

Bei der Suche nach Methoden oder Elementen wird erst der Namensraum der eingebauten Klassen durchsucht, danach erst der der eigenen Klasse. Code einer Netzquelle kann nicht auf den von fremden Netzen zugreifen. Genausowenig ist es ihm möglich, Code eingebauter Klassen zu überschreiben. Eingebaute Klassen können vom Netz nachgeladenen Code nicht unabsichtlich ausführen, sondern nur, wenn es ihnen explizit erlaubt wurde.

8.1.4 Schicht 4: Laufzeitsystem

Durch die bisherigen Schichten wurde sichergestellt, daß lokale Klassen wie z.B. diejenigen, die Dateizugriffe implementieren, vor Ergänzen oder Verändern durch importierten Code geschützt sind.

Die Verwendung der lokalen Klassen wurde allerdings noch nicht kontrolliert. Dies ist Aufgabe der vierten Schicht, dem Laufzeitsystem, das Bytecode ausführt. Eigenständige Java Programme, die nicht in einem Web-Browser ausgeführt werden, sind überhaupt nicht eingeschränkt und dürfen alle Dateien und Netzresourcen verwenden, auf die auch der Benutzer Zugriff hat. Verschiedene Browser implementieren aber auch unterschiedliche Schutzmechanismen für nicht vertrauenswürdige Applets aus dem Netz.

Für Dateizugriffe werden von Sun einstellbare Kontrollisten verwendet (`acl`, Access Control List), die den Zugriff meist restriktiv einschränken, je nachdem, ob das Applet lokal oder über das Netz geladen wurde. Greift ein Applet auf eine verbotene Art und Weise auf eine Datei zu, kann der Benutzer des Browsers über das dann erscheinende Dialogfenster den Zugriff doch noch erlauben.

Für Netzwerksicherheit werden vom `SecurityManager` Funktionen zur Verfügung gestellt, mit denen die Vertrauenswürdigkeit von Code festgelegt werden kann. Bisher wird nur der Ursprung im Netz verwendet, wobei zwischen Quellen innerhalb und außerhalb einer Firewall unterschieden

Autor und die Unversehrtheit des Codes feststellen könnten.

Basierend auf dem Wissen, woher Code kommt und wer ihn gestartet hat, und den Benutzereinstellungen für die jeweilige Quelle kontrolliert der Appletviewer die vom Code aufgebauten Verbindungen. Sie können auf den Ursprung des Code oder bestimmte Rechner beschränkt werden oder ganz verboten werden.

Die folgende Tabelle [21] zeigt, welche Rechte Applets bei Ausführung durch verschiedene Browser haben. Sie ist sicher nicht vollständig. Der Browser HotJava wird ausgeklammert, da er noch nicht offiziell veröffentlicht ist.

Legende	
NN	**N**etscape Navigator 2.x, Applet aus dem **N**etz
NL	**N**etscape Navigator 2.x, Applet aus **l**okalem Filesystem
AN	**A**ppletviewer, JDK 1.x, Applet aus dem **N**etz
AL	**A**ppletviewer, JDK 1.x, Applet aus **l**okalem Filesystem
EJ	**e**igenständige **J**ava Application

	streng ⟶ weniger streng				
	NN	**NL**	**AN**	**AL**	**EJ**
Datei in /home/me lesen, `acl.read=null`	nein	nein	nein	ja	ja
Datei in /home/me lesen, `acl.read=/home/me`	nein	nein	ja	ja	ja
Datei in /tmp schreiben, `acl.write=null`	nein	nein	nein	ja	ja
Datei in /tmp schreiben, `acl.write=/tmp`	nein	nein	ja	ja	ja
Dateiinfos lesen, `acl.read=null, acl.write=null`	nein	nein	nein	ja	ja
Dateiinfos lesen, `acl.read=/home/me, acl.write=/tmp`	nein	nein	ja	ja	ja
Datei löschen, mit `File.delete()`	nein	nein	nein	nein	ja
Datei löschen, mit `/usr/bin/rm`	nein	nein	nein	ja	ja
`user.name` Property lesen	nein	ja	nein	ja	ja
Verbindung zu Port bei Client	nein	ja	nein	ja	ja
Verbindung zu Port bei Drittem	nein	ja	nein	ja	ja
Bibliothek laden	nein	ja	nein	ja	ja
`exit(-1)`	nein	nein	nein	ja	ja
Fenster ohne Warnung öffnen	nein	ja	nein	ja	ja

8.2 Bekannte Schwachstellen, Programmierfehler

Die bisher dargestellte theoretische Grundlage relativiert sich deutlich, wenn es an die praktische Implementierung geht. Hier treten neben Programmierfehlern auch einige Designschwächen auf, die erhebliche Sicherheitslücken öffnen. In [10] werden sie ausführlich dargestellt. Im folgenden werden Programmierfehler in HotJava nicht erläutert, da vor der ersten Veröffentlichung des

anderen Applet-Umgebungen sind JDK 1.0.2 und Netscape Navigator 2.02.

8.2.1 „Denial of Service" Attacken

Nichts hält Java Applets davon ab, soviel Rechenzeit und Speicher wie möglich zu belegen, bis der Rechner nicht mehr verwendungsfähig ist. Außerdem können sie Browser-interne Locks belegen, die den Browser total belegen. Bei HotJava ist dies z.B. die unterste Zeile, was das Nachladen neuer Seiten verhindert, oder bei Netscape die für das Nachschlagen von Internet-Adressen zuständige Klasse `java.net.InetAddress`, was den Aufbau weiterer Netzverbindungen unterbindet.

Das Verhindern solcher Attacken ist schwierig, da zum einen die Attacke verzögert und damit schwer zugeordnet werden könnte, zum anderen solche Applets kaum von anderen mit einem tatsächlich sehr hohen Resourcenverbrauch unterschieden werden können.

8.2.2 Aufbau von Netzverbindungen

Die Regel, daß ein nicht vertrauenswürdiges Applet nur zu seiner eigenen Quelle eine Verbindung aufnehmen darf, wurde dadurch überprüft, daß der Nameserver zu allen IP-Adressen der Quelle und des Rechners befragt wird, zu dem das Applet eine Verbindung aufbauen wollte. Gab es mindestens eine Überschneidung, wurde die Verbindung erlaubt.

Eine durch diese Implementierung ermöglichte Attacke war ein manipulierter Nameserver, der für die Quelle `evil.hacker.org` auch eine IP-Adresse angab, die eigentlich einem anderen Rechner `victim.com` gehörte. Dies hatte es von `evil.hacker.org` geladenen Applets erlaubt, Verbindungen zu `victim.com` aufzubauen, auch wenn dieser Rechner hinter der Firewall lag, hinter der das Applet gestartet wurde.

Weiterhin konnte ein Applet den Nameserver von `evil.hacker.org` befragen, auch wenn es von einer ganz anderen Quelle geladen wurde. Die zu übertragenden Informationen konnten in dem fiktiven Namen, etwa `USERNAME.Joe.Victim.evil.hacker.org`, verschlüsselt werden. Diesen Namen konnte der Nameserver des Angreifers auswerten und so Informationen erhalten, die eigentlich gar nicht an ihn hätten übertragen werden dürfen.

Die bisherigen Schwächen sind inzwischen behoben. Eine immer noch existierende Art, Informationen an beliebige Internetrechner zu schicken, hat ein Applet, wenn auf seiner Quelle ein Mailserver mit SMTP läuft. Dann kann es beliebige Mails verschicken.

8.2.3 Manipulation des Laufzeit-Systems

Bei Netscape sind zwar Applets untereinander getrennt, aber ihre Threads nicht. Da die Threadnamen auch die Appletnamen enthalten, kann ein Applet alle anderen laufenden Applets auflisten. Außerden kann ein Applet die anderen Threads verlangsamen oder sogar stoppen, weil der `SecurityManager` nur die System-Threads schützt.

Der Bytecode-Verifier war fehlerhaft, sodaß der `ClassLoader` durch ein Applet ersetzt werden konnte. Da dieser für das Durchsuchen der Namensräume zuständig ist, kann ein manipulierter `ClassLoader` Applets den Zugriff auf Variablen und Methoden aus anderen Namensräumen erlauben. Außerdem können eigentlich illegale Typumwandlungen erlaubt werden, da diese Überprüfung auch im `ClassLoader` implementiert ist. Dadurch könnte ein Methode etwa private

identische umgewandelt wird, in der die Elemente ungeschützt sind. Dies würde die Sicherheit des Laufzeitsystems völlig untergraben.

8.2.4 Konzeptionelle Schwächen

In Java gibt es keine formale Beschreibung der Semantik oder des Typsystems. Daher kann auch die Korrektheit des Bytecode-Verifiers nicht bewiesen werden. Der Bytecode-Verifier wird durch ungewöhnliche Konzepte wie Exceptions und die Objekt-Initialisierung verkompliziert. Bei Variablen wird nur der Zugriff kontrolliert, sichtbar sind sie auch nach außen. All dies sind Dinge, die durch ein anderes Design hätten behoben werden können.

Es fehlt ein einheitliches Sicherheitskonzept, daß vom System selbst durchgesetzt wird. Statt dessen werden unterschiedlichste Regeln für verschiedene Applets angewendet. Die Anwendung der Regeln muß der Programmierer von sicherheitsrelevanten Methoden selbst überprüfen, was dazu führt, daß verschiedene Teile des Systems zusammenarbeiten müssen, um Sicherheit zu garantieren.

Weiterhin gibt es keine Möglichkeiten, mitzuprotokollieren, welches Applet was wann gemacht hat. Das Java-System sieht hier nichts vor.

8.3 Schlußfolgerung

Java hat sich als interessante und vielversprechende Sprache für die Programmierung von Anwendungen auf Web-Seiten erwiesen. Allerdings sind eine ganze Reihe von Sicherheitslücken aufgetaucht. Auch wenn die meisten davon behoben werden konnten, so lassen sie doch befürchten, daß auch weiterhin Programmierfehler entdeckt werden.

Ein Beweis der Korrektheit und der Sicherheit einer Java-Implementierung scheitert an dem fehlenden formal definierten Sicherheitssystem. Ohne grundlegende Design-Änderungen an der Sprache, dem Bytecode und dem Laufzeitsystem werden Aussagen zur Sicherheit auch weiterhin zu relativieren sein.

Seminarbeitrag von **Matthias Zenger**

9.1 Einleitung

Java ist eine einfache, universell einsetzbare, klassenbasierte, objektorientierte Programmiersprache [20]. Sie wurde vor allem für die Entwicklung von Anwendungen, die auf heterogenen, über das ganze Netzwerk verteilten Umgebungen basieren, entwickelt. Um Java-Anwendungen auf verschiedenen Plattformen verwenden zu können, sind Architekturunabhängigkeit und Portabilität wichtige Grundvoraussetzungen. Deswegen erzeugt ein Java-Compiler, nicht wie üblich, aus dem Quellcode Maschinencode für einen bestimmten Rechner, sondern es wird ein maschinenunabhängiger *Bytecode* für eine abstrakte Stackmaschine, die *Java Virtual Machine*, generiert [23]. Dieser Java Bytecode kann dann auf den verschiedensten Hardware- und Betriebssystemplattformen entweder durch eine Implementierung der *Java Virtual Machine* interpretiert oder durch einen Just-In-Time Compiler weiter in den hardwarespezifischen Maschinencode übersetzt werden.

Der erste Java Compiler, der unabhängig von Sun's Übersetzer *javac* geschrieben wurde, ist EspressoGrinder. EspressoGrinder wurde am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe, unter der Leitung von Martin Odersky, der zugleich auch Hauptautor des Compilers ist, entwickelt. EspressoGrinder ist selbst in Java geschrieben. Er wurde ursprünglich als Teil des *Pizza* Projekts [41] erstellt, um eine Grundlage für die Erweiterung des Typsystems von Java um Konzepte funktionaler Programmiersprachen zu haben. Java stellt hierfür eine gute Basis dar, da Java entsprechende Sicherheitsaspekte garantiert und damit zusammenhängende Konzepte, wie das eines Garbage Collectors, auf der Ebene der Java Virtual Machine bereits vorhanden sind. Erweiterungen am Compiler wurden, im Rahmen dieses Projekts, in drei Bereichen vorgenommen: First-Class Functions, parametrische Polymorphie und algebraische Datentypen.

9.2 Übersicht über die Architektur von *EspressoGrinder*

Der interne Aufbau des Compilers entspricht prinzipiell der üblichen Architektur eines Übersetzers. Die folgende vereinfachte Beschreibung eines Compilerlaufs soll eine Übersicht über die Abfolge der einzelnen Übersetzungsphasen, sowie die dabei beteiligten Komponenten geben.

EspressoGrinder übersetzt die beim Aufruf angegebenen Quelldateien geschlossen. Das bedeutet, daß nach einer Initialisierung zunächst nacheinander alle zu übersetzenden Quelldateien eingelesen und dabei die syntaktisch richtige Verwendung der Sprachkonstrukte überprüft werden. Der Parser generiert bei diesem Vorgang für jede Datei einen abstrakten Syntaxbaum (engl.

abstrakten Syntaxbäumen als Zwischendarstellung des Quelltextes erlaubt die Entkopplung der syntaktischen Analyse von den restlichen Übersetzungsphasen.
Erst nachdem alle Quelldateien zerteilt wurden, wird basierend auf den Syntaxbäumen eine semantische Analyse für jede Übersetzungseinheit durchgeführt. Da hierbei hauptsächlich zur Durchführung von Typprüfungen Typinformationen gesammelt und Kontextbezüge hergestellt werden, ist es in dieser Phase erstmals notwendig auch auf bereits bestehende übersetzte Klassen zuzugreifen. Eine Klassendateiverwaltung ermöglicht den Zugriff auf Klassendateien und verwaltet alle bereits geladenen bzw. zu generierenden Klassen.
Mit der semantischen Analyse ist die Analysephase des Compilerlaufs abgeschlossen. Danach wird für alle in den abstrakten Syntaxbäumen enthaltenen Methoden, Konstruktoren und Initialisierungsroutinen Code erzeugt. Am Ende werden alle Klassen jeweils in eine eigenen Klassendatei geschrieben.

Natürlich sind neben den bisher erwähnten Komponenten noch weitere Einheiten wie z.B. zur lexikalischen Analyse, Fehlerbehandlung und Symboltabellenverwaltung notwendig. Diesbezügliche Details folgen in späteren Abschnitten. Eine Übersicht, die nur die wesentliche funktionale Struktur im Aufbau von EspressoGrinder aufzeigt, gibt Abbildung 2.

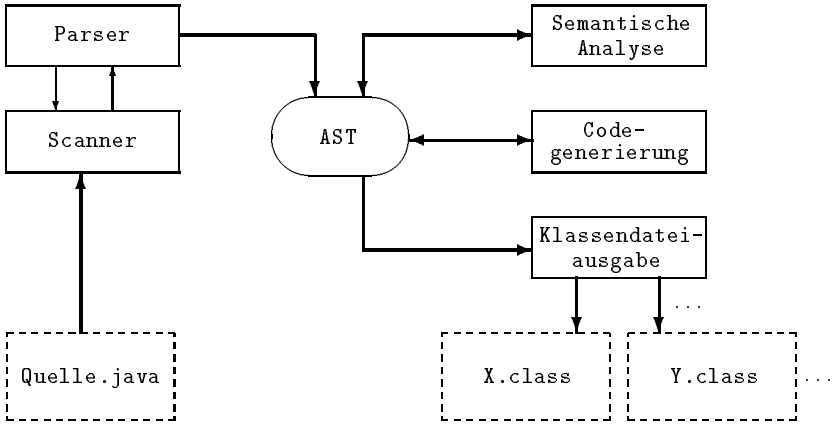


Abbildung 2: Funktionale Struktur von EspressoGrinder

9.3 Lexikalische und syntaktische Analyse

9.3.1 Fehlerbehandlung

Da ein Programm auf vielen verschiedenen Ebenen Fehler enthalten kann, können auch während allen Übersetzungsphasen Fehler vom Compiler entdeckt werden. Um eine einheitliche Fehlerbehandlung und -ausgabe in allen Phasen des Übersetzungsvorgangs zu ermöglichen, wird zur Fehlerausgabe die Klasse `Report` bereitgestellt.

Vor dem jeweiligen Start einer Übersetzungsphase wird mittels der Methode `open` das aktuell zu analysierende Programm der Fehlerausgabeklasse mitgeteilt. Später auftretende Fehler bzw. Warnungen können dann einfach durch Aufrufe von `error` oder `warning` ausgegeben werden. Dabei wird neben der Fehlermeldung noch jeweils die Zeile ausgegeben, innerhalb der der Fehler lokalisiert wurde. Außerdem wird ein Fehlermeldungszähler verwaltet, um am Ende einer Übersetzungsphase entscheiden zu können, ob eine Fortsetzung des Übersetzungsvorgangs noch sinnvoll ist.

`String filename;`	Aktueller Quelltextname
`int nerrors;`	Anzahl gemeldeter Fehler
`int nwarnings;`	Anzahl gemeldeter Warnungen
`void open(String filename);`	Bearbeitung neuer Übersetzungseinheit
`void close();`	Bearbeitung beenden
`void error(int pos, String msg);`	Fehler melden
`void warning(int pos, String msg);`	Warnung ausgeben

Abbildung 3: Schnittstelle der `Report`-Klasse

Wie man aus der Schnittstellenbeschreibung in Abbildung 3 entnehmen kann, werden Positionen im Quelltext innerhalb von EspressoGrinder einheitlich in einem einzigen Integer-Wert codiert. Nur in der `Report`-Klasse ist eine Decodierung dieser Positionsangabe in Zeilen- und Spaltennummer notwendig.

Da jede Fehlerausgabe auf eine Stelle im Quelltext verweist, mußte beim Entwurf des Compilers sichergestellt werden, daß zu jeder Zeit für einen möglicherweise auftretenden Fehler auch die dazugehörige Position im Quelltext verfügbar ist. Deswegen werden praktisch bei allen Datenstrukturen Positionsangaben mitgeführt, die auf die Stelle im Quelltext verweisen, auf die sich das Datum bezieht.

9.3.2 Lexikalische Analyse

Bei der lexikalischen Analyse wird der Quelltext vom Anfang bis zum Ende gelesen und dabei in Symbole (engl. *token*) aufgeteilt. Ein Symbol stellt eine Folge von Zeichen dar, die zusammen eine bestimmte Bedeutung haben. Ist hier von Zeichen die Rede, so sind Zeichen des Unicode Zeichensatzes gemeint [58]. Java Programme, die nur mit ASCII-Zeichen abgefaßt sind, können Unicode Escape-Sequenzen enthalten, die bereits während der lexikalischen Analyse aufgelöst werden müssen.
Zur Durchführung der lexikalischen Analyse in EspressoGrinder wird die Klasse `Scanner` verwendet. Eine stark vereinfachte Übersicht über deren Schnittstelle gibt Abbildung 4.

Scanner	Bedeutung
`int sym;`	Symbol-Kennung
`int pos;`	Position im Quelltext
`Name name;`	Lexem als Name
`long intVal;`	Integer-Wert einer Zahl
`double floatVal;`	Fließkomma-Wert einer Zahl
`Scanner(FileInputStream in);`	Konstruktor
`void nextsym();`	Nächstes Symbol einlesen

Abbildung 4: Schnittstelle der `Scanner`-Klasse

Für jeden zu übersetzenden Quelltext wird eine Instanz der `Scanner`-Klasse angelegt. Dabei wird als erstes der gesamte Quelltext in einen Puffer gelesen, um den Zugriff auf *lookahead*-Zeichen zu vereinfachen. Beim Zerteilen des Quelltextes wird dann vom Parser sukzessive die Methode `nextsym` aufgerufen, die das nächste Symbol einliest. Dabei wird das Attribut `sym` mit einer Kennung für das gelesene Symbol und `pos` mit der Position im Quelltext versehen. Es gibt, grob gegliedert, fünf Klassen von Symbolen: Literale, Bezeichner, Operatorsymbole, Trennzeichen und Schlüsselwörter. Da jedes Java-Schlüsselwort ein eigenes Symbol darstellt, wird zur Ermittlung der Symbolkennung die Tabelle `Scanner.key` verwendet, die zu jedem Schlüsselwortnamen die entsprechende Symbolkennung enthält. Für Ausgabezwecke wird zusätzlich noch eine

korrespondierenden Namen liefert. Handelt es sich bei einem gelesenen Symbol um ein Literal für eine Zahl, so bekommt zusätzlich, je nachdem ob es sich um eine Integer- oder Fließkommazahl handelt, entweder das Attribut `intVal` oder `floatVal` den entsprechenden Wert zugewiesen. Wird ein Symbol durch mehrere verschiedene *Lexeme* repräsentiert, so enthält das Attribut `name` einen Verweis auf einen Eintrag in der Namensverwaltung von EspressoGrinder.

Die interne Namensverwaltung wird durch die Klasse `Name` abgewickelt. Zeichenfolgen werden dort aufeinanderfolgend in einem reservierten Speicherbereich abgelegt. Eine Hashtabelle enthält Referenzen auf alle abgelegten Namen. Da beim Eintragen eines neuen Namens nur dann ein neuer Eintrag in der Hashtabelle vorgenommen wird, wenn dieser Name bisher noch nicht bekannt ist, ist ein Name eindeutig durch seine Position im reservierten Speicherbereich bestimmt. Die Gleichheit zweier Namen läßt sich so durch einen Vergleich ihrer beiden Speicheradressen feststellen. Damit wird der Umgang mit Symboltabellen, wie man im folgenden sehen wird, wesentlich beschleunigt und vereinfacht.

Die Fehlerbehandlung während der lexikalischen Analyse fällt im Prinzip recht einfach aus. Tritt ein Fehler bei der Bestimmung des nächsten Symbols auf, wie etwa ein falsch geschriebenes Schlüsselwort, ein unbekannter Operator oder auch eine fehlerhafte numerische Angabe, so wird eine Fehlerbeschreibung ausgegeben und `nextsym` liefert dem Parser einfach den Wert `NoSy`, womit angezeigt wird, daß das aktuell zu lesende Symbol fehlerhaft ist.

9.3.3 Syntaktische Analyse

Ebenso wie für den Scanner wird für jeden zu übersetzenden Quelltext ein eigener Zerteiler instanziiert, der mit dem dazugehörigen Scanner initialisiert wird. Letzterer liefert einen Strom von Symbolen an den Zerteiler, der diesen auf Syntaxfehler hin untersucht. Zerteiler werden durch die Klasse `Parser` realisiert.

Bei dem Zerteiler handelt es sich um einen üblichen *recursive decent*-Parser, der aufbauend auf einer kontextfreien LL(1)-Grammatik entworfen wurde. Dieser prädiktive Parser liefert als Ausgabe seiner syntaktischen Analyse einen *abstrakten Syntaxbaum*, der jeweils eine gesamte Übersetzungseinheit darstellt.

AST	Bedeutung
`int pos;` `int tag;`	Zugehörige Stelle im Quelltext Knoten-Art
`Typ typ;`	Typ, der mit Knoten assoziiert ist

`AST(int p, int t);`
`void enter(Env e);`
`void enterFields(Env e);`
`Typ attr(Env e, int kind, Typ pt);`
`Item gen();`
`void output();`
	Konstruktor
	(siehe nachfolgenden Abschnitt)
	(siehe nachfolgenden Abschnitt)
	Attributierung des Knotens
	Code für Knoten erzeugen
	In Klassendatei ausgeben

Abbildung 5: Basisklasse eines Syntaxbaumknotens

Die interne Datenstruktur, mit deren Hilfe abstrakte Syntaxbäume aufgebaut werden, besteht aus Klassen, die von der Basisklasse `AST` abgeleitet sind. Eine Klassenbeschreibung ist in Abbildung 5 gegeben. Bei der Konstruktion des abstrakten Syntaxbaums in der syntaktischen Analyse können in den Knoten bereits alle Attribute eingetragen werden, die in direktem Zusammenhang zum Quelltext stehen. Für die Basisklasse `AST` sind dies nur die Felder `pos` und `tag`. Das Feld `pos`

Kennung, die die Art des AST-Knotens (siehe Tabellen in Abbildung 7) näher charakterisiert.

Innerhalb der syntaktischen Analyse werden zum Aufbau des abstrakten Syntaxbaums nur die Konstruktoren der von `AST` abgeleiteten Klassen benötigt. Die beiden Methoden `enter` und `enterFields` werden zu Beginn der semantischen Analyse zur Initialisierung und Eintragung von Objekten in die Symboltabelle verwendet, die durch den betrachteten Knoten definiert werden. Mit der Methode `attr` wird schließlich der abstrakte Syntaxbaum bewertet, indem die übrigen Attribute ausgewertet werden. Mittels `gen` wird zur Codegenerierungsphase Code für den jeweiligen Knoten des Syntaxbaums generiert, der dann mit `output` in eine Klassendatei geschrieben wird.

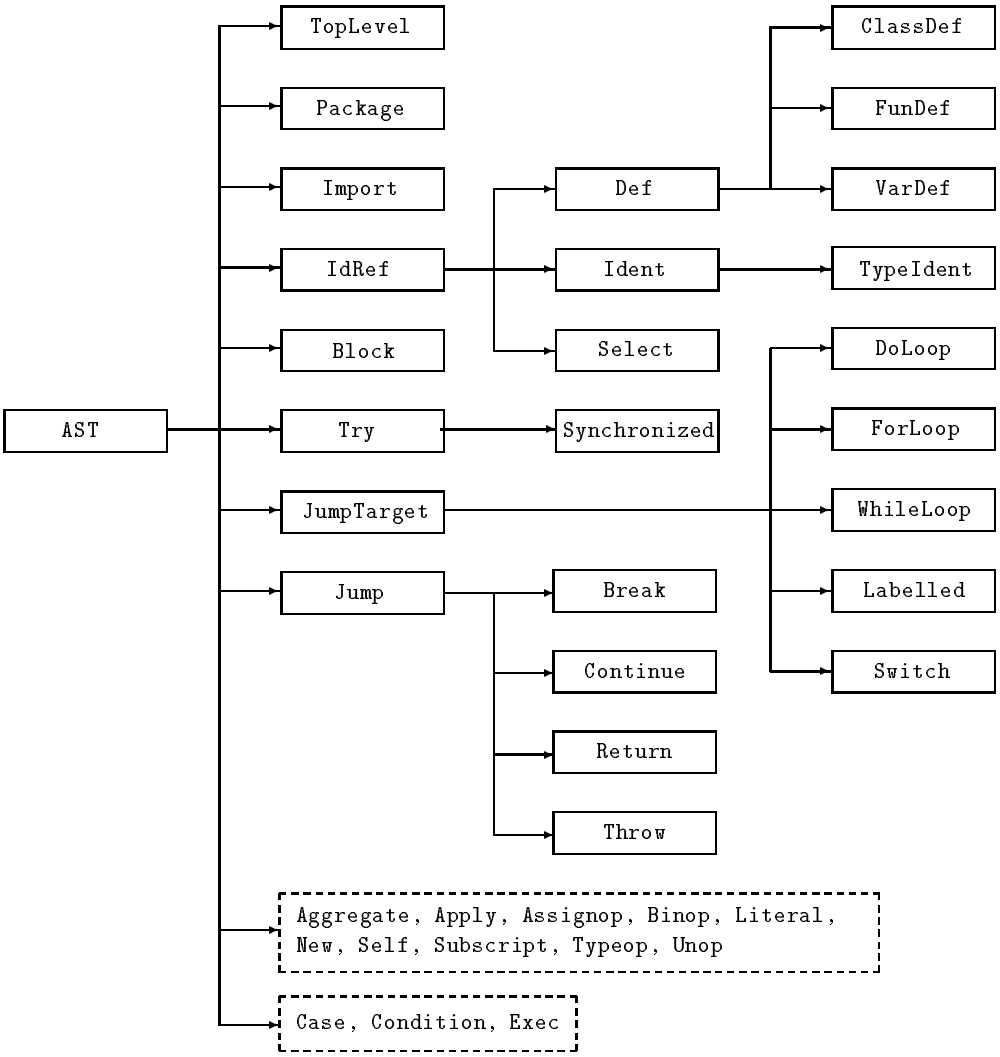


Abbildung 6: AST Klassenhierarchie

Die Tabellen in Abbildung 7 geben eine Übersicht über die abstrakte Syntax. Die obere Tabelle beschreibt die Struktur der abstrakten Syntaxbäume, in der unteren Tabelle werden alle, in einem abstrakten Syntaxbaum vorkommenden Knotentypen beschrieben. Neben den *Tags* sind dort alle die Attribute angegeben, die der Parser selbst beim Zerteilungsvorgang einträgt. Im abstrakten Syntaxbaum kann man sich diese Attribute bildlich als Verweise auf die Söhne vorstellen. Eine komplette Klassenhierarchie aller `AST`-Unterklassen ist in Abbildung 6 zu finden. Aus Gründen

```
compilation_unit  = [Package] {Import} {ClassDef}
statement         = FunDef(FUNDEF)
                  | VarDef
                  | Exec
                  | Block
                  | ForLoop
                  | WhileLoop
                  | DoLoop
                  | Conditional(statement)
                  | Return
                  | Throw
                  | Labelled
                  | Try
                  | Synchronized
                  | Switch
                  | Break
                  | Continue
expression        = Ident
                  | Select(expression)
                  | Apply(expression)
                  | Subscript(expression)
                  | New
                  | Aggregate
                  | Conditional(expression)
                  | Literal
                  | Self
                  | Binop
                  | Assignop
                  | Typeop
                  | Unop
type              = qualid
                  | Subscript(type)
constr            = Apply(qualid)
                  | Subscript(constr)
                  | qualid
qualid            = Ident
                  | Select(qualid)
```

Abstrakte Syntaxbeschreibung			**Anmerkung**
`Aggregate`	`= AGGREGATE`	`elems:{expression}`	Implizite Initialisierung eines Arrays
`Apply(A)`	`= APPLY`	`fun:A args:{expression}`	Methodenanwendung: `fun(args)`
`Assignop`	`= anASSIGNOP`	`left:expression right:expression`	Zuweisung: `left anASSIGNOP right`
`Binop`	`= aBINOP`	`left:expression right:expression`	Operatoranwendung: `left OP right`
`Block`	`= BLOCK`	`mods:int stats:statement`	Block: {...}
`Break`	`= BREAK`	`label:[Name]`	Break-Anweisung
`Case`	`= CASE`	`pat:[expression] stats:statement`	Case-Anweisung
`ClassDef`	`= CLASSDEF`	`name:Name mods:int extends:[qualid] implements:qualid body:Block`	Vollständige Definition einer Klasse
`Conditional(A)`	`= COND(A)`	`cond:expression thenpart:A elsepart:A`	if (`cond`) `thenpart` else `elsepart`
`Continue`	`= CONTINUE`	`label:[Name]`	Continue-Anweisung
`DoLoop`	`= DOLOOP`	`cond:expression body:statement`	Do-While-Schleife
`Exec`	`= EXEC`	`expr:expression`	Ausdruck als Anweisung ausführen
`ForLoop`	`= FORLOOP`	`e1:[expression] e2:[expression] e3:[expression] body:statement`	for(`e1; e2; e3`) `body`
`FunDef(A)`	`= A`	`name:[Name] mods:int dcltyp:[type] param:VarDef thrown:type [body:Block]`	Definition einer Funktion
`Ident`	`= IDENT`	`name:Name`	Bezeichner
`Import`	`= IMPORT`	`pid:qualid`	Import-Anweisung
`Labelled`	`= LABELLED`	`label:Name stat:statement`	Anweisung mit vorangestelltem Label: `label: stat`
`Literal`	`= aLITERAL`	`val:Object`	Literal für (konstantes) Objekt `val`
`New`	`= NEW`	`constr`	New-Ausdruck
`Package`	`= PACKAGE`	`pid:qualid`	Package-Deklaration
`Return`	`= RETURN`	`expr:[expression]`	Return-Anweisung
`Select(A)`	`= SELECT`	`struc:A name:Name`	Qualifizierter Zugriff: `struc.name`
`Self`	`= aSELF`		Repräsentiert `this` oder `super`
`Subscript(A)`	`= SUBSCRIPT`	`struc:A index:[expression]`	Indizierter Zugriff auf Array: `struc[index]` oder Typkonstruktor
`Switch`	`= SWITCH`	`sel:expression cases:Case`	Switch-Anweisung
`Synchronized`	`= SYNCHRON`	`lock:expression stat:statement`	Synchronized-Anweisung
`Throw`	`= THROW`	`expr:expression`	Throw-Anweisung
`Try`	`= TRY`	`body:Block catchers:FunDef(CATCH) finalizer:[Block]`	Try-Catch-Anweisung
`Typeop`	`= aTYPEOP`	`expr:expression dcltyp:type`	Typ-Casting
`Unop`	`= aUNOP`	`operand:expression`	Anwendung eines unären Operators
`VarDef`	`= VARDEF`	`name:Name mods:int dcltyp:type init:[expression]`	Variablen-Deklaration
`WhileLoop`	`= WHILELOOP`	`cond:expression body:statement`	While-Schleife

Abbildung 7: Abstrakte Syntax

aufgeführten Klassen sind einzelne Unterklassen von `AST`.

Als Ergebnis der syntaktischen Analyse einer Übersetzungseinheit liefert der Parser jeweils einen `TopLevel`-Knoten. In diesem Knoten werden Referenzen auf alle Deklarationen, d.h. `package`-, `import`-Anweisungen und Klassendefinitionen im Quelltextes gesammelt.

Nach dem Ende der Syntaxanalysephase liegt für alle Übersetzungseinheiten jeweils eine Wurzel eines eigenen abstrakten Syntaxbaums, repräsentiert durch einen `TopLevel`-Knoten vor. Von diesen Wurzeln aus werden dann alle nachfolgenden Übersetzungsphasen gestartet.

Die Fehlerbehandlung während des Zerteilens basiert auf einer *panischen Recovery*-Strategie. Wenn der Parser einen Fehler entdeckt – das tut er z.B. automatisch, wenn er vom Scanner den Wert `NoSy` geliefert bekommt – wird der erkannte Fehler, sofern dies noch nicht geschehen ist, ausgegeben und es werden die nachfolgenden Eingabesymbole solange überlesen, bis ein Symbol gefunden wird, das ein erfolgreiches Fortsetzen verspricht. Zu diesen „synchronisierenden Symbolen“ gehören im Parser von EspressoGrinder der Strichpunkt, die zum aktuellen Block gehörende schließende geschweifte Klammer sowie das Schlüsselwort `class`. Treten während des Zerteilens Fehler auf, so wird am Ende der syntaktischen Analyse der Übersetzungsvorgang abgebrochen.

Zur Veranschaulichung des Übersetzungsvorgangs werden die einzelnen Übersetzungsphasen anhand einer kleinen Beispielklasse `XYZ` betrachtet.

```
class XYZ
{
    long a;

    void fn(byte x)
    {
        int b = x + 16;
        this.a = b;
    }
}
```

In Abbildung 8 ist der abstrakte Syntaxbaum dargestellt, der vom Compiler während der syntaktischen Analyse für die Klasse `XYZ` intern aufgebaut wird. Für jeden Knoten des Syntaxbaums wurden nur die wichtigen Attribute angegeben. Attribute, die erst während der semantischen Analyse ausgewertet werden, sind im Diagramm kursiv gedruckt.

9.4 Semantische Analyse

Während der semantischen Analyse wird der vom Parser erzeugte abstrakte Syntaxbaum auf semantische Korrektheit hin untersucht. Hierzu wird der Syntaxbaum durchlaufen und für jeden Knoten werden zugehörige Attributwerte durch Auswertung von semantischen Regeln bestimmt. Zur Bestimmung der Attributwerte an den Knoten werden wechselseitig rekursive Methoden verwendet, die die Semantikregeln, die mit dem entsprechenden AST-Knoten assoziiert sind, simulieren.

9.4.1 Umgebungen und Gültigkeitsbereiche

Um das „Weiterreichen“ ererbter Attribute zu vereinfachen, werden die meisten vererbten Attribute in *Umgebungen* zusammengefaßt. Für jedes Sprachkonstrukt, in dem Deklarationen möglich

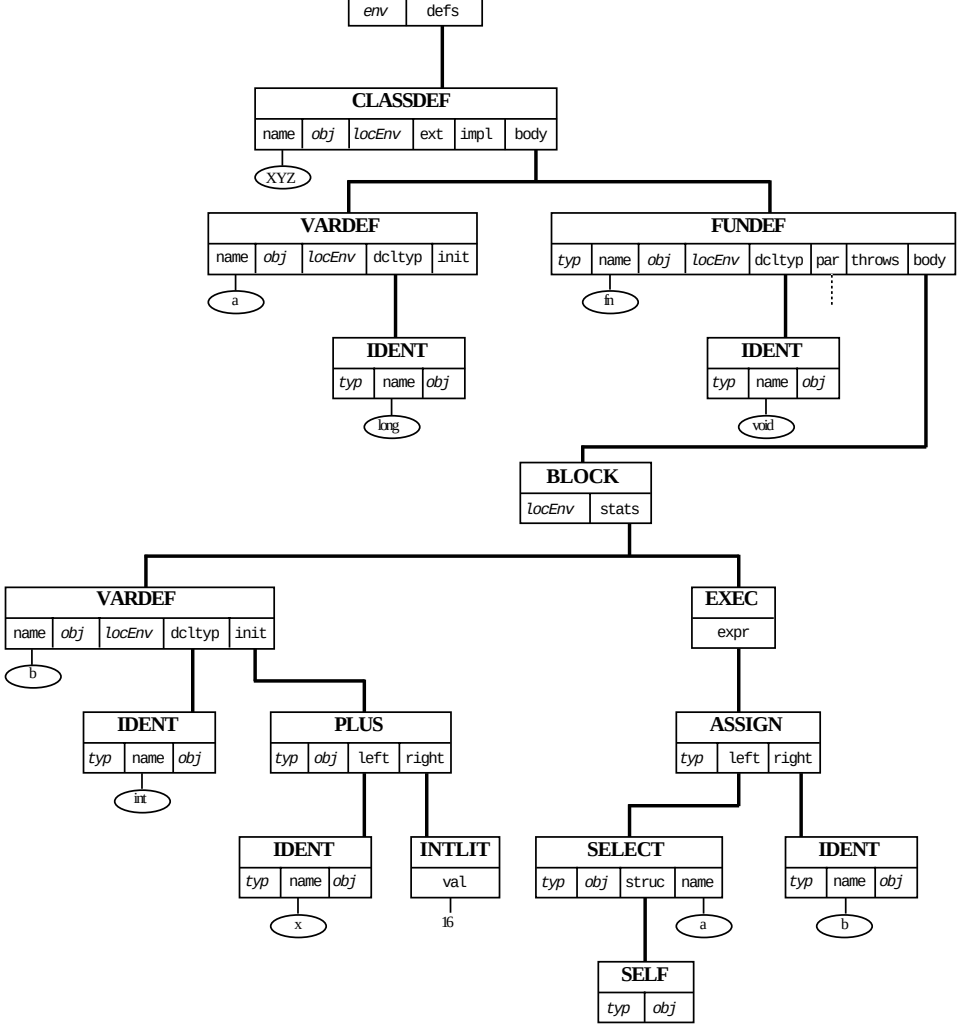


Abbildung 8: Beispiel zur syntaktischen Analyse

sind, wird jeweils eine eigene Umgebung angelegt, die den Namensraum der zugehörigen lexikalischen Ebene repräsentiert. Da diese Sprachkonstrukte geschachtelt sein können, sind dies natürlich auch die damit assoziierten Umgebungen. Die Klasse `Env` repräsentiert eine solche Umgebung innerhalb von EspressoGrinder. Abbildung 9 erklärt alle Attribute von `Env`.

Wie man aus Abbildung 9 erkennen kann, ist eine Umgebung stets mit einem eigenen *Gültigkeitsbereich* verknüpft. Die Gültigkeitsbereichs-Informationen werden in Instanzen der Klasse `Scope` verwaltet. Diese Klasse stellt nichts anderes als eine Realisierung einer Symboltabelle dar, in der bei der Attributierung des abstrakten Syntaxbaums Bindungsinformationen deklarierter Namen abgelegt werden. Immer wenn man bei der semantischen Analyse auf einen Bezeichner stößt, kann man im aktuellen Gültigkeitsbereich nach dem mit ihm verknüpften, bei der Deklaration definierten und dabei in die Symboltabelle eingetragenen Objekt suchen. Deswegen sind effizientes Einfügen und vor allem effizientes Auffinden innerhalb der Tabelle wichtig. Da implizit Gültigkeitsbereiche geschachtelt sind, muß es ebenso effizient möglich sein, einen neuen inneren Gültigkeitsbereich aufbauend auf einem äußeren Bereich anzulegen.

`Scope` realisiert die Symboltabelle mittels einer Hashtabelle. Die Überlauflisten werden direkt

`Env`	`next;`	Nächst äußere Umgebung
`AST`	`parent;`	Knoten, zu dem die Umgebung gehört
`TopLevel`	`toplevel;`	AST-Knoten der zugehörigen Übersetzungseinheit
`FunDef`	`enclMeth;`	Methode, in der die Umgebung sich befindet
`ClassDef`	`enclClass;`	Klasse, in der die Umgebung liegt
`Name`	`packageName;`	Name der zugehörigen Package
`Scope`	`scope;`	Zur Umgebung gehörender Gültigkeitsbereich
`Scope`	`globalScope;`	Zugehöriger `TopLevel`-Gültigkeitsbereich
`Scope`	`importScope;`	Gültigkeitsbereich der *Import-Handles*
`TypSet`	`reported;`	Menge aller abzufangenden und weiterzureichenden Ausnahmen
`boolean`	`isStatic;`	Gehört Umgebung zur Klasseninitialisierung?
`boolean`	`isSelfCall;`	Werden Argumente für einen `this`- oder `super`-Konstruktor-aufruf ausgewertet?

Abbildung 9: Schnittstelle der `Env`-Klasse

mit `ScopeEntry` Instanzen gebildet, in denen auch die zu verwaltenden Informationen abgelegt sind.

Aus Effizienzgründen ist es möglich, daß sich zwei ineinander geschachtelte Gültigkeitsbereiche eine Hashtabelle teilen. In den meisten Fällen ist dies auch wirklich sinnvoll, da die lexikalischen Sichtbarkeitsregeln von Java fordern, daß innerhalb eines Gültigkeitsbereichs auch alle die Bezeichner sichtbar sind, die in äußeren Gültigkeitsbereichen bzw. Namensräumen liegen und nicht durch innere Deklarationen verdeckt werden. Deswegen legt EspressoGrinder beispielsweise für Blöcke niemals `Scope` Instanzen mit eigener Hashtabelle an, sondern verwendet die Tabelle der nächst äußeren Umgebung mit. Damit muß es aber auch möglich sein, alle Einträge die nur einen Gültigkeitsbereich betreffen, beim Verlassen dieses Bereichs aus der Tabelle wieder ausräumen bzw. bei einem erneuten Betreten wieder einräumen zu können. Hierfür werden alle Einträge eines Gültigkeitsbereichs zusätzlich noch linear verkettet, wodurch ein einfacher Durchlauf der Liste zum Ein- bzw. Ausräumen der Einträge in die bzw. aus der Hashtabelle genügt.

In einigen Fällen ist es jedoch sinnvoller, eine eigene Hashtabelle für einen Gültigkeitsbereich zu verwenden. Die Transparenz zu dem nächst äußeren Bereich wird durch ein Kopieren der Hashtabelle, nicht der Einträge selbst, erreicht. Beispielsweise werden für die Gültigkeitsbereiche, die den Namensraum einer Klasse oder einer Übersetzungseinheit repräsentieren, jeweils eigene Hashtabellen angelegt.

Um einen Eindruck von den verschiedenen Umgangsformen mit Symboltabellen zu geben, sind in der Schnittstellenbeschreibung der Klasse `Scope` in Abbildung 10 neben den Variablen auch alle relevanten Methoden aufgeführt. In welchen Fällen, welche Form einer Symboltabelle konkret verwendet wird, kann hier, aufgrund der vielen Fälle, nicht näher diskutiert werden.

9.4.2 Deklarationen

In Java können auf verschiedenen Ebenen Klassen, Funktionen und Variablen deklariert werden. Nach der syntaktischen Analyse liegen solche Deklarationen innerhalb des abstrakten Syntaxbaums in Form von `ClassDef`-, `FunDef`- und `VarDef`-Knoten vor. Abbildung 11 zeigt die Klassenhierarchie dieser AST-Knoten. Die Attribute einer Klasse sind in dieser Abbildung meist in zwei Gruppen geteilt. Die obere Gruppe gibt die Attribute an, die direkt vom Parser ausgewertet werden, die Attribute der unteren Gruppe werden in der semantischen Analyse berechnet.

`Scope next;`	Nächst äußerer Gültigkeitsbereich
`Object obj;`	Knoten zu dem die Umgebung gehört
`ScopeEntry[] hashtable;`	Hashtabelle aller Einträge
`ScopeEntry elems;`	Anker für linear verkettete `ScopeEntry`-Liste aller Einträge dieses Gültigkeitsbereichs
`Scope(Scope next, Obj owner, ScopeEntry hashtable[]);`	Neues `Scope`-Objekt das die übergebene Hashtabelle mitbenutzt
`Scope(Scope next, Obj owner);`	Neues `Scope`-Objekt mit eigener neuer Hashtabelle
`Scope duplicate();`	Bettet in aktuellen `Scope` neuen `Scope` ein, der mit dem äußeren `Scope` eine Hashtabelle teilt
`void baseOn(Scope s);`	Plaziert `Scope` direkt in `Scope s` durch Kopieren der Referenzen auf alle Überlauflisten von `s`
`void include(Scope s);`	Fügt alle Einträge aus `s` ein
`void enter(Obj obj);`	Fügt neues Objekt ein
`ScopeEntry lookup(Name name);`	Sucht nach einem bestimmten Objekt
`void leave();`	Räumt alle Einträge des `Scopes` aus der Hashtabelle aus
`void restore();`	Fügt alle Einträge des `Scopes` wieder in Hashtabelle ein

ScopeEntry	Bedeutung
`Obj obj;`	Referenziertes Objekt
`ScopeEntry shadowed;`	Nächster Eintrag in der Überlaufliste
`ScopeEntry sibling;`	Nächster Eintrag im Gültigkeitsbereich
`Scope scope;`	Zugehöriger Gültigkeitsbereich
`ScopeEntry next();`	Liefert nächstes gleichnamiges Objekt

Abbildung 10: Schnittstelle der `Scope`-Klasse

Durch eine Deklaration wird ein Bezeichner an einen Verwendungszweck gebunden. Während der semantischen Analyse wird für jede Deklaration ein Objekt angelegt, das alle notwendigen Bindungsinformationen zusammenfaßt. Dieses Objekt charakterisiert also, im Gegensatz zum mehrdeutigen Bezeichner, eindeutig eine konkrete Variable, Klasse oder Funktion. Diese Objekte werden direkt als Symboltabelleneinträge in dem Gültigkeitsbereich einer Umgebung abgelegt. Beschrieben werden solche Objekte in EspressoGrinder durch Unterklassen der Klasse `Obj`. Diese enthält als grundlegende Bindungsinformationen im wesentlichen, die mit der Deklaration verbundenen Modifikatoren, den deklarierten Namen, den zugehörigen Typ, sowie einen Verweis auf das Objekt, innerhalb dessen die Deklaration stattfand. Die Klassenhierarchie der AST-Deklarationsknoten impliziert eine duale Struktur auf Seiten der `Obj` Klassen. Abbildung 12 zeigt diese Hierarchie.

9.4.3 Typsystem

Java ist stark typisiert, d.h. für jede Variable und jeden Ausdruck ist bereits zur Übersetzungszeit der Typ bestimmbar. Das Java Typsystem teilt Datentypen grob in zwei Kategorien ein: eingebaute, *einfache Typen* mit Werte-Semantik der Objekte und *Referenztypen* mit Referenz-Semantik der Objekte. Die Referenztypen gliedern sich in Klassentypen, Interfacetypen und Feldtypen. Um die statische Typprüfung innerhalb von EspressoGrinder zu vereinfachen, wird diese Unterscheidung nach außen hin verdeckt, indem alle vorkommenden Typen als Instanzen der Klasse `Typ` vereinbart werden (Abbildung 13).

Über die Kennung `tag` kann der vorliegende Typ genau identifiziert werden. Abbildung 14 listet alle *Typ-Tags* auf. Jeder Datentyp wird genau durch eine Klasse `obj` definiert. Interfaces werden intern wie Klassen behandelt und definieren selbstverständlich stets auch einen Typ. Da einfache Typen in Java keine definierende Klasse besitzen, wird bei der Initialisierung dieser Typen jeweils

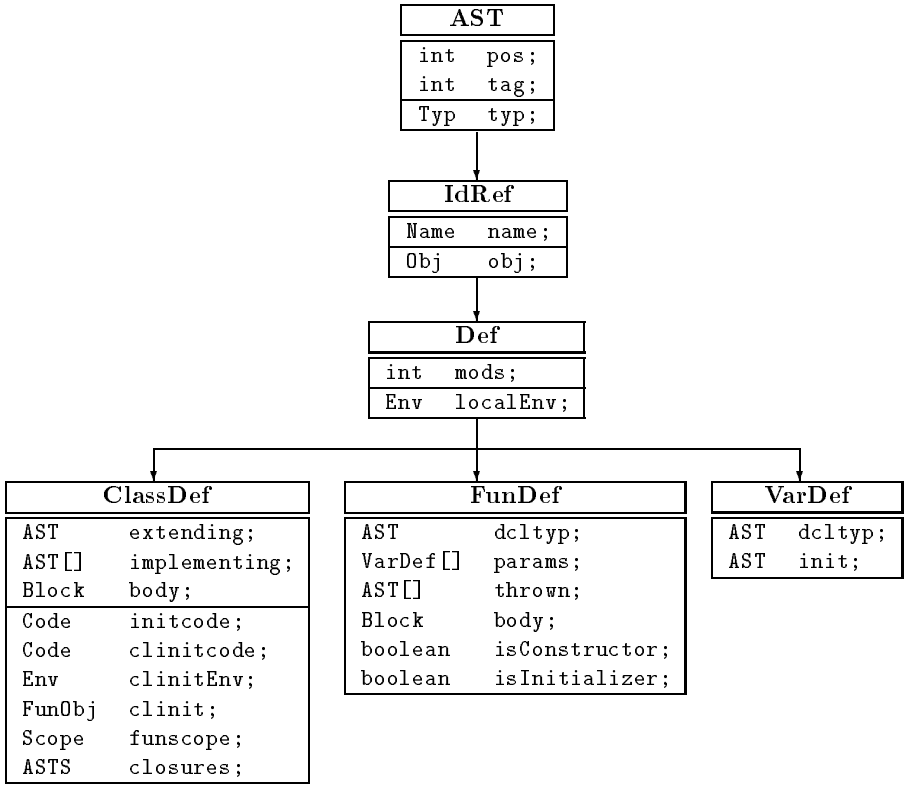


Abbildung 11: Klassenhierarchie der AST-Deklarationsknoten

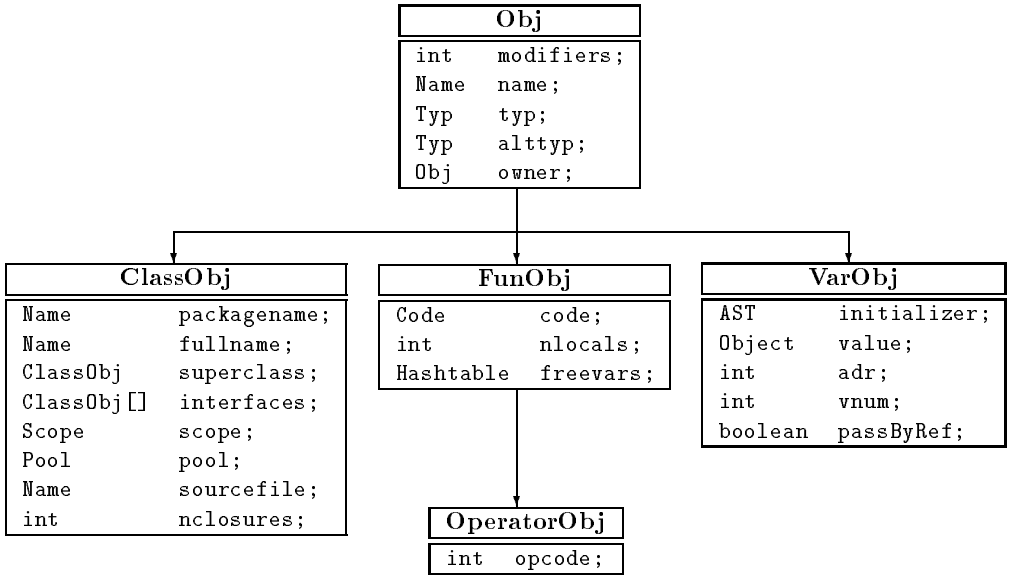


Abbildung 12: Klassenhierarchie von Symboltabelleneinträgen

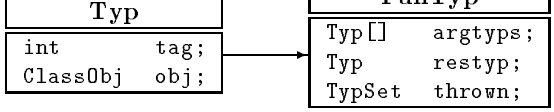


Abbildung 13: Typenrepräsentation in EspressoGrinder

ERRTYP	INT	BOOLEAN	FUNCTION
BYTE	LONG	CLASS	NULLTYP
CHAR	FLOAT	VOID	ANYTYP
SHORT	DOUBLE	ARRAY	PACKAGETYP

Abbildung 14: Typ-Tags

ein Klassenobjekt angelegt, das intern den Typ repräsentiert, sonst aber bedeutungslos ist.

Neben den erwähnten Typen werden in EspressoGrinder auch noch einige zusätzliche Typen `ERRTYP`, `NULLTYP` und `ANYTYP` definiert, die effektiv das Typsystem, das zur Spezifikation korrekter Java Programme benötigt wird, erweitern. Diese vereinfachen bzw. vereinheitlichen den Typprüfungsvorgang und ermöglichen es beispielsweise erst, daß sich der Typüberprüfer nach dem Entdecken eines Typfehlers wieder erholt und noch sinnvoll den restlichen Teil des Syntaxbaums überprüfen kann.
Eine weitere Auswirkung dieser Erweiterung ist die nun erst mögliche Realisierung der Idee, ausnahmslos jedem Konstrukt der Sprache, also nicht nur Ausdrücken, einen Typ zuzuordnen. Warum dies sinnvoll ist, erkennt man besonders wieder im Fehlerfall: tritt ein Typfehler innerhalb einer Anweisung auf, so vererbt sich der zugeordnete Typ `ERRTYP` zunächst auf die gesamte Anweisung, danach auf den umschließenden Block, bis schließlich die gesamte Übersetzungseinheit den Fehlertyp besitzt.

Bisher unerwähnt blieb die `Typ`-Unterklasse `FunTyp`, die einen Funktionstyp darstellt. Für alle Methoden wird der korrespondierende Funktionstyp aus den Argumenttypen, dem Ergebnistyp und den Ausnahmetypen der Methode synthetisiert. Über Funktionstypen wird bei der Bewertung des Syntaxbaums die Zulässigkeit von Methodenaufrufen nachgeprüft bzw. bei überladenen Methoden wird hiermit die Bestimmung der konkret aufzurufenden Methode erst ermöglicht.

9.4.4 Bewertung des abstrakten Syntaxbaums

EspressoGrinder führt die semantische Analyse in drei getrennten Durchläufen durch die abstrakten Syntaxbäume durch, wobei jedoch nicht immer ein gesamter Baum betrachtet wird. Jeder Durchlauf wird jeweils ausgehend von der Wurzel eines Syntaxbaums gestartet. In den ersten beiden Durchläufen werden hauptsächlich Attribute vererbt, während im dritten Lauf im wesentlichen Attribute synthetisiert und Überprüfungen durchgeführt werden.

Der Grund für die Bewertung des Baums in mehreren Phasen liegt vor allem in der durch Java vorgegebenen Hierarchie der Namensräume. Diese wird durch die in Abbildung 15 gezeigte Anordnung von Gültigkeitsbereichen innerhalb des Compilers nachgebildet. Einträge aus äußeren Gültigkeitsbereichen sind stets vollständig in eingebetteten Bereichen sichtbar. Bei geschachtelten Gültigkeitsbereichen, die innerhalb des Gültigkeitsbereichs für die Parameter einer Methode, in Abbildung 15 Methodengültigkeitsbereich genannt, liegen, gelten andere Sichtbarkeitsregeln. Hier sind in einem inneren Gültigkeitsbereich nur die Vereinbarungen sichtbar, die textuell vor dem inneren Bereich in äußeren Gültigkeitsbereichen gemacht wurden.

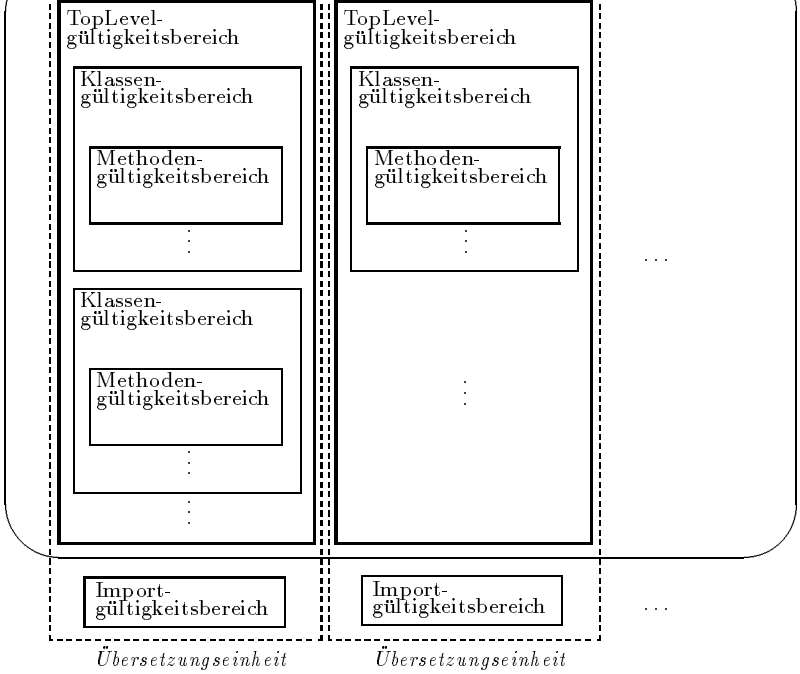


Abbildung 15: Gliederung von Gültigkeitsbereichen

Der Aufbau und die Initialisierung der geschachtelten Gültigkeitsbereiche läuft von außen nach innen ab. Der globale Gültigkeitsbereich `Predef.scope` enthält vordefinierte Bezeichner, Operatoren in allen überladenen Varianten sowie alle vordefinierten Typen. Bei der Initialisierung dieses Gültigkeitsbereichs werden auf jeden Fall die Klassen `Object`, `String`, `RuntimeException`, `Throwable` und `Error` geladen. Für Feldzugriffe wird außerdem ein eigener Gültigkeitsbereich `Predef.arrayScope` angelegt.

1. Durchlauf: Für jede Übersetzungseinheit beginnt der erste Durchlauf durch den abstrakten Syntaxbaum mit dem Einrichten einer `TopLevel`-Umgebung, in welche dann während des Durchlaufs der Packagename sowie entsprechende Deklarationsobjekte für alle vereinbarten Klassen eingetragen werden. Dazu genügt es, lediglich die `Package`- und `ClassDef`-Knoten in der obersten Ebene des Baums zu besuchen. Das Aufsammeln aller deklarierten Klassen muß, bevor mit dem zweiten Durchlauf begonnen werden kann, für alle Übersetzungseinheiten abgeschlossen sein, da Java keine Konventionen besitzt, daß eine Klasse im Quelltext zuerst vereinbart werden muß, bevor sie als Typangabe verwendet werden darf.

2. Durchlauf: Der zweite Durchlauf hat zwei grundlegende Aufgaben. Zum einen werden die Import-Anweisungen ausgewertet, zum anderen werden die Felder der definierten Klassen in den jeweiligen lokalen Gültigkeitsbereich der Klasse eingetragen.

Das Auswerten einer Import-Anweisung besteht im Generieren von *Import-Handles*, welche in den entsprechenden Import-Gültigkeitsbereich der `TopLevel`-Umgebung eingetragen werden. Dieser Bereich enthält nur solche Verweise. Handelt es sich bei dem Argument einer Import-Anweisung um eine Package, dann werden für alle existierenden Klassen der Package Import-Handles erzeugt und eingetragen. Ein solches Import-Handle stellt nichts anderes als eine Beziehung zwischen einer Namensabkürzung und dem voll qualifizierten Namen einer Klasse dar.

dafür, daß Klassendateien erst geladen werden, wenn auch wirklich ein Zugriff auf die Klasse erfolgt.

Zum Auswerten der Felder einer Klassendefinition wird zunächst ein eigener Gültigkeitsbereich angelegt, der den Namensraum der Klasse widerspiegelt. Darüberhinaus werden Referenzen auf die, durch die `extends`- und `implements`-Klauseln bestimmten Klassenobjekte, ermittelt. Dabei kann es vorkommen, daß die Klassendatei einer referenzierten Klasse erst geladen werden muß. Außerdem müssen dabei die Restriktionen, die für Oberklassen und Interfaces gelten, nachgeprüft werden.
Nun werden alle Felddeklarationen in den Klassengültigkeitsbereich eingetragen, wobei stets sichergestellt werden muß, daß keine doppelten Deklarationen vorliegen. Für eine Variable einer Klasse reichen eine konsistente Modifikatorenmenge und der Typ für das Anlegen eines neuen Symboltabelleneintrags aus. Man beachte jedoch, daß das Ermitteln des Typs möglicherweise wiederum zum Nachladen einer Klasse führen kann.
Handelt es sich bei dem Feld um eine Methode, so wird auch hier der zugehörige Funktionstyp ermittelt, was stets mehrere Konsistenzprüfungen nach sich zieht. Außerdem wird eine eigene lokale Umgebung, die den Parameternamensraum der Methode repräsentiert, eingerichtet und initialisiert.

Werden innerhalb einer Klassendefinition keine Konstruktoren angegeben, so erzeugt der Übersetzer selbst einen einfachen Konstruktor, der lediglich den Konstruktor der Oberklasse aufruft. Dazu wird der abstrakte Synatxbaum der Klassendefinition um einen neuen `FunDef`-Knoten ergänzt.

Vererbte Merkmale können im allgemeinen zum jetzigen Zeitpunkt noch nicht in die Umgebung der Klasse eingetragen werden, da die Felder für Oberklassen und Interfaces möglicherweise noch nicht vollständig bekannt sind. Stattdessen findet die Herstellung der vollständigen Klassenumgebung erst später statt, sobald die Umgebung der Klasse das erste Mal benötigt wird. Dieser Vorgang ist rekursiv und zieht möglicherweise entlang der Oberklassenordnung eine ganze Reihe solcher Vervollständigungen nach sich.

3. Durchlauf: Im letzten Durchlauf werden schließlich für die Knoten der gesamten abstrakten Syntaxbäume die restlichen Attribute, die nicht die Codegenerierung betreffen, ausgewertet. Dabei handelt es sich im wesentlichen um vier zu erledigende Aufgaben:

1. Synthetisierung des Typs aller Konstrukte im Syntaxbaum
2. Typverifizierung aufbauend auf dem ermittelten Typ
3. Namensauflösung und damit zusammenhängende Überprüfungen
4. Sprungzielermittlung und Sammlung von Kontextinformationen der Sprunganweisungen

Initiiert werden alle drei Aufgaben von jeweils wechselseitig rekursiven Methoden `attr`, die aufbauend auf weitergereichten ererbten Attributen die synthetisierten Attribute an einem Knoten ableiten und je nach Knotentyp Konsistenzprüfungen durchführen. Die AST-Knoten für die meisten Anweisungskonstrukte von Java haben in dieser Analyse nur eine vermittelnde Rolle, ihr eigener Typ hat nur die Funktion, Fehler anzuzeigen.

In den übrigen Knotentypen treten Typsynthetisierung und -verifizierung meist in Kombination auf. Der Typprüfer stellt hier nichts anderes dar, als eine Implementierung des zugrundeliegenden Typsystems. Er verifiziert, daß der abgeleitete Typ eines Konstruktes mit dem Typ

diese Verifikation ist oftmals erst eine erfolgreich durchgeführte Namensauflösung.

Ziel der Namensauflösung ist es, je nach Kontext, eine Verbindung von einem Bezeichner zu einer Klasse, einem Feld oder einer lokalen Variablen herzustellen. Bezüge zu Variablen lassen sich relativ einfach mittels der aktuellen Symboltabelle bestimmen. Bei Methoden sieht die Sache, durch die Möglichkeit der Methodenüberladung, etwas komplizierter aus: es müssen alle überladenen Methoden mit betreffendem Bezeichner betrachtet und darunter diejenige Variante herausgesucht werden, die formell am besten zum aktuellen Funktionstyp paßt. Auf diese Weise wird bereits zur Übersetzungszeit, bis auf Methodenüberschreibung, die zur Laufzeit konkret aufzurufende Methode bestimmt. In Bezug auf die Typbestimmung wird hiermit effektiv aus einer Menge möglicher Typen, der tatsächliche Typ ausgewählt. Nun wird auch klar, warum im zweiten Durchlauf alle Methoden erst einmal in der Umgebung der eigenen Klasse eingetragen werden mußten. Ohne diesen Schritt könnte man jetzt im allgemeinen nichts über die aufzurufende Methode und damit auch nichts über den mit ihr assoziierten Typ aussagen.

Ein Beispiel soll den Vorgang der Namensauflösung für einen überladenen Methodenaufruf verdeutlichen. Es sei folgende, hier unvollständig abgedruckte Klassendefinition gegeben:

```
class Maximum
{
    static int max(int x, int y) { ... }
    static int max(int x, int y, int z) { ... }
    static float max(float x, float y) { ... }
}
```

Innerhalb des Namensraums der Klasse `Maximum` gibt es drei Methoden mit gleichem Namen aber unterschiedlicher Signatur. In der Symboltabelle des Klassengültigkeitsbereichs befinden sich diese drei Einträge innerhalb einer einzigen Überlaufliste. Es wird nun folgender Aufruf der überladenen Methode `max` betrachtet: „`float res = Maximum.max(2.0,3.0)`“. Die Attributierungsfunktion für den dazugehörigen `Apply`-Knoten im abstrakten Syntaxbaum ermittelt zunächst die Typen der aktuellen Parameter und bestimmt damit den Funktionstyp der aufzurufenden Methode. Im Beispiel ist dies der Typ `(float, float -> Typ.anyTyp)`. Innerhalb des Compilers bestimmt dann eine universelle Routine zur Namensauflösung `Attr.resolve` aus allen überladenen Versionen von `max`, diejenige Variante mit dem am besten passenden Typ. Diese Typen wurden ja bereits im zweiten Durchlauf aus den formalen Parametern der Methoden abgeleitet. Im Beispiel ist die letzte Methode mit dem Funktionstyp `(float, float -> float)` die einzige, zum Methodenaufruf passende Funktion, womit das zum Bezeichner `Maximum.max` in diesem Beispiel gehörende Funktionsobjekt gefunden wäre.

Bei der Sprungzielermittlung geht es um die Bestimmung der Sprachkonstrukte, im abstrakten Syntaxbaum repräsentiert durch Unterklassen von `JumpTarget`, auf die sich Sprunganweisungen, dargestellt durch `Jump`-Unterklassen, beziehen. Dazu werden, ausgehend von der aktuellen Umgebung, die äußeren Umgebungen nacheinander betrachtet, bis schließlich das betreffende Spachkonstrukt gefunden wird, aus welchem mittels der Sprunganweisung herausgesprungen werden soll. Für alle `try-catch`-Anweisungen, die während dieser Analyse betrachtet werden, müssen – falls vorhanden – die zugehörigen `finally`-Klauseln aufgesammelt werden. Nur so kann der Compiler dafür sorgen, daß das Ausführen eines Sprungs stets das Ausführen der `finally`-Klauseln aller verlassenen `try-catch`-Blöcke zur Folge hat.

Für die Beispielklasse `XYZ` aus Abschnitt 9.3.3 wird in Abbildung 16 der, durch die semantische Analyse ermittelte, attributierte Syntaxbaum schematisch dargestellt. Da schon bereits bei einem sehr kleinen Beispiel die Verweisstruktur im Baum, aufgrund der Komplexität, nicht mehr

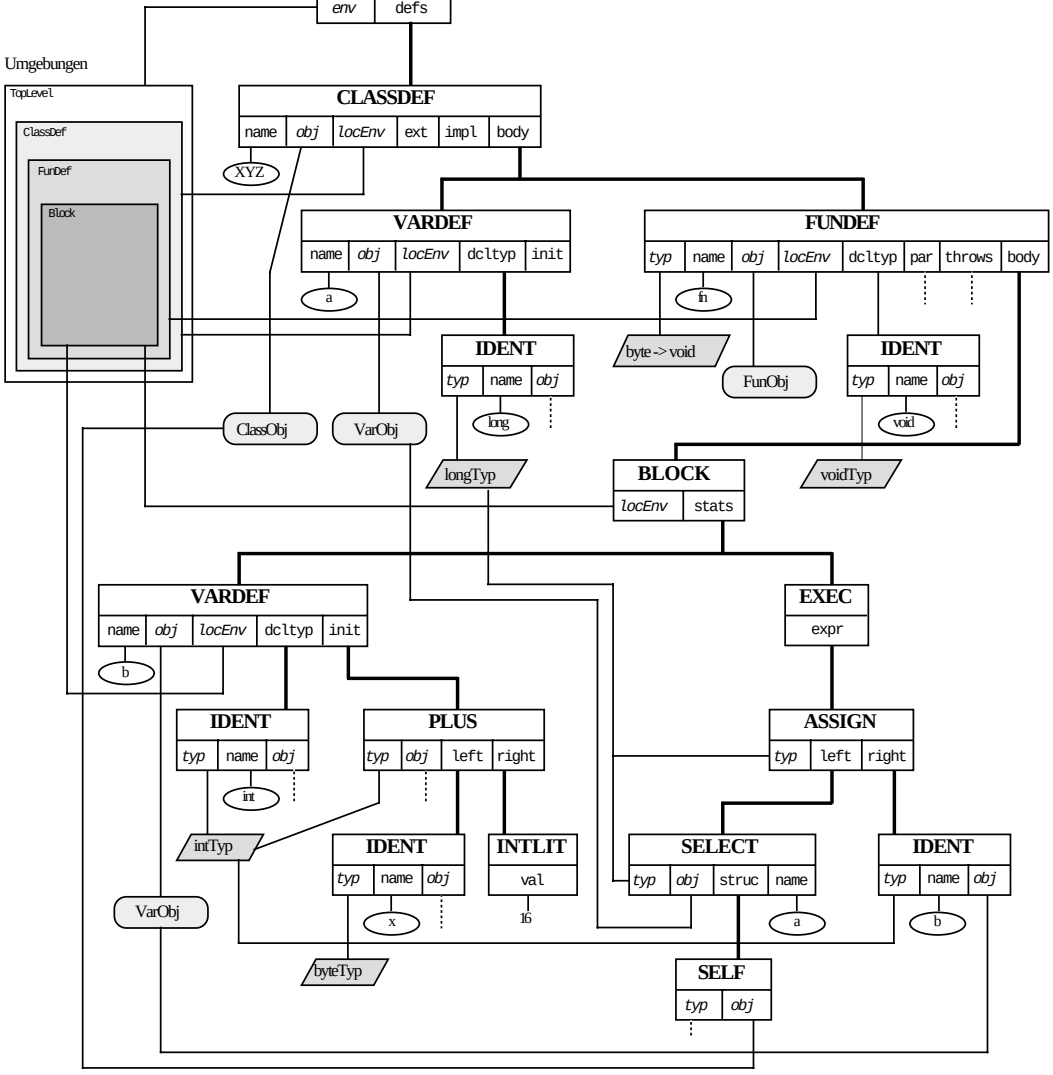


Abbildung 16: Beispiel zur semantischen Analyse

überschaubar dargestellt werden kann, wurden im Diagramm nur die wichtigsten Verweise eingetragen. Auch auf die Initialisierung der während der semantischen Analyse erzeugten Objekte kann hier nicht näher eingegangen werden. Außerdem wurde, der Übersicht wegen, auf die Darstellung, des im zweiten Durchlaufs für die Klasse erzeugten Standardkonstruktors, verzichtet.

9.4.5 Klassendateiverwaltung

Die Klassendateiverwaltung wurde mit der Klasse `ClassFile` realisiert. Sie stellt Klassendateilade- und -speicherfunktionen zur Verfügung und verwaltet alle bereits geladenen und neu zu generierenden Klassen. In der Hashtabelle `loaded` werden hierzu unter dem vollständigen Klassennamen die bekannten Klassen in Form von Klassenobjekten abgelegt. Über diese Tabelle kann dann jederzeit das zu einer referenzierten Klasse gehörige `ClassObj` bestimmt werden. Wird dabei festgestellt, daß sich eine Klasse noch nicht im Speicher befindet, wird sie automatisch geladen. Das Laden einer Klasse zieht im allgemeinen auch das Laden der Oberklasse und aller

Klassenumgebung mit allen vererbten Merkmalen hergestellt werden kann.

9.5 Codeerzeugung

Während der Phase der Codeerzeugung wird für alle Methoden, Konstruktoren und Klasseninitialisierungsblöcke *Bytecode* generiert. Es wird keine Zwischendarstellung verwendet. Der Bytecode für die *Java Virtual Machine* (kurz JVM) ist bereits so einfach, daß man den abstrakten Syntaxbaum direkt zur Generierung des Zielcodes verwenden kann. Auch eine eigene Codeoptimierungen wird zur Zeit, abgesehen von Konstantenfaltung und vom Entfernen passiven Codes, nicht durchgeführt. Der erzeugte Code ließe sich zwar sicher an einigen Stellen noch optimieren, der zu erwartende Gewinn würde, auch im Hinblick auf die Interpretierung des maschinenunabhängigen Bytecodes, allerdings den hierfür nötigen Aufwand nicht rechtfertigen.

9.5.1 Coderepräsentation

EspressoGrinder erzeugt für jede Klasse eine Klassendatei, deren Aufbau in [23] genau beschrieben wird. Darin werden, neben Informationen bezüglich der Oberklasse und implementierten Interfaces, für jede Variable und Methode der Klasse ein Datenblock mit allen nötigen Informationen über das Merkmal abgelegt. Referenzen auf Konstanten, wie z.B. Namen, Signaturen und konstante Werte, werden dabei stets als Index auf einen Eintrag im *Konstantenpool* der Klasse gespeichert. Der Konstantenpool ist ein lineares Feld, in welchem die Werte, aller in der Klasse vorkommenden Konstanten, gesammelt sind. Informationen für den Debugger, wie etwa die *LineNumberTable*, die eine Zuordnung von Quellcode zu Bytecode ermöglicht, die *LocalVariableTable*, die lokale Variablen verzeichnet, aber auch der Code für eine Methode selbst, werden als Attribute zu dem Datenblock eines Merkmals gespeichert. Alle hierfür benötigten, und in den bisherigen Phasen noch nicht ermittelten Daten, müssen während der Codeerzeugungsphase bestimmt werden. Dies beschränkt sich weitgehend auf die Auswertung des `Code`-Attributs von `ClassDef`- und `FunDef`-Knoten des abstrakten Syntaxbaums. Die Klasse `Code` faßt alle relevanten Werte zusammen. Wie man in Abbildung 17 sieht, enthält diese Klasse ebenso auch alle Informationen, die zur Debuggerunterstützung wahlweise mit in die Klassendatei geschrieben werden können.

Code		Bedeutung
`Obj`	`owner;`	Objekt zu dem der Code gehört
`boolean`	`isStatic;`	Code für Klasseninitialisierung?
`Bits`	`uninits;`	BitSet aller lokalen Variablen (Bit ist gesetzt, wenn Variable initialisiert)
`int`	`max_locals;`	Maximale Anzahl lokaler Variablen
`int`	`max_stack;`	Maximale Stackgröße
`byte[]`	`code;`	Codepuffer
`int`	`cp;`	Zeiger auf aktuelle Instruktion
`exc_table`	`exceptionTable;`	Tabelle aller abzufangenden Ausnahmen
`ln_table`	`lineNumberTable;`	LineNumberTable Attribut
`lvar_table`	`localVariableTable;`	LocalVariableTable Attribut

Abbildung 17: Interne Coderepräsentation

Um bei der Codeerzeugung bereits eine der Klassendatei nahe Darstellung des Codes zu verwenden, wird korrespondierend zum Konstantenpool der Klassendatei ein entsprechender Pool beim

tentabelle mittels eines dynamisch verwalteten Felds und einer zusätzlichen Hashtabelle, die ein schnelles Auffinden eines Eintrags innerhalb des Felds ermöglichen soll (Abbildung 18). Da die Klassendefinitionen in den abstrakten Syntaxbäumen nacheinander übersetzt und anschließend gleich in eine Datei geschrieben werden, genügt es, nur mit einem einzigen Konstantenpool zu arbeiten. Nach dem Übersetzen einer Klasse, muß dieser Pool dann zur erneuten Verwendung wieder zurückgesetzt werden.

Pool	Bedeutung
`Object[] pool;`	Feld aller abgelegten Konstanten
`ConstEntry[] constants;`	Hashtabelle aller Einträge (zum schnellen Suchen im Pool)
`int put(Object o);`	Konstante im Pool ablegen
`int get(Object o);`	Index einer Konstante im Pool ermitteln
`void reset()`	Pool zur Wiederverwendung zurücksetzen

ConstEntry	Bedeutung
`ConstEntry next;`	Nächster Eintrag in der Überlaufliste
`Object obj;`	Eigenlicher Eintrag im Konstantenpool
`int index;`	Index innerhalb des Pools

Abbildung 18: Realisierung eines Konstantenpools

9.5.2 Unterstützung der Codeerzeugung für die JVM

Die grundlegende Infrastruktur zur Codeerzeugung wird in EspressoGrinder durch die Klasse `Gen` bereitgestellt. Sie ist ganz auf die Architektur der JVM als Stackmaschine zugeschnitten.

Gen	Bedeutung
`static Code code;`	Aktuell zu generierender Code
`static Pool pool;`	Aktueller Konstantenpool
`static boolean alive;`	Ist momentan der auszugebende Code aktiv?
`static int stacksize;`	Aktuelle Stackgröße
`static int statPos;`	Quellcodeposition
`static int[] stackdiff;`	Tabelle, die angibt, wie OpCodes die Stackgröße beeinflussen

Abbildung 19: Unterstützung der Codeerzeugung

Lokale Variablen werden innerhalb der JVM für jede Methode getrennt als 32-bit-Wort Offset von Null an vergeben. `Gen` sorgt, neben der Offsetvergabe dafür, daß Variablen, bevor sie gelesen werden, auch initialisiert sind. Das Lesen uninitialisierter Variablen ist in Java unzulässig und führt zu einer Fehlermeldung.

Während der Codegenerierung wird durchgehend die aktuelle Größe des Operandenstacks ermittelt. Hierzu wird die Tabelle `Gen.stackdiff` herangezogen, die für jeden OpCode der JVM angibt, wie die Stackgröße beeinflußt wird.

Bei Sprunganweisungen wird untersucht, ob nachfolgender Code überhaupt eventuell auf anderem Weg erreicht werden kann oder ob er als passiver Code niemals abgearbeitet wird. EspressoGrinder entfernt passiven Code und liefert für nicht erreichbare Anweisungen entsprechend Warnungen.

Neben der Codeerzeugung wird von `Gen` die komplette Abwicklung der Konstantenpoolverwaltung durchgeführt.

Die Codeerzeugung wird mittels wechselseitig rekursiver Methoden `gen` durchgeführt. Diese geben, angewendet auf Knoten des abstrakten Syntaxbaums, Deskriptoren zurück, welche eine unmittelbare Codeerzeugung für die, durch die Knoten repräsentierten Sprachkonstrukte ermöglichen. In der Methode `gen` kann nicht immer direkt für den betreffenden Knoten Code erzeugt werden, da oftmals die Lage innerhalb des Kontextes für den konkret zu generierenden Bytecode maßgeblich ist. Betrachtet man beispielsweise die Zuweisung „`this.a = b`“, so ist für den rechten Bezeichner `b` Code für das Laden dessen r-Werts zu erzeugen, wohingegen beim linken Bezeichner `this.a` der l-Wert relevant ist und Code zum Sichern eines Wertes in der bezeichneten Variable auszugeben ist. Diese Unterscheidung kann aber nicht auf der Ebene der Bezeichner, sondern erst in der darüberliegenden Ebene der Zuweisung getroffen werden.

Die Deskriptoren beschreiben adressierbare Einheiten auf die lesend und schreibend zugegriffen werden kann. Innerhalb von EspressoGrinder stellt die Klasse `Item` mit ihren Unterklassen `LocalItem`, `BasedItem`, `ImmediateItem`, `AssignItem`, `CondItem`, `ClassItem` und `NewArrayItem` alle nötigen Deskriptor-Ausprägungen dar. Die Bedeutung der einzelnen `Item`-Unterklassen ist in Abbildung 20 beschrieben.

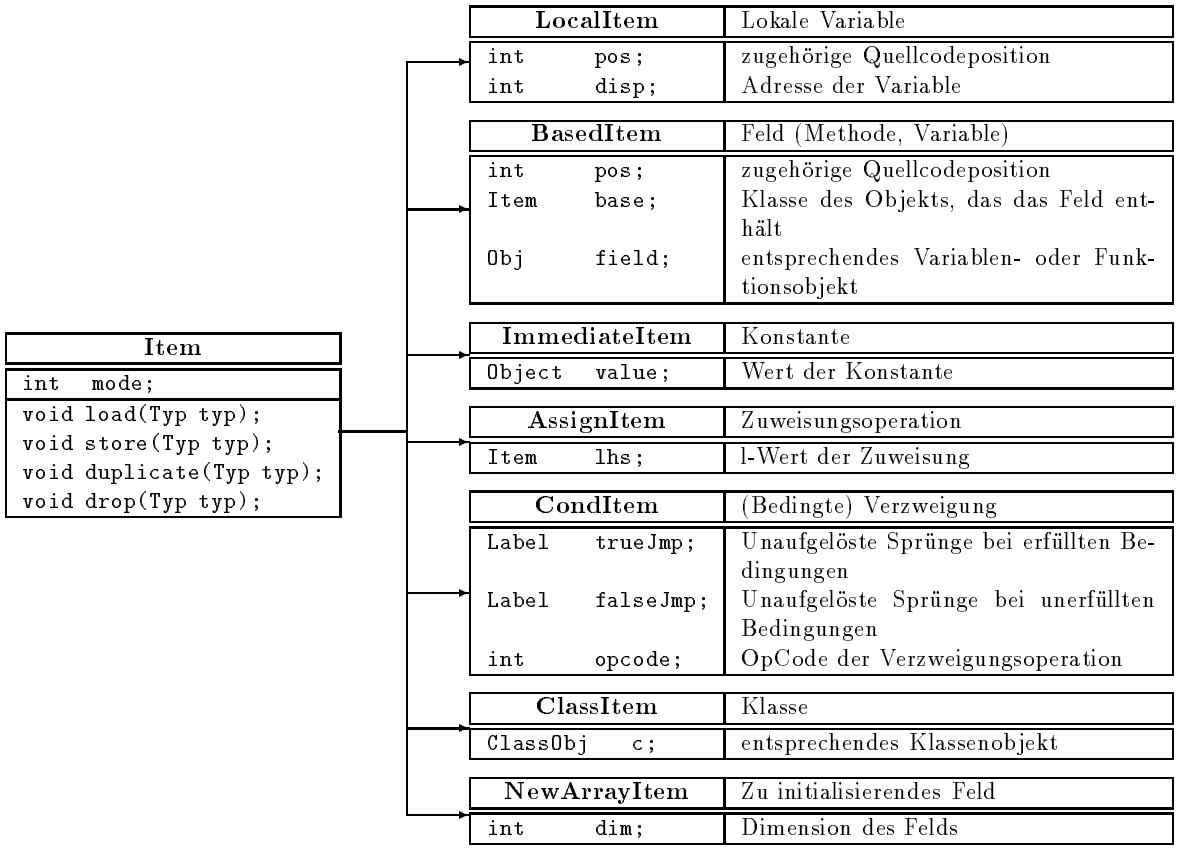


Abbildung 20: Adressierbare Einheiten

Einziges gemeinsames Feld aller Items ist `mode`, das die zugrundeliegende Adressierungsart kennzeichnet. Abbildung 21 listet alle möglichen Adressierungsmodi auf.

Für die verschiedenen Adressierungsarten, eigentlich müßte man eher von Zugriffsprotokollen sprechen, werden jeweils mehrere Methoden zur Codegenerierung definiert. Die Methoden `load`

STACKmode	SUPERmode	FIELDmode	NEWARRAYmode
INDEXEDmode	LOCALmode	METHmode	ASSIGNmode

Abbildung 21: Adressierungsarten

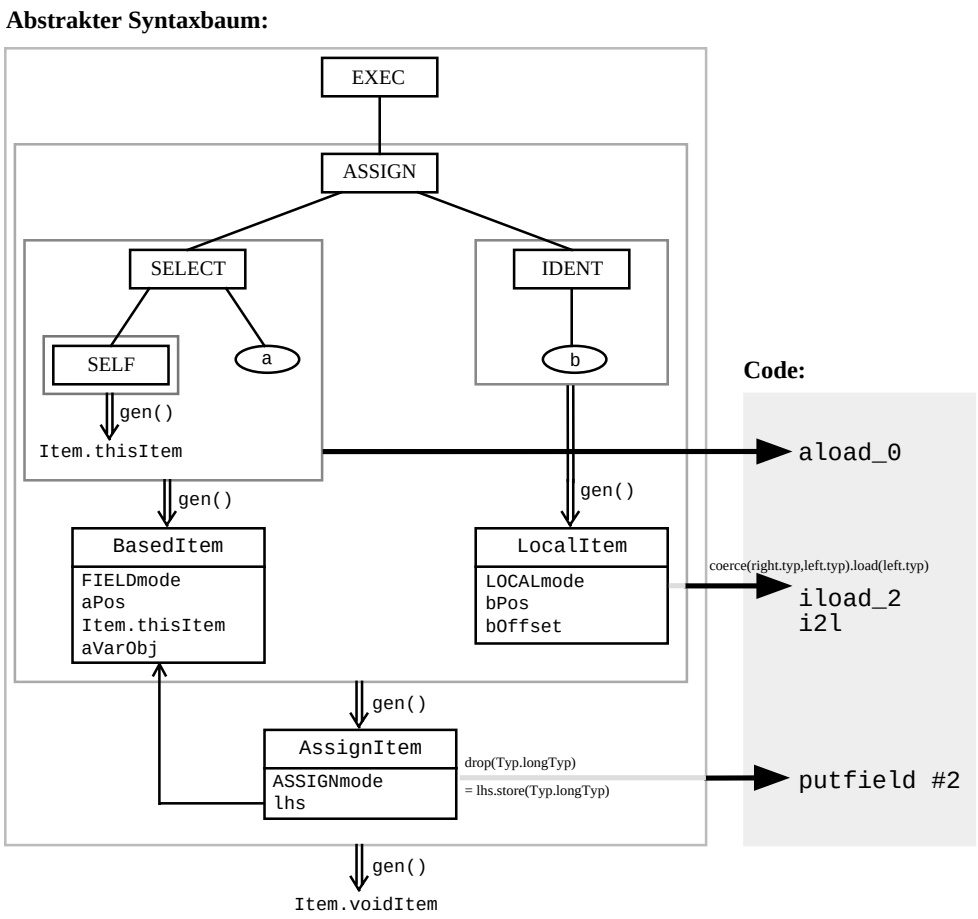


Abbildung 22: Codegenerierung für die Anweisung `this.a = b`

und `store` generieren Code für einen lesenden bzw. schreibenden Zugriff auf ein Objekt. Die Methode `duplicate` erzeugt Code, um ein Objekt zweimal verwenden zu können, `drop` generiert Code, der die weitere Verwendung eines Objekts verhindert. Die betreffenden Objekte werden jeweils durch ein Item und eine zugehörige Typangabe festgelegt.

Bevor nun die Codegenerierungstechnik, die auf der Verwendung der Items basiert, erläutert wird, muß noch auf spezielle Items eingegangen werden, die innerhalb des Compilers nur genau ein einziges Mal existieren. Es sind dies die Items mit den Adressierungsarten `VOIDmode`, `STACKmode`, `THISmode`, `SUPERmode` und `INDEXEDmode`.

Das Item `Item.voidItem` mit dem Adressierungsmodus `VOIDmode` charakterisiert eigentlich kein adressierbares Objekt und in keiner der Methoden des Zugriffsprotokolls wird folglich Code erzeugt. Alle Sprachkonstrukte von Java die zur Gliederung oder Kontrollflußsteuerung verwendet werden und sich damit auf kein adressierbares Objekt beziehen, wird dieses Item zugeordnet. Das Item `Item.stackItem` beschreibt genau das oberste Element auf dem Operandenstack. Eine sich auf dem Stack befindliche Referenz auf ein Array-Element, dargestellt durch zwei Ein-

`Item.thisItem` und `Item.superItem` referenzieren die aktuelle Instanz, wobei `Item.superItem` diese als Instanz der Oberklasse sieht.

In Abbildung 22 wird exemplarisch gezeigt, wie für obiges Beispiel „`this.a = b`“ Code generiert wird. Dabei wird angenommen, `this.a` sei ein Feld vom Typ `long` und `b` sei eine lokale Variable vom Typ `int`. Die obere Hälfte der Abbildung zeigt schematisch den abstrakten Syntaxbaum der Anweisung. In der unteren Hälfte werden die Items dargestellt, die durch Anwendung von `gen` auf die Teilbäume des abstrakten Syntaxbaums ermittelt werden. Die rekursive Funktionsweise von `gen` ist für die geschachtelte Struktur im Diagramm verantwortlich.

Wie man sieht, werden zunächst die Items für die linke und rechte Seite der Zuweisung bestimmt. Der Ausdruck `this.a` wird durch ein `BasedItem`, die lokale Variable `b` durch ein `LocalItem` charakterisiert. Bereits beim Generieren des `BasedItem`s wird Code erzeugt, der die Objektreferenz auf die aktuelle Instanz auf den Operandenstack legt. Nun wird Code generiert, der das `LocalItem` auf den Stack lädt und in diesem Fall auch noch für eine Typkonversion sorgt. Ausgehend von dem `AssignItem`, das man erhält, wird schließlich noch für die linke Seite der Zuweisung ein Speicherbefehl ausgegeben.

Unter den Items ist noch die Klasse `CondItem` erwähnenswert. Diese findet zum Übersetzen von bedingten Anweisungen Verwendung. Ein solches Item besitzt zwei *Label*-Listen, in denen jeweils unaufgelöste Sprunganweisungen, die typischerweise bei vorwärtsgerichteten Sprüngen auftreten, gesammelt werden. Die eine Liste enthält nur Sprünge, die bei erfüllten Bedingungen ausgeführt werden, die andere Liste enthält analog nur Sprünge, die bei unerfüllten Bedingungen stattfinden. Ist man bei der Codeerzeugung an der Stelle angekommen, auf die sich die Sprunganweisungen eines `CondItem`s beziehen, so können alle Sprünge in den beiden Listen nachträglich aufgelöst und das Sprungziel jeweils auf den aktuellen Programmzähler gesetzt werden. Dieser *Fixup* muß alle möglichen Sprunganweisungen der JVM unterstützen.

Im folgenden Listing ist für die Beispielklasse `XYZ` aus Abschnitt 9.3.3 der komplette, vom Compiler erzeugte Bytecode abgebildet. Der für die Anweisung „`this.a = b`“ erzeugte Code ist in der Methode `fn` ab Adresse 5 vorzufinden. Wie man sieht, hat der Compiler automatisch für die Klasse einen Konstruktor generiert, der lediglich den überschriebenen Konstruktor der Oberklasse `java.lang.Object` aufruft.

```
class XYZ extends java.lang.Object
{
    long a;
    void fn(byte);
    public XYZ();

  Method void fn(byte)
    0 iload_1                   /* Wert von x auf Stack laden    */
    1 bipush 16                 /* 16 auf Stack legen            */
    3 iadd                      /* Stackwerte addieren           */
    4 istore_2                  /* Ergebnis in b speichern       */
    5 aload_0                   /* this-Referenz auf Stack laden */
    6 iload_2                   /* Wert von b auf Stack legen    */
    7 i2l                       /* int -> long Konversion        */
    8 putfield #2 <Field XYZ.a> /* Wert in this.a ablegen        */
   11 return

  Method XYZ()
    0 aload_0
    1 invokeonvirtual #1 <Method java.lang.Object.<init>>
    4 return
```

```
  Local variables for method void fn(byte)
    byte x;                     /* pc = 0, length = 11, slot 1   */
    int x;                      /* pc = 0, length = 11, slot 2   */
}
```

9.6 Erweiterung: First-Class Functions

9.6.1 Verwendung der Spracherweiterung

Von den in der Einleitung erwähnten Erweiterungen von *Pizza* gegenüber Java, sind innerhalb von EspressoGrinder nur Funktionen höherer Ordnung implementiert. Durch die Compileroption „`-extended`“ kann man diese Erweiterung des Typsystems einschalten. Ist die Compileroption aktiviert, stehen Funktionen als Datentypen zur Verfügung. Mit folgender Syntax, dargestellt in EBNF, werden Funktionen konstruiert:

```
<lambda>    ::= fun <formals> <throwsDcl> <block>
<formals>   ::= "(" [<type> <ident> {"," <ident>}] ")"
<throwsDcl> ::= [throws <type> {"," <type>}]
```

Funktionsdefinitionen werten stets zu *Closures* aus. *Lexikalische Closures* sind prinzipiell Funktionen, in denen man sich alle Variablen als gebunden vorstellen kann. Variablen, die innerhalb der Funktionsdefinition frei vorkommen, werden zum Zeitpunkt der Generierung der Funktion, mit den korrespondierenden Variablen der Umgebung assoziiert, sodaß auch nach dem Beenden der Methode, innerhalb der die Funktionsdefinition stattfand, ein entsprechender Bezug zu den möglicherweise lokalen Variablen nicht verloren geht. Die Bindung der freien Variablen findet also zum Zeitpunkt der Funktionserzeugung statt, indem die aktuelle Umgebung als Auswertungskontext für die Funktion festgelegt wird. Dadurch wird sichergestellt, daß, egal an welcher Stelle im Programm ein Aufruf einer Closure auch erfolgt, diese stets im korrekten lexikalischen Kontext ausgewertet wird.
Funktionstypen zur Spezifikation der Signatur einer Closure werden gemäß folgender Syntax notiert:

```
<funtype>   ::= "(" [<type> {"," <type>}] <arrowtype> ")"
<arrowtype> ::= "->" <type> <throwsDcl>
```

Ein weiteres neues Sprachkonstrukt stellt ein *Sequenz*-Ausdruck dar. Es handelt sich dabei um einen normalen Block, mit dem Unterschied, daß für den Block ein Rückgabewert definiert wird. Es folgt die Syntax zur Notation von solchen Sequenz-Blöcken. Ein Rückgabewert kann darin, wie bei Funktionen und Methoden, mittels `return` zurückgegeben werden.

```
<seqexpr>   ::= seq <block>
```

Die folgenden Beispielklassen sollen die Verwendung von Closures demonstrieren. Die Methode `mkincr` generiert eine Funktion vom Typ `(int -> int)`, welche den Parameterwert um einen festen Wert `delta` erhöht und als Funktionswert zurückgibt. In der Methode `main` der Klasse `FooMain` werden dann verschiedene Closures generiert und aufgerufen. Wie man an dieser Klasse sieht, kann man eine Closure, ist sie einmal einer Variablen zugewiesen, genauso wie eine normale Methode der Klasse aufrufen.

```
{
    (int -> int) mkincr (int delta)
    {
        return fun (int x) { return x+delta; }
    }
}

public class FooMain
{
    public static void main (String[] argv)
    {
        Foo             f = new Foo();
        (int -> int)    inc1 = f.mkincr(1);     // zwei Closures generieren
        (int -> int)    inc3 = f.mkincr(3);
        System.out.println(inc1(inc3(10)));     // gibt 14 aus
    }
}
```

9.6.2 Übersetzung von Closures

EspressoGrinder erzeugt Bytecode für die Java Virtual Machine, welche speziell auf die Programmiersprache Java zugeschnitten ist. Außer Java selbst lassen sich deswegen, abgesehen von verwandten objektorientierten Programmiersprachen, wie beispielsweise *Smalltalk*, oder imperativen Sprachen wie *Ada*, funktionale Programmiersprachen eher schlecht in Java Bytecode übersetzen. Die Abbildung der *first-class functions* von EspressoGrinder auf Java Bytecode ist deswegen nur sehr umständlich über die Generierung neuer „Closure-Klassen“ möglich, die jeweils für eine Klasse alle Funktionstypen repräsentieren. Anhand der Klasse `Foo` aus dem Beispiel aus Abschnitt 9.6.1 wird im folgenden die Übersetzung der Closures erläutert [39].

Gewöhnlicherweise werden Closures durch einen Zeiger auf den Code der Funktion und eine Liste freier Variablen repräsentiert. Da es innerhalb einer Closure möglich ist, auch freie Variablen zu ändern, müßte man, wollte man eine Closure als eine gewöhnliche Methode implementieren, zusätzlich zu den aktuellen Parametern der Funktion, alle freien Variablen beim Aufruf per Referenz übergeben. Neben der Tatsache, daß es keine Code-Zeiger in Java gibt, tritt also auch das Problem des durch Java nicht unterstützen Referenzaufrufs auf.

EspressoGrinder modelliert Closures als Objekte der Klasse `espresso.lang.Closure`, welche neben einem entsprechenden Konstruktor nur eine Methode `$apply` enthält. Eine Closure wird in diesem Modell immer über eine Anwendung der `$apply`-Methode auf das die Closure implementierende Objekt ausgeführt. Code-Zeiger werden hiermit vermieden.

Um den Klassenkontextbezug einer Closure nicht zu verlieren, wird die eigentliche Implementierung einer Closure in Form einer Methode `$closure<`*n*`>` in der Klasse untergebracht, in der sich die Definition befindet. Um für die Closure-Methoden eindeutige Namen vergeben zu können, werden die in einer Klasse deklarierten Closures fortlaufend durchnumeriert. Variablen, die in einer Funktion frei vorkommen, werden in der entsprechenden Closure-Methode als zusätzliche Parameter übergeben. Freie Variablen, die im Definitionskontext der Closure lokal deklariert wurden, können innerhalb einer Closure neue Werte zugewiesen bekommen. Diese Variablen müssen über einen Referenzaufruf an die Closure-Methode übergeben werden. In EspressoGrinder wird diese Übergabe dadurch bewerkstelligt, daß die Werte der „Referenzvariablen“ durchgehend in einelementigen Arrays untergebracht werden, die dann beim Aufruf einer Closure-Methode anstelle der eigentlichen Variablen übergeben werden. Eine Änderung des Variablenwertes ist

kung auf den zugehörigen Auswertungskontext. Der Aufruf der Closure-Methoden erfolgt durch die `$apply`-Methode des die Closure implementierenden Objekts.

EspressoGrinder erzeugt für jede Klasse `XYZ`, in der mindestens eine Closure-Definition vorliegt, eine Unterklasse `XYZ$closure` der abstrakten Klasse `espresso.lang.Closure`, welche alle in der Klasse deklarierten Closure-Objekte – eine Funktionsdefinition wertet stets zu einem Closure-Objekt aus – repräsentiert. Diese Closure-Klasse sieht im allgemeinen folgendermaßen aus:

```
class XYZ$closure extends espresso.lang.Closure
{
    private int         $tag;           // Identifikator einer Closure-Definition in XYZ
    private Foo         $receiver;      // XYZ Objekt, in dem Closure konstruiert wurde
    private Object[]    $fvars;         // Liste freier Variablen in der Closure

    Foo$closure (Object[] freevars, Foo receiver, int tag) { ... }
    Object $apply (Object[] args) throws MessageNotUnderstood { ... }
}
```

Die Closures innerhalb einer Klasse werden vom Compiler durchnumeriert. Da eine Closure-Klasse stets zur Beschreibung aller Closure-Objekte, die in einer Klasse deklariert werden, herangezogen wird, kann über das Feld `$tag`, das effektiv der gewünschten Closure-Nummer entspricht, eine bestimmte Closure referenziert werden. Das Objekt, in dem eine Closure erzeugt wurde, wird im Attribut `$receiver` abgelegt. Innerhalb der Methode `$apply` wird abhängig von der Closure-Nummer `$tag` die passende Closure-Methode aufgerufen. `$receiver` gibt an, auf welches Objekt die von `$apply` aufzurufende Closure-Methode sich bezieht. Als weitere Auswertungskontextinformationen werden die Werte für freie Variablen bzw. oben erwähnte Pseudo-Referenzen, die beim Closure-Methodenaufruf zusammen mit den aktuellen Parametern übergeben werden müssen, in dem Feld `$fvars` gesichert. Die Darstellung erfolgt uniform über ein `Object`-Array. Bei der Übergabe werden die Werte jedoch in ihre natürliche, nicht-uniforme Repräsentation konvertiert. Die Konvertierung in die uniforme Darstellung als Objekt erfolgt mit den `espresso.lang.Closure.box`-Methoden. Für ein inverses *unbox* verwendet der Compiler die bereits in den Klassen für die einfachen Typen vorhanden Konvertierungsmethoden.

Wie der Compiler die Klasse `Foo` aus Abschnitt 9.6.1 übersetzt, kann dem folgenden Quelltext entnommen werden. Er enstand aus dem vom Compiler erzeugten Bytecode, der zurück in Java übersetzt wurde.

```
class Foo
{
    espresso.lang.Closure mkincr(int delta)
    {
        return new Foo$closure({espresso.lang.Closure.box(delta)},this,0);
    }

    int $closure0 (int $delta, int x)
    {
        return x + $delta;
    }
}
```

Wie man sieht, wurde der Funktionstyp durch eine Angabe der entsprechenden Closure-Klasse ersetzt. Eine Closure wird in der Methode `mkincr` durch einen Aufruf des Konstruktors der

Übersetzer erzeugten Closure-Methode `$closure0` vorzufinden. Die einzige ursprünglich frei vorkommende Variable ist `delta`. Sie muß zusätzlich zu den Parametern an die Closure-Methode übergeben werden.

Der folgende Quelltext zeigt die von EspressoGrinder für die Klasse `Foo` erzeugte Closure-Klasse `Foo$closure`. Der Compiler generiert wohlgemerkt direkt Java Bytecode für diese Klasse. Der abgedruckte Java Quelltext entstand wiederum durch Rückübersetzung aus dem Bytecode.

```
class Foo$closure extends espresso.lang.Closure
{
    private int         $tag;                 // Identifikator einer Closuredefinition in Foo
    private Foo         $receiver;            // Bezogenes Foo Objekt
    private Object[]    $fvars;               // Liste freier Variablen in Closure

    Foo$closure (Object[] freevars, Foo receiver, int tag)             // Closure Konstruktor
    {
        $fvars    = freevars;
        $receiver = receiver;
        $tag      = tag;
    }
                                              // Anwendung einer Closure auf die Argumente args
    Object $apply (Object[] args) throws MessageNotUnderstood
    {
        Object[]    allargs = combine($fvars,args);   // Alle freien Variablen und Parameter
                                                      // in gemeinsames Feld schreiben
        switch (tag)                                  // Fallunterscheidung fuer alle Closures
        {                                             // aus Klasse Foo
            case 0:
                return box($receiver.$closure0(       // $closure0 aus Foo Objekt $receiver auf-
                            allargs[0].intValue(),    // rufen. Die Parameter werden aus allargs
                            allargs[1].intValue())); // herausgenommen (unbox)
            default:
                throw new MessageNotUnderstood($tag); // Aufruf einer unbekannten Closure
        }
    }
}
```

Neben dem Konstruktor für `Foo`, der sozusagen eine neue Closure-Instanz anlegt, kümmert sich die Methode `$apply` um die Ausführung einer Closure. Die Funktionsargumente müssen in einem Feld übergeben werden, weshalb der Compiler jeden Closureaufruf in einen `$apply`-Aufruf mit vorangehender Erstellung eines Argumentfeldes transformiert. Da die `$apply`-Methode für die Ausführung aller in einer Klasse deklarierten Closures ausgelegt sein muß, ist darin abhängig von der Closure-Nummer `$tag` eine Fallunterscheidung für den Aufruf der zugehörigen Closure-Methode vorzufinden.

Würde man für jede Funktionsdefinition eine eigene, die Funktion repräsentierende Closure-Klasse generieren, so könnte zwar die Fallunterscheidung in `$apply` entfallen, man würde aber bei größeren Programmen auch entsprechend viele Klassendateien mit praktisch identischem Aufbau erzeugt bekommen, die bei der Programmausführung alle einzeln eingeladen werden müßten. Die Zusammenfassung aller Closure-Repräsentationen einer Klasse zu einer einzigen Closure-Klasse bringt also sicherlich keine Effizienzeinbuse mit sich, sondern erleichtert auf der anderen Seite sogar noch die Zuordnung von Closure-Klassen zu definierenden Klassen.

Abschließend werden noch einmal alle Transformationen aufgeführt, die EspressoGrinder beim Übersetzen von Funktionen höherer Ordnung in Closure-Klassen durchführen muß:

2. Deklarationen der Form

$$\texttt{f = fun}(p_1, \ldots, p_n)\ \texttt{\{...\}}$$

werden durch einen Aufruf des zugehörigen Closure-Konstruktors ersetzt:

$$\texttt{f = new XYZ\$closure(\{box}(v_1)\texttt{,...,box}(v_m)\texttt{\},this,}\ nr\texttt{)}$$

Um diese Anweisung generieren zu können, muß der Compiler eine Datenflußanalyse starten, um alle freie Variablen $v_1, \ldots, v_m$, die in der Funktion verwendet werden, herauszufinden. Außerdem muß die zugehörige Closure-Methode `$closure`$nr(a_1, \ldots, a_n)$, die die eigentliche Implementierung der Closure enthält, natürlich noch erzeugt werden. Diese Methode erweitert den abstrakten Syntaxbaum der Klasse um einen neuen `FunDef`-Knoten.

3. Closure-Aufrufe der Form

$$\texttt{x = f}(p_1, \ldots, p_n)$$

werden auf Aufrufe der `$apply`-Methode der zugehörigen Closure-Instanz zurückgeführt:

$$\texttt{x = f.\$apply(\{box}(p_1)\texttt{,...,box}(p_n)\texttt{\})}$$

4. Lokale Variablen oder Parameter, die in einer Closure-Definition frei vorkommen, müssen als „Referenzvariablen“ behandelt werden; d.h. es wird, statt direkt einen Variablenwert zu speichern, ein einelementiges Array angelegt, welches als einzigen Eintrag den Variablenwert besitzt. Bei Aufrufen einer Closure-Methoden durch `$apply` wird damit diese Variable automatisch via Referenzaufruf übergeben.

5. Für jede Klasse `XYZ`, in der Closures erzeugt werden, wird Bytecode für die Closure-Klasse `XYZ$closure` generiert

Seminarbeitrag von **Jörn Rehse**

Dieser Seminarbeitrag beschäftigt sich mit der Kombination von Java und CORBA, davon ausgehend, daß das Zielpublikum Kenntnisse von Java aber keine Kenntnisse von CORBA besitzt. Innerhalb der Einleitung wird kurz erläutert was CORBA grundsätzlich ist und wie es sich in die Object Management Architecture (OMA) einordnet. Danach folgt im zweiten Abschnitt die Erläuterung der CORBA Spezifikation. Im dritten Abschnitt wird dann auf die Kombinationsmöglichkeit von Java und CORBA eingegangen.

Die OMG. Die Object Management Group (OMG) ist eine non-profit Organisation, deren Zielsetzung es ist die „Theorie und Praxis der objektorientierten Technologie in der Software Entwicklung [8] zu unterstützen. Die OMG besteht aus über 500 Mitgliedern, zum größten Teil Firmen, wie Sun, Digital Equipment oder Hewlett-Packard, um nur drei der prominentesten zu nennen. Die OMG bemüht sich um die Schaffung eines Industriestandards für die Benutzung von Netzwerkobjekten.

Die OMA. Das erste Produkt einer solchen Standardisierungsbemühung ist die Object Management Architecture (OMA). Die OMA besteht aus vier Teilen, den Object Request Broker, die Object Services, die Common Facilities und die Application Objects. Die Common Request Broker Architecture (CORBA) definiert nun einen Standard für den Object Request Broker Teil der OMA. Was das ist, soll im nächsten Abschnitt erläutert werden.

10.1 Erläuterung der CORBA Spezifikation

10.1.1 Das Objektmodell

Vor der weiteren Erläuterung des CORBA Standards muß das CORBA zugrunde liegende Verständnis davon, was ein Objekt überhaupt ist und was es tut, erläutert werden. Diese Auffassung von Objekten heißt das CORBA Objektmodell. Es definiert die Terminologie mit der CORBA arbeitet.

Das CORBA Objektmodell ist ein „klassisches Objektmodell [8], das heißt, es entspricht der Vorstellung von Objekten, die auch in den meisten Programmiersprachen und objektorientierten Architekturansätzen propagiert wird.

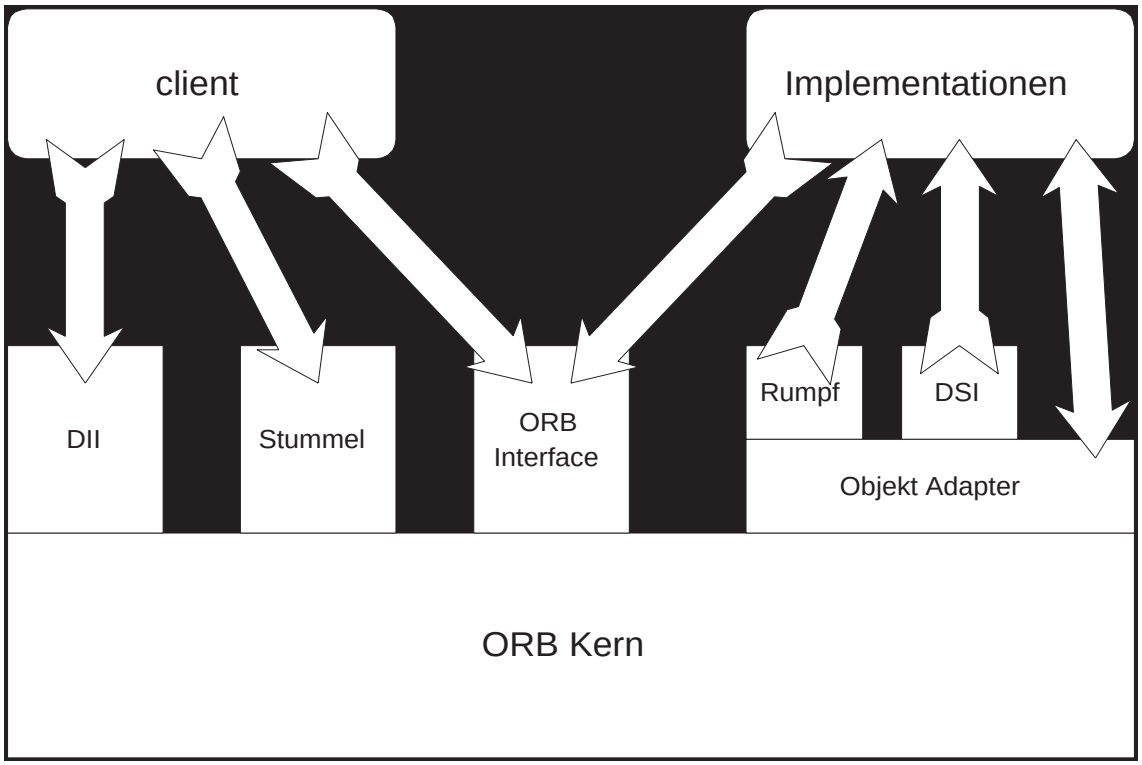
Der grundsätzliche Vorgang, der von dem Objektmodell unterstützt wird, ist, daß ein Client eine Nachricht (Request) an ein Objekt schickt. Das Objekt interpretiert die Nachricht und entscheidet, welcher Dienst ausgeführt werden soll. In der CORBA Lesart sind Objekte grundsätzlich

zuführenden Dienstes und eventuell benötigte Parameter.

Ein Objektsystem ist eine Menge von Objekten. Entsprechend dem Prinzip der Datenkapselung sind die Dienstnehmer von den Diensterbringern des Objektsystems durch eine genau definierte Schnittstelle getrennt. (Zur Erläuterung: Die Clients sind isoliert von der Implementierung der Dienste, sei es die Repräsentation von Daten oder Code). Insbesondere gibt es aus der Sicht des Clients auch keine besonderen Mechanismen zum Erzeugen oder Zerstören eines Objekts. Für ihn entstehen und vergehen die Objekte als Ergebnisse von Requests. Ein Client erfährt von einem neuentstandenen Objekt durch den Erhalt einer Objektreferenz.

Eine Schnittstelle ist eine (mit IDL spezifizierte) Beschreibung der Operationen, die ein Client von einem Objekt anfordern kann. Ein Objekt erfüllt (satisfies an interface) eine Schnittstelle genau dann, wenn es als Zielobjekt jeder möglichen in der Schnittstelle beschriebenen Anforderung (Request) genutzt werden kann. Aus der Schnittstellenbeschreibung geht hervor, von welchem Datentyp die Parameter der Operation sind. Parameter sind also Exemplare (Instances) eines Datentyps. Parameter, die ein Objekt bezeichnen, heißen Objektnamen (Object Names). Parameter, die ein eindeutiges Objekt bezeichnen, heißen Objektreferenzen (Object Reference). Um einen Dienst von einem Objekt anfordern zu können, muß der Client eine Objektreferenz für das Objekt besitzen. Eine Objektreferenz erlaubt das eindeutige Identifizieren eines Objekts. Operationen sind insofern generisch, als daß die gleiche Operation (beschrieben durch eine Schnittstelle) durchaus verschieden implementiert sein kann. Es ist durchaus möglich, daß verschiedene Implementierungen völlig unterschiedliches Verhalten zeigen.

Eine Implementierung eines Objekts besteht aus Code, der die vom Objekt bereitgestellten Dienste erbringt (Methode), und Daten.



Der Zweck eines ORBs ist es, die Requests der Clients zu einer Implementierung des Objektes, das den gewünschten Dienst anbietet, zu bringen und die Ergebnisse wiederum an den Client zurück zu liefern. Dazu muß der ORB in der Lage sein, den Code und die Daten, die die Implementierung des Objektes sind, zu finden und auf die Entgegenahme des Requests vorzubereiten. Außerdem muß der ORB den Datentransport zwischen Client und Objekt bewerkstelligen. Durch den ORB wird die räumliche Position der Objekte und die Sprache ihrer Implementierung vor dem Client versteckt. Der ORB wird in der Literatur oftmals als ein „Objekt-Bus“oder „Software-Bus“ bezeichnet.

Nachdem ein Client eine Objektreferenz erhalten hat, kann er Requests an dieses Objekt richten. Dies geschieht entweder durch den Aufruf von Stummel-Routinen (stub-routines) oder dynamisch über die Dynamische Aufrufschnittstelle DII (Dynamic Invocation Interface). Durch die Stubs oder die DII gelangt der Request in den Kern des ORB (ORB Core). Allerdings hat das Objekt keine direkte, durch den ORB zur Verfügung gestellte Möglichkeit, festzustellen, woher der Request kam. Sicherheitsmechanismen müssen also extra realisiert werden, falls der ORB keine Authentifikationsdienste unterstützt.

Nachdem der ORB nun die Implementierung des Objektes gefunden hat, leitet er den Request durch einen Objektadapter (Object Adapter) und durch den darauf sitzenden/dazugehörigen IDL Rumpf (IDL Skeleton), einer Art Gegenstück zu den Client-seitigen Stubs, an das Objekt weiter.

Die Sichtweise des ORB als einförmiges Gebilde bestehend aus dem Kern, den Stubs, der DII, den Adaptern und den Skeletten ist eine Modellvorstellung, die in konkreten CORBA Implementierungen nicht eingehalten werden braucht. Wichtig ist nur, daß die ORB Implementierung die im CORBA Standard definierten Schnittstellen zur Verfügung stellt. Der ORB Core ist der Teil des ORBs, der die grundlegenden Objekt-Handhabungsmechanismen zur Verfügung stellt. CORBA ist so entworfen worden, daß verschiedene Implementierung von ORBs die Objekte unterschiedlich handhaben können. Dazu wurde der ORB unterteilt in den Kern und die Komponenten oberhalb des Kerns, die die Besonderheiten der Implementierung verdecken.

Es gibt viele verschiedene Arten, eine Implementierung eines ORB in einem System unterzubringen. Günstig für die Leistung, Sicherheit und Stabilität ist es sicherlich, den ORB als Dienst eines Betriebssystems anzubieten (System-Based Orb) [8]. Es könnten auch ein oder mehrere ORBs zentral auf einem oder auf mehreren Servern installiert sein (Server-Based ORB). Falls ein geeigneter Kommunikationsweg zwischen Clients und den Implementierungen der Objekte vorhanden ist, kann der ORB auch aus Routinen in den Clients und den Implementierungen der Objekte bestehen (Client-and-Implementation-Resident-ORB).

10.1.3 Statische Objektaufrufe

Stummelroutinen und IDL. Gegeben sei die Situation, in der ein Client im Besitz einer Objektreferenz ist und somit Dienste anfordern kann. Wie bereits erwähnt, ist das einzige Wissen, das ein Client über einen Dienst haben kann, die zugehörige Schnittstelle. Um also Dienste von einem Objekt anfordern zu können, muß der Client wissen, welche Dienste das Objekt anbietet. Aus der Sicht des Clients heißt dies, er muß wissen, wie die Schnittstellen der Dienste des Objekts aussehen. Eine komplette Beschreibung der Schnittstelle ist alles, was nötig ist (neben der Objektreferenz natürlich), um einen Dienst von einem Objekt anzufordern. Für diese Beschreibung wird in CORBA die objektorientierte Schnittstellen-Definitionssprache OMG IDL (Interface

synonym. Es sei aber darauf hingewiesen, daß es noch andere Schnittstellen-Definitionssprachen gibt, wie etwa ASN.1 oder Microsoft's ODL. Eine Schnittstellenbeschreibung in IDL definiert eine Schnittstelle vollständig und spezifiziert alle ihre Parameter. IDL ist eine Beschreibungssprache und enthält deshalb keinerlei Unterstützung für algorithmische Konstrukte oder Variablen. Da CORBA sprach- und systemunabhängig ist und insbesondere die Zusammenarbeit von Clients und Objekten, die in unterschiedlichen Sprachen implementiert worden sind, erlauben soll, muß ein Programm, der IDL Compiler, die IDL Definition auf Konstrukte in der benutzten Programmiersprache abbilden (language mapping). Der IDL Compiler erzeugt Codier- und Decodierfunktionen für die Client Stubs und das Objekt Skeleton in der Sprache, in der der Client bzw. das Objekt implementiert wurde. Diese Funktionen formen die Schnittstellenparameter in eine übertragbare Datenrepräsentation um bzw. rekonstruieren aus der Übertragungsrepräsentation die Parameter (marshalling und demarshalling). Eine Abbildungsvorschrift (das language mapping) legt für eine Programmiersprache fest, welche IDL Elemente in welche Sprachelemente der Zielsprache umgesetzt werden. Die erste Programmierpsrache für die eine OMG IDL Sprachabbildung definiert wurde war C. Bald folgten C++ und Smalltalk. Interessant ist in diesem Zusammenhang natürlich die IDL nach Java Abbildung, die im zweiten Abschnitt näher erläutert wird.

Die Benutzung der deskriptiven Sprache IDL zur Beschreibung von Schnittstellen ist ein Architekturkonzept von CORBA, das die strikte Trennung von Schnittstelle und Implementierung durch seine bloße Existenz unterstützt.

Sprachbeschreibung der Beschreibungssprache. Im folgenden werden einige der grundlegenden Sprachelemente von OMG IDL beschrieben, dabei sollte man sich stets vor Augen halten, daß IDL Code eine Definition von Dingen ist und kein Ablaufplan, der von einer Maschine abgearbeitet wird. Dinge, die in der folgenden Beschreibung von IDL erwähnt werden, sind also in erster Linie Dinge, für deren Definition (oder vielmehr für die Definition von Exemplaren davon) IDL syntaktische Mittel zur Verfügung stellt. IDL ist sehr C++ ähnlich. Genauer gesagt ist es eine erweiterte Teilmenge von C++. In [8] wird die Erwartung geäußert, daß die IDL Spezifikation der ANSI C++ Spezifikation in ihrer Entwicklung folgen wird.[1]

IDL stellt grundlegende Datentypen (z.B. float, long oder char), zusammengesetzte Datentypen (z.B. struct oder union) und generische Datentypen (templates wie z.B. sequence oder string) zur Verfügung. Jeder IDL Datentyp wird auf Datentypen in der Zielsprache abgebildet.

Exceptions sind Datenstrukturen (ähnlich „struct“ in C), die als Rückgabewerte bei Requests Ausnahmezustände anzeigen und näher beschreiben. Zu einer Schnittstelle eines Objekts gehören nicht nur die Typen der Parameter und Rückgabewerte einzelner Methoden, sondern auch das Verhalten bei Fehlern. In solchen Fällen liefert ein Objekt eine Meldung über die Art des Fehlers in einer Exception.

Operationsdeklarationen in IDL sind C Funktionsdeklarationen ähnlich, sie enthalten unter anderem den Namen der Operation und die Typen der Parameter und des Rückgabewertes. Zusätzlich kann angegeben werden, welche Exception im Mißerfolgsfall ausgelöst wird (raises expression) und auf welchen Teil des Client Kontext der Diensterbringer zugreifen darf (context expression).

Eine Interfacedeklaration besteht nun aus Typ-, Exception- und Operationsdeklarationen. Interface ist der wichtigste der IDL Typen, da er CORBA Objekte beschreibt, in dem er die

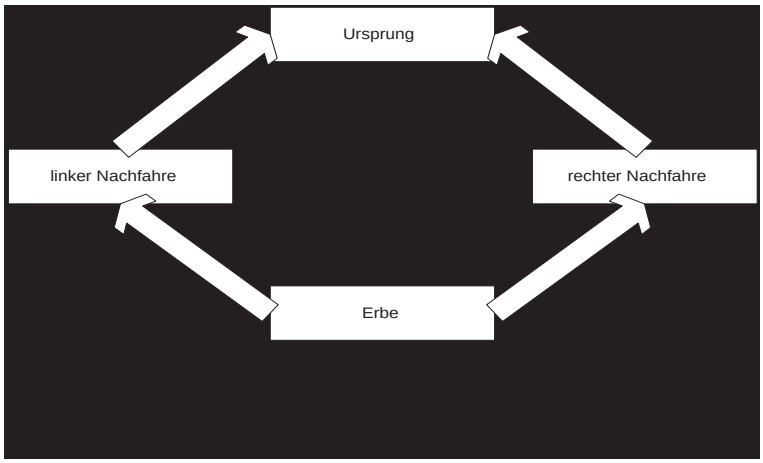
[1] „The IDL specification is expected to track relevant changes to C++ introduced by the ANSI standardization effort.“, [8], S. 46.

jekt sichtbar ist.[2] In einer Operation kann ein Objekt als Parameter oder Rückgabewert definiert werden, einfach indem der Name (Identifier) des Interfaces als Datentyp verwendet wird. Zusätzlich zu den bereits erwähnten Operationen können für ein Interface auch sogenannte Attribute deklariert werden. Hierbei handelt es sich um Werte, für die automatisch Zugangsfunktionen (Accessor Functions) zur Verfügung gestellt werden. Die Angabe eines Attributs ist äquivalent zur Deklaration von Funktionen zum Schreiben und Lesen eines Wertes.

Eines der grundlegensten Sprachprinzipien von OMG IDL ist die Vererbung. Teil des Kopfes (Header) einer IDL Sprachdefinition ist die Vereerbungsspezifikation (Inheritance Specification), die festlegt von welchem anderen Interface das neue Interface erben soll. Das vererbende Interface heißt Base Interface, das erbende Interface wird abgeleitetes (derived) Interface genannt. Die Vererbung ist transitiv. Ein Base Interface, das explizit in der Inheritance Specification genannt wird, heißt „direct base“ des deklarierten Interfaces. Ein Interface, das nicht explizit genannt wird, sondern Base Interface eines der in der Inheritance Specification genannten Interfaces ist, heißt „indirect base.“ Die Elemente eines Base Interfaces können so benutzt werden, als seien sie Elemente des abgeleiteten Interfaces. Um explizit auf Elemente eines Base Interfaces zuzugreifen steht der Auflösungsoperator (Name Resolution Operator) „::“ zur Verfügung. Dadurch ist es auch möglich, auf definierte Bezeichner aus Base Interfaces zuzugreifen, die im abgeleiteten Interface neu definiert wurden. Ein abgeleitetes Interface darf Bezeichner für Typen, Konstanten oder Exceptions, die geerbt worden sind, umdefinieren. Operationen dürfen allerdings nicht neu definiert werden. Ein Interface kann von beliebig vielen Base Interfaces abgeleitet sein (multiple inheritance), wobei die Reihenfolge keine Rolle spielt. Allerdings darf ein Interface höchstens einmal als direct Base eines abgeleiteten Interfaces angegeben werden. Als indirect Base darf es unbegrenzt oft auftauchen.

Unbedingt sollte man sich im klaren sein, daß eine Schnittstelle, die von einer anderen Schnittstelle erbt, nur deren Schnittstellenbeschreibung erbt, aber keinsfalls die Implementierung. In IDL gibt es kein Konzept von Implementierung oder Zustand eines Objektes.

Jeder IDL File bildet einen „Bennenungsbereich“ (Naming Scope). Interfacedefinitionen, Operationsdefinitionen und einige andere Definitionsarten bilden wiederum verschachtelte Benennungsbereiche. Ein Bezeichner kann in einem Bereich jeweils nur einmal definiert werden. In verschachtelten Bereichen jedoch können Bezeichner umdefiniert werden. IDL stellt ein Konstrukt namens Module zur Verfügung, das benutzt werden kann, um die Benennungsbereiche zusätzlich zu strukturieren.



[2] In diesem Sinne ist Interface auch der Typ der Objektreferenzen.

10.1.4 Das Interface Repository

Das Schnittstellenlager (Interface Repository) ermöglicht die dauerhafte Lagerung von IDL Schnittstellendefinitionen. Schnittstellendefinitionen werden im Schnittstellenlager als Objekte untergebracht. Zusätzlich enthält das Lager auch noch Typdefinitionen in der Form von sogenannten Typecodes. Aus einem Typecode ist die vollständige Struktur eines Datentyps erkennbar.

Insgesamt ist das Schnittstellenlager als eine Menge von Objekten organisiert. Auf diesen Objekten sind Operationen definiert, mit denen unter anderem ein Client auf Schnittstellendefinitionen zugreifen kann. Besitzt ein Client eine Objektreferenz auf ein Objekt, so kann er auf diese Referenz die Operation get_interface() anwenden. Diese Operation liefert eine Referenz auf ein Objekt im Interface Repository (nämlich ein InterfaceDef Objekt), das die Schnittstelle des Objektes beschreibt, auf das get_interface() angewendet wurde.

Die Dienste, die das Schnittstellenlager anbietet, liefern also unter anderem eine vollständige Beschreibung aller Operationen, die auf ein Objekt anwendbar sind, und erlauben es, die komplette Vererbungshierarchie von Objekten zu „durchwandern."

Von den Clients kann das Schnittstellenlager zu verschiedenen Zwecken verwendet werden. Ein Interface Browser könnte Entwicklern zum Beispiel erlauben, wiederverwendbare Softwarekomponenten aufzuspüren. Auch andere Arten von Objektbrowsern, die es erlauben, immer wieder neue Dienste zu einem System hinzuzufügen, sind denkbar. Die Hauptaufgabe des Schnittstellenlagers ist allerdings die Datentypinformation für die Dynamische Aufrufschnittstelle (Dynamic Invocation Interface DDI) zur Verfügung zu stellen.

Das Implementierungslager ist mit dem Schnittstellenlager verwandt. In ihm verwaltet der ORB Informationen über die ihm bekannten Implementierungen von Objekten. Diese Informationen sind unter anderem notwendig, um die Implementierungen zu lokalisieren.

10.1.5 Das Dynamic Invocation Interface (DII)

Zusätzlich zu den statischen Stummelroutinen, die durch einmaliges Übersetzen von IDL Definitionen erzeugt werden, ist oftmals noch ein weiterer Zugang für Objektaufrufe durch den Client wünschenswert und zwar ein solcher Zugang, der Clients erlaubt Objekte aufzurufen, deren Schnittstellen sie beim Übersetzen noch nicht kennen. Eine solche Möglichkeit bietet die Dynamische Aufrufschnittstelle DII. Prinzipiell ist die DII ein generischer Stummel auf der Client Seite des ORB, der beliebige Aufrufe (über den ORB) an beliebige Objekte weiterleiten kann. Um die Schnittstellendaten des gewünschten Zielobjektes herauszufinden, kann der Client nun das Schnittstellenlager benutzen. Wichtig ist festzustellen, daß es keinen semantischen Unterschied zwischen einem Aufruf, der das DII benutzt, und einem Aufruf, der durch einen statischen Stummel in den ORB eintritt, gibt. Das aufgerufenen Objekt kann beide Wege nicht unterscheiden. Abschließend bleibt zur DII zu sagen, daß ihre Benutzung meistens nicht so effizient ist, wie die einer genau auf einen Schnittstellentyp zugeschnittenen Stummelroutine.

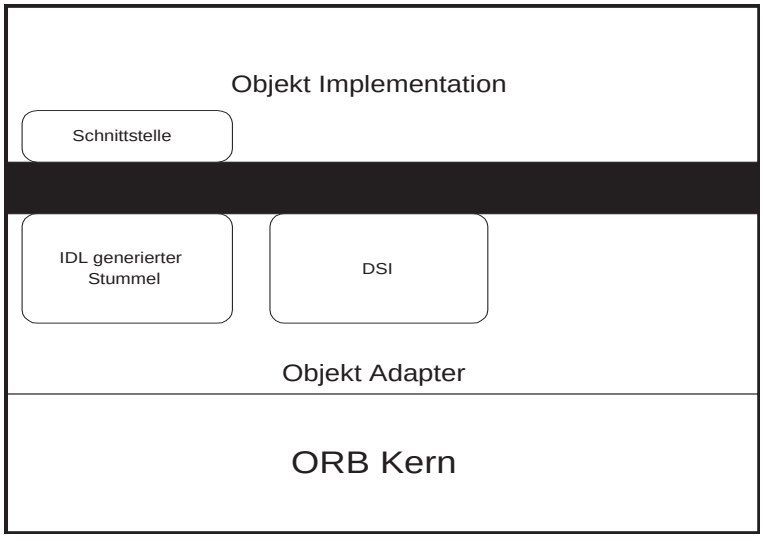
10.1.6 Das Dynamic Skeleton Interface (DSI)

Die Dynamische Rumpfschnittstelle DSI ist das Analogon zur DII auf der Server-Seite des ORB. Durch sie ist es möglich, Anfragen vom ORB an Objektimplementierungen weiterzuleiten, die

Situation entspricht also genau der Situation der DII, nur daß die unwissenden Teilnehmer diesmal nicht die Clients sondern die Implementierungen sind. Auch sind die Wege, die eine Anfrage nehmen kann, wieder semantisch gleich. Ein Client, der ein Objekt aufruft, kann nicht entscheiden, ob die Implementierungen über einen statischen Rumpf oder die DSI mit dem ORB kommuniziert.

10.1.7 Objektadapter

Zusätzlich zu den bisher erwähnten Transportfunktionen für Anfragen stellt ein ORB auch noch verschiedene Dienste (ORB Services) zur Verfügung. Bei ORB Diensten handelt es sich neben der eindeutigen Benennung von Objekten im Netzwerk, dem Methodenaufruf, der Ausnahmebehandlung, der Generierung und der Interpretation von Objektreferenzen auch um Dienste, die die eindeutige Identifikation und Authentifikation von Kommunikationspartnern gewährleisten. Es wird also eine Möglichkeit benötigt, mit der eine Objektimplementierung auf die ORB Dienste zugreifen kann. Dies wird durch Objektadapter (OA) bewerkstelligt. Je nach ORB Implementierung erbringt der Objektadapter die Dienstleistungen selbst oder delegiert sie an andere Teile des ORB. In jedem Fall können die Objektimplementierungen den Unterschied nicht feststellen, da sie nur über eine Schnittstelle an den Adapter gebunden sind und seine Implementierung vor ihnen versteckt ist [60]. Objektadapter sind allerdings nicht nur für den Zugriff der ObjektImplementierungen auf ORB Dienste zuständig, sondern auch die statischen und dynamischen Rümpfe sind direkt an die Objektadapter angebunden, die somit nahezu die gesamte Kommunikation zwischen ORB und ObjektImplementierung übernehmen (bis auf die Kommunikation durch die Objektschnittstelle). Der OA ist dafür verantwortlich, den ORB mit den Implementierungen zu verknüpfen. Ein ORB kann mehrere Objektadapter zur Verfügung stellen, die auf bestimmte Gruppen von Objekten zugeschnitten sind. Es wird aber allgemein davon ausgegangen, daß nur wenige Arten von Objektadaptern benötigt werden. In der CORBA Spezifikation wird nur ein Objektadapter BOA (Basic Object Adapter) definiert, der für die Zusammenarbeit mit einer großen Anzahl verschiedener ObjektImplementierungen ausreichen dürfte. Wie bereits erwähnt, ist in der CORBA Spezifikation offen gelassen, ob ein Objekt als zusammenhängendes Programm implementiert ist oder ob es aus mehreren Teilen besteht. Konsequenterweise unterstüzt der BOA ObjektImplementierungen unabhängig davon, aus wievielen Programmteilen, Skripten oder Modulen sie bestehen.



Ein typischer Interaktionsablauf zwischen dem BOA und einer ObjektImplementierung sieht wie folgt aus: Als erstes startet der BOA die Objektimplementierung (Implementation activation), da jedes Objekt, das auf Anfragen reagieren soll, auch tatsächlich durch ein laufendes Programm realisiert werden muß. Ein solches Programm wird in der CORBA Spezifikation als Server bezeichnet. Es gibt vier verschiedene Arten von Verhaltensregeln, denen ein Server folgen kann (activation policies). In diesem Beispiel soll von einem Server ausgegangen werden, der als einzelne Implementierung viele Objekte zur Verfügung stellen kann (shared server policy). Andere Varianten sind, daß nur ein Objekt auf einem Server aktiv sein kann (unshared server policy), daß jede Methode eines Objektes von einem Server implementiert wird (server-per-method policy) oder das Aktivieren eines Servers durch Komponenten außerhalb des BOA (persistent server policy). Als nächstes unterrichtet der Server den BOA davon, daß er seine Initialisierungsphase abgeschlossen hat und bereit ist, auf Anfragen einzugehen. Sobald die erste Anfrage für ein Objekt eingeht, befiehlt der BOA dem Server das Objekt zu aktivieren. Von jetzt ruft der BOA beim Eintreffen von Anfragen über die Rumpfschnittstellen die entsprechenden Objektmethoden auf. Die Implementierung kann wiederum die vom BOA angebotenen Dienste, wie z.B. das Erzeugen weiterer Objekte, benutzen.

Theoretisch könnte es unendlich viele Objektadapter geben, da aber die Objektimplementierungen auf die Adapter angewiesen sind, sollte es möglichst wenig verschiedene Objektadapter geben, die Objektadapter sollten also möglichst allgemein gehalten werden. Außer dem BOA werden in der CORBA Spezifikation noch ein Objektadapter, der für Objekte, deren Implementierung eine Programmbibliothek ist, gedacht ist (library object adapter) und ein Objektadapter für den Zugang zu Objekten in einer objekt-orientierten Datenbank (object-oriented database adapter) erwähnt.

10.1.8 Die ORB Schnittstelle (ORB Interface)

Die ORB Schnittstelle führt direkt in den ORB. Es ist die Schnittstelle, durch die jene Operationen auf Objekte aufgerufen werden können, die auf allen Objekten definiert sind, und durch die jene ORB Dienste, die in jeder ORB Implementierung die selben sind, genutzt werden können.

10.1.9 ORB Interoperability Architecture

Jeder ORB muß eine Standardrepräsentation für zu übertragende Daten definieren (on-the-wire format, flat data representation). Es wäre wünschenswert, wenn alle ORB Implementierungen das selbe Format benutzten, aber die Entwicklung hat sich anders abgespielt. Deshalb entstand ein großer Bedarf nach einem in der CORBA Spezifikation definierten Standard, der die „Fähigkeit zur Zusammenarbeit“ (Interoperability) von ORBs erhöht. Während in Version 1.0 der CORBA Spezifikation ganze drei Seiten dem Thema gewidmet sind, sind es in CORBA 2.0 fünf Kapitel. Die Zusammenarbeit zwischen ORBs ist aber erstrebenswert, da so für den Client eines ORBs eine ganze Palette aufrufbarer ObjektImplementierungen zur Verfügung steht. Die ORB Interoperability Architecture kann als die grundlegendste Neuerung in CORBA 2.0 angesehen werden. Sie ist der erste große Einfluß tatsächlicher ORB Entwicklungen auf CORBA.

Auch wurde darauf Wert gelegt, ORB spezifische Details vor den Applikationen zu verstecken. Eine CORBA Applikation braucht (kann) keine Kenntnisse davon haben, wie sich die ORBs unterhalten.

kennt, kommunizieren kann, werden in der Spezifikation der ORB Interoperability Architecture sogenannte Brücken (bridges) definiert. Dabei handelt es sich entweder um Brücken, die direkt von einem ORB zum anderen führen (full bridges) oder um Brücken, die von den ORBs zu einem gemeinsam benutzten Medium führen (half bridges). Per Definition sind Brücken nichts anderes als Implementierungen von Abbildungsregeln, die die Verhaltensrweisen von Adreßräumen (domains) aufeinander abbilden. Es kann allerdings vorkommen, daß nicht alle Dienste, die ein ORB anbietet auf einen anderen ORB abgebildet werden können. Brücken können entweder direkt im Inneren des ORBs implementiert (in-line bridges) oder sich aber wie andere Applikationsobjekte außerhalb des ORBs befinden und über die DSI oder die statischen Rümpfe angesprochen werden (request-level bridges).

GIOP, IIOP und ESIOP. Die grundlegende Definition einer Standard-Transfersyntax (Datenrepräsentation auf niedriger Ebene) ist das GIOP (General Inter-ORB Protocol). Das GIOP kann auf jedem beliebigen verbindungsorientierten Transportprotokoll aufbauen, das eine kleine Menge von Bedingungen erfüllt [8]. ORBs die GIOPs benutzen, die nicht auf dem selben Transportprotokoll aufbauen, können nicht direkt über ihre GIOPs kommunizieren. Da sie aber beide der selben GIOP Definition folgen, können sie leicht aufeinander abgebildet werden. Das IIOP (Internet Inter-ORB Protocol) ist eine konkrete Ausprägung der abstrakten GIOP Definition. Das Transportprotokoll, auf dem das IIOP basiert ist das TCP/IP (Transfer Control Protocol/Internet Protocol). Es ermöglicht CORBA ORBs, miteinander über das Internet zu kommunizieren, ist also ein Schlüssel zu verteilten Objekten im Internet mit CORBA (auch ohne Java). In seiner Rolle im Internet ist das IIOP mit dem HTTP oder dem FTP vergleichbar. Damit ORBs auch mit bestehenden Rechnerinfrastrukturen kommunizieren können, läßt die Interoperability Architecture Raum für umgebungsspezifische Inter-ORB Protokolle, sogenannten ESIOPs (Environment specific Inter-ORB Protocols).

Im Frühjahr 1996 gab es, bis auf Black Widow, keine ORBs, die die ORB Interoperability Spezifikation von CORBA 2.0 erfüllten.

10.1.10 Zusammenfassung

CORBA ist die Spezifikation eines Standards für ORBs. ORBs erlauben objektorientierten Applikationen, auf einer höheren Abstraktionsebene als dem Versand von Datenpaketen miteinander zu kommunizieren. CORBA konforme ORBs sollen in heterogenen, verteilten Umgebungen Clients ermöglichen, Objekte zu lokalisieren und Anfragen an sie zu richten. Die Schnittstellen, die den Zugang zum ORB erlauben, sind mit Hilfe der Schnittstellenbeschreibungssprache IDL definiert. Zusätzlich ist es möglich, über dynamische Schnittstellen auf Objekte zuzugreifen, deren Schnittstellenspezifikation zur Zeit der Übersetzung der Client-Applikation noch nicht bekannt war. Durch die ORB Interoperability Architecture sollen verschiedene ORB Implementierungen besser zusammenarbeiten können.

10.2 Verteilte Objekte im Inter-/Intranet mit Java und CORBA

Wie nun lassen sich die Möglichkeiten zur Erstellung von verteilten, objektorientierten Anwendungen von Java und CORBA kombinieren?

Einige Konzepte von Java und CORBA scheinen sich auf den ersten Blick zu widersprechen. Am deutlichsten wird der Unterschied bei der Frage danach, wo ein Objekt sich während seiner

am selben Ort, und zwar auf dem Rechner des Dienstnutzers. Ein Applet enthält den Code, der ein Objekt implementiert. Dieser Code bewegt sich durch das Netz. Das Objekt selbst wird aber erst im Rechner, auf dem das Applet zur Ausführung kommt, instanziiert. Von diesem Rechner wird auch der Zustand des Objekts verwaltet. Code und Zustand, also das Objekt, befinden sich auf einem Rechner und werden sich auch nicht von dort wegbewegen. Java stellt keine Möglichkeit zur Verfügung, um auf Objekte zuzugreifen, die bereits irgendwo existieren, was CORBA aber erlaubt. Nach der Auffassung von CORBA bleibt ein Objekt ebenfalls immer am selben Ort, allerdings nicht unbedingt auf dem Rechner, auf dem das Clientprogramm läuft, sondern irgendwo auf einem Server. Es bewegen sich nur Objektreferenzen. Auf die Frage, wo sich ein Objekt befinden sollte, geben CORBA und Java also grundverschiedene Antworten.

Wie also lassen sich diese Ansichten verbinden?

Da Java eine Programmiersprache ist und ein CORBA ORB sprachunabhängig sein soll, ist es möglich, Java in das CORBA Konzept völlig dem CORBA Paradigma entsprechend einzubinden. Man behandelt in diesem Fall Java genauso wie alle anderen Programmiersprachen, die mit CORBA ORBs zusammenarbeiten. Genau wie für C++ oder Smalltalk wird „einfach“ eine IDL nach Java Sprachabbildung definiert, und schon können Clients und Serverobjekte in Java geschrieben werden. Hierbei gingen allerdings alle jene Aspekte von Java, die die Sprache für den Netzbetrieb so geeignet machen, verloren. Besonders das Prinzip des mobilen Codes fände überhaupt keine Anwendung mehr.

Eine deutlich interessantere Kombination wird von den Herstellern der CORBA ORBs, die mit Java Programmen zusammenarbeiten können, tatsächlich verfolgt. Die Fähigkeiten von Java und CORBA lassen sich zur Realisierung von verteilten Objekten im World Wide Web zusammenbringen. Bei diesem Ansatz wird ein Java Applet, das alle jene Komponenten eines CORBA ORB (bzw. einen vollständigen ORB) implementiert, die auf der Client-Seite gebraucht werden, in einen Java fähigen Browser geladen. Das Applet enthält also die Client-Applikation und alle für die Kommunikation nötigen Komponenten. Die Übertragung des Applets ist allerdings auch der einzige Moment, in dem Javas mobiler Code zum Einsatz kommt. Der danach folgende Ablauf entspricht wieder ganz dem CORBA Modell. Das Applet kann nun auf CORBA Objekte zugreifen, die irgendwo im vom ORB erreichbaren Teil des Netzes liegen. Mit Java können also über das Internet CORBA Objekte auf jedem Rechner, auf dem ein Java fähiger Browser läuft, benutzt werden, ohne daß irgendwelche CORBA konformen Komponenten installiert sind.

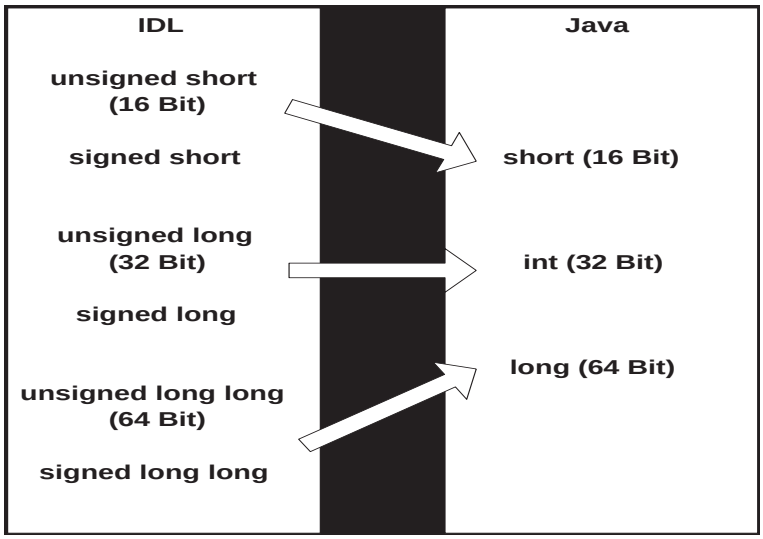
Ob sich im Java Applet nur ein kleiner Teil der ORB Funktionalität befindet, nämlich der Teil zur Kommunikation, oder ob ein vollständiger ORB im Applet enthalten ist, hängt von der jeweiligen Implementierung ab.

10.2.1 Die Sprachabbildung von IDL nach Java

Jeder Anbieter eines ORB, der auch mit Java zusammenarbeiten soll, muß einen IDL nach Java Überstetzer zur Verfügung stellen. Ein solcher Übersetzer basiert auf einer Abbildungsvorschrift der einen Sprache in die andere (language mapping). Die hier behandelte Sprachabbildung wurde im Februar 1996 von JavaSoft vorgestellt. Es handelt sich nicht um eine von der OMG offiziell in den CORBA Standard aufgenommene Abbildung. Bis jetzt wurde noch keine IDL nach Java Abbildung von der OMG in die CORBA Spezifikation aufgenommen. Die Abbildungsvorschriften der einzelnen Hersteller unterscheiden sich aber nur in geringen Teilen. Trotzdem muß darauf hingewiesen werden, daß bis zum Erscheinen einer von der OMG offiziell abgesegneten Übersetzungsvorschrift Java Stummel oder Rümpfe nur mit ORBs des Herstellers, der auch den Übersetzer mit dem sie übersetzt wurden entwickelt hat, zusammenarbeiten können.

gegeben werden, wie IDL auf Java abgebildet wird. Besonders interessant ist dabei natürlich, wie jene IDL Sprachkonstrukte, für die Java keine offensichtlich gleichbedeutende (isomorphe) Konstrukte zur Verfügung stellt, abgebildet werden.[3]

Modules werden zu Packages. Ein IDL Module deklariert einen Benennungsbereich, in dem ein Name genau einmal definiert werden darf. Die direkte Entsprechung ist also eine Java package.



Integers. Die Java Integer Typen short (16 Bit), int (32 Bit) und long (64 Bit) sind vorzeichenbehaftet (signed) und entsprechen den vorzeichenbehafteteten IDL Integertypen short, long und long long [30]. Zusätzlich stellt IDL allerdings noch drei entsprechende Integertypen ohne Vorzeichen (unsigned) zur Verfügung (nämlich unsigned short, unsigned long und unsigned long long). Dieses ist eines der Probleme, das bei der Entwicklung einer IDL nach Java Sprachabbildung gelöst werden muß. In der von JavaSoft definierten Sprachabbildung werden die unsigned Integer Typen auf ihre vorzeichenbehafteten Entsprechungen abgebildet. Aus einem unsigned short wird zum Beispiel ein signed short. In ihrer binären Repräsentation sind die Zahlen zwar gleich, aber der Java Interpreter behandelt nun große IDL Integers als negative Zahlen. Der Programmierer eines Java Programms, das einen so generierten Stummel enthält, muß also, falls er mit IDL Integers ohne Vorzeichen arbeitet, selbst Operationen (besonders Verleiche und Modulo) bereitstellen, die Java Integers behandeln, als seien sie unsigned.

Eine alternative Möglichkeit für eine Sprachabbildung ist es, die unsigned short auf int und unsigned long auf long abzubilden, also auf den jeweils nächsten, größeren Datentyp. Für die 64 Bit Zahlen ist dies natürlich keine Alternative, trotzdem hätte man das Problem wenigstens für unsigned short und unsigned long gelöst. In der JavaSoft Sprachabbildung wurde dieser Weg nicht gewählt, da es für erstrebenswert gehalten wird, eine x Bit Repräsentation wieder in eine x Bit Repräsentation zu überführen [30].

Der 64 Bit IDL Integertyp wird in der CORBA 2.0 Definition allerdings gar nicht erwähnt und kommt auch im Orbix IDL Java mapping nicht vor. Dort wird unsigned short auf short und

[3] In JavaSofts Sprachabbildung nach [30] fehlen bis jetzt der IDL Datentyp „any“ sowie die für CORBA grundlegende Einbindung der DII und der DSI.

in Java long umgewandelt.

Strings. Der IDL Stringtyp wird auf den Java Typ java.lang.String abgebildet. Hieraus ergeben sich zwei Probleme: Erstens kann ein java.lang.String String UNICODE Zeichen enthalten, während IDL Strings nur aus Zeichen aus der UNICODE Untermenge ISO 8859.1 bestehen dürfen. Dieses Problem wird in den vom Übersetzer erzeugten Java Routinen (die Teile des Stummels bzw. Rumpfes sind) für das Umwandeln der Datenrepräsentation (dem marshalling und demarshalling) gelöst. Vor dem Umwandeln eines Java Strings in die Übertragungsrepräsentation wird überprüft, ob die Zeichen zulässige ISO 8859.1 Zeichen sind. Falls sie nicht zu dieser UNICODE Untermenge gehören sollten, wird eine omg.corba.CharacterRangeException ausgelöst. Das zweite Problem wird ebenfalls vom Stummelcode bearbeitet. Ein IDL String kann nämlich mit einer Längenbegrenzung (bounded string) versehen sein, während java.lang.string Strings keine Längenbegrenzung haben (unbounded string). Beim marshalling und demarshalling wird nun die Längenbegrenzung überprüft. Allerdings geht aus dem JavaSoft Dokument [30] nicht hervor, wie der Stummel reagiert, falls ein Java Programm versucht, einen String, der länger ist als durch die Begrenzung zugelassen, an einen ORB zu übergeben. Möglicherweise wird wieder eine Exception ausgelöst oder aber der überschüssige Anteil wird ohne Warnung abgeschnitten.

Zusammengesetzte Datentypen am Beispiel enum. Die in IDL bekannten zusammengesetzten Datentypen (constructed types) enum, struct, union, sequence und array werden bis auf sequence und array, die auf Java Arrays abgebildet werden, auf entsprechende Java Klassen abgebildet, die die jeweilige Semantik der IDL Typen implementieren.

Als Beispiel soll hier die Definition von Aufzählungstypen mit dem Schlüsselwort „enum“ dienen. Ein Aufzählungstyp (enumerated type) besteht aus einer geordneten Liste von Bezeichnern. Die Ordnung wird durch die Reihenfolge der Einführung der möglichen Werte des Typs in der Definition erklärt und ist notwendig, um auf einem Aufzählungstyp Vergleiche und Vorgänger/Nachfolger Funktionen zu definieren (also auch für Operationen wie Addition oder Multiplikation).

Beispiel für die Definition eines Aufzählungstyps in IDL:

```
module Beispiel {
       enum lecker_Essen { Pommes, Currywurst, Burger, Döner }
} ;
```

Der neu definierte Datentyp heißt nun lecker_Essen und kann die Werte Pommes, Currywurst, Burger oder Döner annehmen.

In einem IDL Aufzählungstyp dürfen bis zu 2^{32} Bezeichner spezifiziert werden. Also muß auf ein Java Konstrukt abgebildet werden, das ebenfalls eine derart große Menge unterschiedlicher Werte zuläßt. Sowohl die JavaSoft Sprachabbildung, als auch die zu Orbix gehörende, bilden auf eine Java Klasse ab, die für jeden möglichen Wert, den eine Variable vom Aufzählungstyp annehmen kann, einen entsprechenden int Wert zur Verfügung stellt.

Der JavaSoft IDL nach Java Compiler übersetzt das Beispiel so:

```
package Beispiel;
public class lecker_essen {
```

```
                                    Currywurst=1,
                                    Burger=2,
                                    Döner=3;
            public static final int narrow(int i)
                                    throws omg.corba.EnumerationRangeException
                                    { ... }
}
```

Die für die Werte eines Aufzählungstyps definierten Variablen sind Klassenvariablen. Sie dürfen nicht verändert werden (final), da sie semantisch nicht die Rolle einer Variable sondern die Rolle des Werts einer Variable spielen. Aus dem selben Grund sind sie Klassenvariablen (static). Es hätte keinen Sinn, verschiedene Instanzen der Klasse lecker_Essen einzuführen, die jeweils eine eigene Variable Burger besäßen. Die möglichen Werte eines Datentyps haben keine individuelle Existenz, was sie ja prinzipiell von Objekten unterscheidet. (Das heißt: eine „Eins“ ist immer gleich jeder anderen „Eins“. „Einsen“ werden nicht voneinander unterschieden. Genauso ist das hier mit „Pommes“).

Für den Interpreter sind die Variablen, die semantisch zwar vom Aufzählungstyp sind, jedoch ints. Sie können also Werte annehmen, die außerhalb des Bereiches liegen, auf dem Werte für die Aufzählung definiert sind. Um dies zu überprüfen, fügt der Übersetzer automatisch die Methode narrow ein. Ist der Parameter ein zulässiger Wert für den Aufzählunstyp (das heißt in diesem Fall ein int von zwischen 0 und 3), so gibt narrow genau den Parameter wieder zurück. Ist dies aber nicht der Fall, so wird eine omg.corba.EnumerationRangeException ausgelöst. Dies ist eine RunTime Exception, daher können Programmierer narrow einfach in ihre Programme einbauen, sobald sie ints benutzen, die von ihrer Semantik her zu einem Aufzählungstyp gehören, ohne zusätzliche Exceptions zu deklarieren. Werden alle Integervariablen, die Werte des Aufzählungstyps repräsentieren, immer zusammen mit narrow benutzt, so ist zugesichert, daß sie immer im definierten Bereich bleiben.

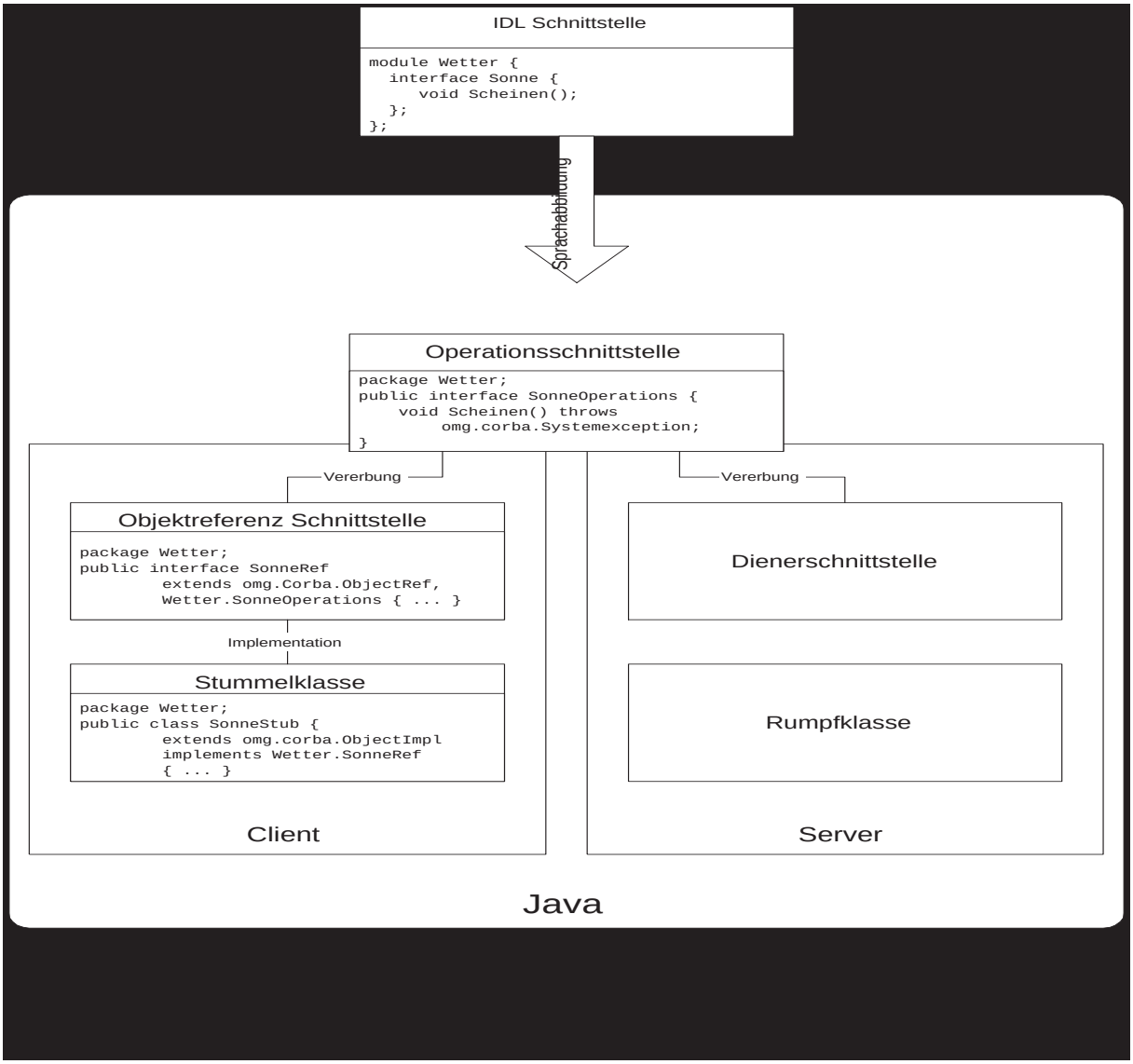
Folgendes Beispiel zeigt die Verwendung des Aufzählungstyps aus obigem Beispiel. Hiermit soll auch verdeutlicht werden, wie ein Integer Wert mit einer Aufzählungssemantik zusammen mit der Methode narrow benutzt wird. Mittagessen ist natürlich vom Typ int.

```
switch (Beispiel.lecker_essen.narrow(Mittagessen)) {
   case Beispiel.lecker_essen.Pommes:
       System.out.println("Es gibt Pommes."); break;
   case Beispiel.lecker_essen.Currywurst:
       System.out.println("Es gibt Currywurst."); break;
   case Beispiel.lecker_essen.Burger:
       System.out.println("Es gibt Burger."); break;
   case Beispiel.lecker_essen.Doener:
       System.out.println("Es gibt Doener."); break;
   default:
       System.out.println("Unmoeglich"); break;
}
```

Der default Fall kann nie eintreten, da narrow bereits eine exception ausgelöst hätte.

tions auf Java stellt der Übersetzer eine Klasse omg.corba.CORBAException zur Verfügung. Es gibt zwei Unterklassen von omg.corba.CORBAException: omg.corba.SystemException und omg.corba.UserException. Die Standard IDL Systemexceptions, die in der CORBA Spezifikation definiert werden, sind Unterklassen von omg.corba.SystemException. In IDL definierte Exceptions erben von omg.corba.UserException. omg.corba.CORBAException ist Unterklasse von java.lang.Exception und ist somit „throwable“.

Interfaces. Der Zweck von IDL Code ist natürlich die Definition einer Schnittstelle. Deshalb ist der interessanteste Teil der Sprachabbildung von IDL auf Java die Frage, wie die Schnittstellendefinition mit dem IDL Schlüsselwort „interface“ mit den dazugehörigen Vererbungsregeln auf Java übertragen wird. Der Übersetzer muß in der Lage sein, Stummel und Rümpfe für die Client- bzw. die Server-Seite zu erzeugen. Der Stummel muß dem Client Java Applet die Möglichkeit geben, Objektreferenzen von Server Objekten zu erlangen. Die Signatur (der „Typ“) der Objektreferenz wird durch die IDL Schnittstellendefinition definiert.



daß Java Klassen nur von genau einer Klasse direkt erben können (der immediate superclass), während die Vererbungshierarchie in IDL nicht unbedingt durch einen Baum dargestellt werden können muß (multiple inheritance). Dementsprechend müssen Java Interfaces verwendet werden, um dieses Problem zu umgehen. Da aber auch ausführbarer Programmcode erzeugt werden muß, müssen IDL Schnittstellendefinitionen auch auf Klassen abgebildet werden. Ein Übersetzer, der entsprechend der JavaSoft Sprachabbildung vorgeht, erzeugt aus einer Definition zwei Java interface Deklarationen und eine Java Klasse auf der Clientseite. Das eine Java interface heißt Java Dienstschnittstelle (Operations Java Interface), das andere heißt Java Objektreferenz-Schnittstelle (Object Reference Java Interface). Die Klasse heißt Java Stummelklasse (Java Stub Class).

Die Java Dienstschnittstelle wird nicht direkt vom Programmcode auf der Clientseite benutzt. Sie gibt die Signaturen der Operationen an, die in der IDL Schnittstelle definiert werden. Die vom Übersetzer erzeugte Schnittstelle hat den Namen der IDL Schnittstelle mit dem Suffix „Operations“.

Sei zum Beispiel die folgende IDL Schnittstellendefinition gegeben:

```
module Beispiel {
       typedef ... Ergebnistyp; /* IDL Typdefinition */
       typedef ... Parametertyp; /* irgendwelcher Typen */
       exception etwas_stimmt_nicht { ... } ;
       interface Dienst {
                 Ergebnistyp eine_Operation(in Parametertyp arg)
                             raises etwas_stimmt_nicht;
       } ;
} ;
```

In diesem IDL Beispiel werden zuerst zwei IDL Typen und eine IDL exception definiert, deren Details hier nicht weiter interessieren. Danach wird eine Schnittstelle „Dienst“ definiert, die eine Operation mit dem Namen „eine_Operation“ zur Verfügung stellt. eine_Operation gibt Ergebnisse vom Typ Ergebnistyp zurück und wird mit einem Parameter vom Typ Parametertyp aufgerufen. Das IDL Schlüsselwort „in“ bezeichnet die gewünschte Aufrufsemantik. „in“ entspricht der von Java benutzten Wertübergabe (pass-by-value), in der das Argument von der Operation nicht verändert werden kann. Die anderen von IDL angebotenen Aufrufsemantiken sollen hier nicht beschrieben werden. „raises“ gibt an, welche Exception von dieser Operation ausgelöst werden kann.

Der Übersetzer erzeugt hieraus folgenden Java Code für die Java Dienstschnittstelle:

```
package Beispiel;
... Javadefinition des Ergebnistyps und des Parametertyps ...
... Javacode für die IDL Exception etwas_stimmt_nicht ...
package Beispiel;
public interface DienstOperations {
       Beispiel.Ergebnistyp Dienst eine_Operation(Beispiel.Parametertyp arg)
               throws omg.corba.SystemException, Beispiel.etwas_stimmt_nicht;
}
```

in Java notwendig ist. Wenden wir uns also als nächstes der Java Objektreferenz-Schnittstelle zu. Die vom Client Programm benutze Schnittstelle ist die Java Objektreferenz-Schnittstelle. Sie erbt von der zugehörigen Java Dienstschnittstelle und erweitert sie um jene Operationen, die auf allen Objektreferenzen möglich sein müssen, in dem sie von der Klasse omg.corba.ObjectRef erbt, die all diese Methoden deklariert. Außerdem erbt die Objektreferenz-Schnittstelle von all jenen Objektreferenz-Schnittstellen, die im IDL Quellcode als Elternteile dieser Schnittstelle angegeben sind. Das heißt: Die IDL Vererbungshierarchie wird in Java in der Objektreferenz-Schnittstelle modelliert. Die Objektreferenz-Schnittstelle definiert den Typ für eine Objektreferenz. Java Objektreferenz-Schnittstellen haben den Namen der IDL Schnittstelle erweitert um den Suffix „Ref“.

Gegeben sei folgende IDL Schnittstellendefinition:

```
module Beispiel {
       interface Ursprung { } ;
       interface Linker_Nachfahre : Urpsrung { } ;
       interface Rechter_Nachfahre : Ursprung { } ;
       interface Erbe : Linker_Nachfahre, Rechter_Nachfahre { } ;
} ;
```

Hieraus macht der Übersetzer folgende Objektreferenz-Schnittstellen:

```
package Beispiel;
public interface UrsprungRef extends omg.Corba.ObjectRef,
        Beispiel.UrsprungOperations { }
package Beispiel;
public interface Linker_NachfahreRef extends omg.Corba.ObjectRef,
        Beispiel.UrsprungRef,
        Beispiel.Linker_NachfahreOperations { }
package Beispiel;
public interface Rechter_NachfahreRef extends omg.Corba.ObjectRef,
        Beispiel.UrsprungRef,
        Beispiel.Rechter_NachfahreOperations { }
package Beispiel;
public interface ErbeRef extends omg.Corba.ObjectRef,
        Beispiel.Linker_NachfahreRef,
        Beispiel.Rechter_NachfahreRef,
        Beispiel.ErbeOperations { }
```

Somit wurde die in der IDL Schnittstellendefinition erklärte Vererbungsstruktur vollständig auf die Objektreferenz-Schnittstellen übertragen, und die Java Signaturen (Typen) der Objektreferenzen für Objekte, die die jeweilige IDL Schnittstelle implementieren, wurden ebenfalls definiert.

Der dritte Bestandteil des Javacodes, den der Übersetzer aus einer IDL Schnittstellendefinition erzeugt, ist die Java Stummelklasse. Ihr Name ist der Name der IDL Schnittstelle mit dem Suffix „stub“. Man sollte sich vor Augen halten, daß insbesondere die Java Stummelklasse (und eigentlich auch die beiden Java Schnittstellen) Teile des Object Request Brokers sind, die auf der

Schnittstelle gegebenen Operationen. Diese Implementierung ist nun aber natürlich nicht der Programmcode, der die Funktionalität dieser Operationen tatsächlich bereitstellt, sondern nur der Programmcode, der für eine Weitergabe des Aufrufs an die, möglicherweise weit entfernte, Implementierung sorgt. Zum Beispiel durch die Weitergabe an einen auf einem anderen Rechner laufenden ORB. Dieser Vorgang ist für das Java Programm, das diese Methoden benutzt, allerdings nicht sichtbar und hängt von der ORB Implementierung ab, die diese Sprachabbildung benutzt. Zusätzlich zu den Operationen der Objektreferenz-Schnittstelle (dies sind natürlich auch jene, die die Objektreferenz-Schnittstelle erbt) sorgt die Stummelklasse auch dafür, daß solche Operationen, die für jede Objektreferenz, nach CORBA Spezifikation, zur Verfügung stehen müssen, implementiert sind (das sind dementsprechend die, die die Objektreferenz-Schnittstelle von omg.corba.ObjectRef erbt).[4] Einige der Implementierungen erbt die Stummelklase von omg.corba.ObjectImpl, andere werden in jeder Stummelklasse neu implementiert. Auf jede Objektreferenz anwendbaren Operationen sind isA, release und narrow. IsA gibt einen Boolean Wert zurück, der anzeigt, ob das durch die Objektreferenz angegebene Objekt die als Parameter übergebene Schnittstelle erfüllt. Release zeigt an, daß die Objektreferenz nicht mehr benötigt wird. Narrow führt eine Typumwandlung durch. Eine Objektreferenz wird als Parameter übergeben. Die Signatur der Objektreferenz wird zu der Signatur der Objektreferenz-Schnittstelle, zu der die Stummelklasse gehört, aus der narrow aufgerufen wurde. In jeder Stummelklasse müssen nun die static Version von isA (wovon es in omg.corba.ObjectImpl auch eine nicht static Version gibt) sowie narrow implementiert werden (zusätzlich zu den Operationen, die die Schnittstelle eigentlich anbietet).

Der Java nach IDL Übersetzer erzeugt aus der IDL Definition

```
module Wetter {
       interface Sonne {
                 void Scheinen();
       };
};
```

die Java Stummelklasse

```
package Wetter;
public class SonneStub
       // die Stummelklasse erbt Implementierungen
       // von omg.corba.ObjectImpl und implementiert
       // die in der zugehörigen Objektreferenzschnittstelle
       // definierten Operationen
       // (d.h. macht den Zugriff über den ORB Kern möglich).
       extends omg.corba.ObjectImpl implements wetter.SonneRef {
       // eine SystemException tritt auf, falls es beim Aufruf
       // des Server Objekts über den ORB irgendwo Probleme gibt
       public void Scheinen() throws omg.corba.SystemException { ... }
       public static boolean isA(omg.corba.ObjectRef arg)
              throws omg.corba.SystemException { ... }
       // Man beachte die zusätzliche Exception, die
```

[4] Es handelt sich bei omg.corba.ObjectRef um die Objektreferenz- Schnittstelle für die CORBA::Object Schnittstelle, von der alle CORBA IDL Schnittstellen erben.

```
        // erfolgreich sein muß.
        public static wetter.SonneRef narrow(omg.corba.ObjectRef arg)
                throws omg.corba.NarrowCoercionException,
                        omg.corba.SystemException { ... }
}
```

Zusammenfassend läßt sich also sagen, daß eine IDL Schnittstellendefinition von einem IDL nach Java Übersetzer, der den Richtlinien von JavaSoft entspricht, die Schnittstellendefinition auf drei Komponenten überträgt. Die Komponenten sind: 1. Die Java Dienstschnittstelle, die die Signatur der auf der Schnittstelle definierten Operationen festlegt. 2. Die Java Objektreferenzschnittstelle, die den Typ der zugehörigen Objektreferenz definiert sowie die Vererbungshierarchie modelliert. 3. Die Java Stummelklasse, die die benötigten Implementierungen für den Operationsaufruf liefert.

Diese drei Ergebnisse der Übersetzung decken allerdings nur die Client-Seite ab. Der Übersetzer muß aber nach CORBA Spezifikation nicht nur einen Stummel, sondern auch einen Rumpf auf der Server-Seite erzeugen (hier ist von dem Fall die Rede, daß die Objekte in Java implementiert sind. Natürlich können CORBA Objekte in jeder Sprache implementiert sein, für die eine Sprachabbildung vorliegt). Die Schnittstellendefinition wird auf der Serverseite auf zwei Komponenten übertragen, die hier nur kurz erläutert werden sollen. Diese zwei Komponenten sind die Java Dienerschnittstelle (Java Servant Interface) und die Java Rumpfklasse (Skeleton Java Class). Die Dienerschnittstelle erklärt die Methoden, die eine Implementierung beinhalten muß, um die betreffende Schnittstelle (aus der IDL Quelle) zu erfüllen. Die Dienerschnittstelle ist ein Java interface, das den Namen der IDL Schnittstelle mit dem Suffix „Servant“ trägt. Die Dienerschnittstelle erbt von den Dienerschnittstellen jener IDL Schnittstellendefinition, von denen die eigene IDL Schnittstellendefinition erbt. Zusätzlich erbt die Dienerschnittstelle von der zugehörigen Java Dienstschnittstelle, denn diese enthält die Signaturen aller jener Operationen, die sowohl auf der Client als auch auf der Serverseite in der Schnittstelle enthalten sein müssen. Hier wird deutlich, daß die Dienstschnittstelle nicht wirklich eindeutig zur Client-Seite gehört, sondern sowohl auf der Client-Seite als auch auf der Server-Seite gebraucht wird. Dies ist auch die Erklärung für die Trennung von Dienstschnittstelle und Objektreferenz-Schnittstelle. Die Dienstschnittstelle enthält den Teil der Schnittstelle, der auf Clientseite und auf Serverseite vorhanden ist. Die zugehörige Implementierung, bei der es sich um eine beliebige Javaklasse handeln kann, muß nun nur durch das Java Schlüsselwort „implements“ bekanntgeben, daß sie die Dienerschnittstelle implementiert. Die Implementierung wird selbstverständlich nicht vom Übersetzer generiert, sondern ist ein Java Programm, das Programmierer geschrieben haben, um eine gewisse Funktionalität für die Clients des ORBs zur Verfügung zu stellen.

Die Java Rumpfklasse ist eine Klasse, deren Name ebenfalls der Name der Schnittstelle mit einem Suffix versehen ist (nämlich „Skeleton“). Die Rumpfklasse enthält die Methoden, die nötig sind, um aus einer Instanz der Implementierung eine Objektreferenz zu gewinnen, die an den Client weitergegeben werden kann.

10.2.2 ORB Implementierungen

In diesem Abschnitt sollen drei CORBA gemäße ORB Implementierungen, die einen IDL nach Java Übersetzer beinhalten, vorgestellt werden. Die drei Produkte sind Orbix, NEO/JOE und Black Widow.

Standards auf den Markt gebracht und ist für alle gängigen Betriebssysteme erhältlich. Inzwischen wurde für Orbix eine IDL nach Java Sprachabbildung definiert, die sich, wie bereits angesprochen, in einigen Punkten von der vorgestellten JavaSoft Abbildung unterscheidet. Es gibt eine Java Implementierung des ORB, die, zusammen mit den erzeugten Java Stummeln, als Java Library auf einen Client-Rechner heruntergeladen werden kann.

NEO und JOE. NEO ist SunSofts CORBA konformer ORB. NEO ist das Folgeprojekt von SunSofts DOE (Distributed Objects Environment). JOE ist ein Zusatz für NEO, der den Umgang mit Java und dem World Wide Web ermöglicht. JOE besteht aus einem IDL nach Java Übersetzer, einem in Java implementierten ORB und einer Java API nach NEO. Das Funktionsprinzip ist auch hier wie vorgestellt: Der JOE ORB wird zusammen mit einem Applet in einen Web Browser geladen und regelt die Kommunikation mit den Objekten von NEO ORBs.

JOE benutzt die in Ausschnitten vorgestellte JavaSoft Sprachabbildung. Aufbauend auf das in 10.2.1 vorgestellte Module „Wetter“ soll nun an einem Beipspiel erläutert werden, wie ein Java Programm mit JOE auf ein CORBA Objekt zugreifen kann.

Das folgende Applet sorgt bei seiner Initialisierung dafür, daß die Methode Scheinen() auf einem Objekt ausgeführt wird, dessen Schnittstelle die IDL Definition von Wetter erfüllt. Momentan heißen die CORBA spezifischen Packages unter JOE noch „sunw.*“ und nicht „omg.*“. Um das Applet leichter erklären zu können, wird es in zwei Teilen vorgestellt.

```
import sunw.services.*;
import java.awt.Graphics;
public class Sommer extends sunw.services.JOEApplet
{
       sunw.corba.ObjectRef obj;
       Wetter.SonneRef dieSonne;
       ...
```

Die erklärte Klasse importiert als erstes die in JOE enthaltenen Packages und erbt alle Methoden des Standard JOEApplets, die später im Programm noch verwendet werden. Es wird eine Objektreferenz obj auf ein CORBA Objekt deklariert. Diese Referenz ist aber generisch (d.h. nicht weiter typisiert). Sie könnte auf ein beliebiges CORBA Objekt verweisen. Mit dieSonne wird eine Referenz auf ein Objekt deklariert, das die Schnittstelle Wetter.SonneRef hat. SonneRef enthält entsprechend der Sprachabbildung auch alle Operationen, die in SonneOperations deklariert wurden, somit alle Operationen, die ein entsprechendes Dienstleistungsobjekt, das irgendwo, vielleicht auf einem anderen Rechner, liegt, anbietet. Außerdem erbt Wetter.SonneRef auch von sunw.corba.ObjectRef (entspricht omg.corba.ObjectRef). Das heißt, daß ein Objekt, auf das die Objektreferenz dieSonne verweist, auf jeden Fall alle Schnittstellen erfüllt, die auch obj erfüllt (aber nicht umgekehrt).

```
      void init() {
           obj = find("SonneServer");
           dieSonne = Wetter.SonneStub.narrow(obj);
           dieSonne.Scheinen();
      }}
```

gestellt) aufgerufen. Der String-Parameter ist der Name eines bestimmten Objekts, anhand dessen der vom ORB zur Verfügung gestellte Naming Service vorhandene Objekte identifizieren kann (im Interface und im Implementierung Repository). find überprüft nun, ob das gewünschte Objekt bereits irgendwo vorhanden ist und gibt eine Objektreferenz darauf zurück (hier wird davon ausgegangen, daß das Objekt bereits früher erzeugt wurde). Der Rückgabewert ist eine Objektreferenz auf dieses Objekt, das die Schnittstelle sunw.corba.ObjectRef erfüllt. obj enthält nun diese Objektreferenz. Das Objekt, auf das obj verweist, ist aber eine Instanz der Klasse SonneStub und erfüllt nicht nur sunw.corba.ObjectRef sondern sogar dessen Obermenge Wetter.Sonne. Davon weiß find aber nichts. find liefert nur Objektreferenzen für Objekte, die sunw.corba.ObjectRef erfüllen. Mit Hilfe eines Aufrufes der narrow Methode aus der Klasse SonneStub (narrow ist static) wird nun für das Objekt eine Objektreferenz, die der Schnittstelle von Wetter.Sonne entspricht, erzeugt. Dies gelingt aber nur, wenn das Objekt, auf das obj verweist auch tatsächlich die volle Wetter.SonneRef Schnittstelle unterstützt. Anderenfalls liefert narrow, wie bereits erwähnt, eine Exception. Jetzt hat man eine Objektreferenz, die man benutzen kann, um Methoden aus SonneOperations aufzurufen, als seien sie tatsächlich lokal implementiert.

Black Widow. Auch PostModerns/Visigenics Black Widow funktioniert nach dem genannten Prinzip. Visigenic bietet mit ORBELine bereits einen CORBA konformen ORB an, von dem es aber keine Java Implementierungen gibt. Black Widow beinhaltet einen IDL nach Jave Übersetzer und die Black Widow Java Runtime. Eine Java Bibliothek, die den eigentlichen in Java implementierten ORB darstellt. Die Besonderheit an Black Widow ist aber die Benutzung des IIOP zur Kommunikation zwischen Client-Applikation und Server-Applikation. Damit ist Black Widow die einzige der drei vorgestellten Java ORB Implementierungen, die bereits CORBA 2.0 konform ist und tatsächlich mit CORBA 2.0 ORBs anderer Hersteller kommunizieren könnte.

10.2.3 Zusammenfassung

CORBA konforme ORBs, die versuchen, die Vorteile von Java mit den Vorteilen von CORBA zu verbinden, funktionieren alle nach einem ähnlichen Prinzip. Ein Teil der ORB Funktionalität wird mit einem Java Applet übertragen. Danach verläuft die Zusammenarbeit mit CORBA Dienstgeberobjekten nach den Grundsätzen von CORBA. Es wird kein weiterer Code übertragen, sondern nur Objektreferenzen. Das Java Programm greift auf die auf anderen Rechnern gelagerten Objekte zu, indem es die Methoden der in Java vorhandenen, vom Übersetzer erzeugten, Stummelroutinen aufruft. Das diese Stummelroutinen mit einem Objekt kommunizieren, das gar nicht Teil des laufenden Java Applets ist, bleibt versteckt. Die vorhandenen Implementierungen unterscheiden sich in der benutzten Sprachabbildung und darin, ob sie das IIOP verwenden oder nicht. Alle Hersteller haben angekündigt, daß ihre ORBs das IIOP unterstützen werden.

10.3 Schlußbemerkung

Die Zusammenarbeit von Java und CORBA wird von allen Herstellern, insbesondere vom Java Hersteller und OMG Mitglied Sun, als vielversprechender Ansatz dargestellt. Die Vorteile, die die vorgestellte Art der Zusammenarbeit zwischen Java und CORBA bietet, liegen auf der Hand: Durch CORBA können Java Applets auf leistungsfähige System am „anderen Ende der Leitung“ zugreifen, wodurch die Anwendungsmöglichkeiten des World Wide Web für jede Art von Applikation erreicht wird. Ein Nachteil sowohl von Java, als auch von CORBA ist, daß der Transport von vollständigen Objekten (d.h. inklusive Zustand) nicht unterstützt wird. Dies führt dazu, daß

weiteres möglich, Objekte auf einen mobilen Laptop Computer zu übertragen und später wieder in die ORB Umgebung einzufügen [6]. Dieser Nachteil könnte aber sicherlich durch Methoden, die den gesamten Objektzustand als Ergebnis zurückliefern, umgangen werden. Der bedeutenste Nachteil ist eine Frage des Datenschutzes und spielt bei Intranetanwendungen keine Rolle. Falls aber Privatpersonen immer mehr dazu übergehen (müssen), sich ihre Rechnerdienstleistungen über ein Netzwerk bei Betrieben zu kaufen, so bedeutet dies für die Betriebe, die diese Dienstleistungen bereitstellen, unbeschränktes Wissen darüber, was die Person mit ihrem Rechner tut. Gleichzeitig ist der Betrieb, der die Dienstleistungen bereitstellt, in der Lage, Dienstleistungen zu verweigern oder in veränderter Form zu gewähren.

Seminarbeitrag von **Markus Blaum**

11.1 Wozu Java-Prozessoren?

Nachdem der Internetboom der Programmiersprache Java, die das Unternehmen Sun ursprünglich für Konsumelektronikgeräte entwickelte, zum Durchbruch verhalf, wird Java auf den verschiedensten Plattformen eingesetzt. Die Plattformunabhängigkeit der Java-Applets bringt besonders in heterogenen Systemen viele Vorteile. Entwickler schreiben ihre Appletts in Java und übersetzen sie zu einem sog. Bytecode, welcher einem ausführbaren Binärcode entspricht und nicht mikroprozessorspezifisch ist. Er ist kompakter, sicherer und kann schneller ausgeführt werden als das entsprechende Javaprogramm. Er wird von einem Interpreter, der virtuellen Java-Maschine, zur Laufzeit in den jeweiligen Maschinen-Code übersetzt. Auf den ersten Blick scheint dieser Ansatz ein Rückschritt zu sein, da das Interpretieren zur Laufzeit mehr Zeit und Leistung benötigt.

Hier setzen nun die Java-Prozessoren an, die den Laufzeit-Interpreter überflüssig machen, da sie den Bytecode direkt ausführen, wodurch also letztendlich der Bytecode den eigentlichen Befehlssatz des Java-Prozessors darstellt. Java hat sich als portabel und leistungsstark erwiesen, aber die Größe der Anwendungen wird klar durch die Leistungsfähigkeit des Java-Interpreters begrenzt. So haben Untersuchen ergeben, daß interpretierte Java-Applikationen nur 3-10% der Geschwindigkeit des vergleichbaren Orginal-X86-Codes erreichen (Code der duch einen 32-bit Borland-Übersetzer erstellt wurde.). Sun erwartet daher durch den Einsatz eines Java-Prozessors eine Verbesserung der Ausführungszeit um den Faktor 5 oder mehr. Durch den erzielten Geschwindigkeitsgewinn soll es anderen Anbietern wiederum ermöglicht werden, billige Java-Endgeräte (Internet/Netz-PCs, Router, Modems und verwandte Produkte wie Mobiltelefone und Notepads) zu bauen. Es wird allgemein von einer Geschwindigkeitssteigerung und Hardwareersparnis um ein Fünfzigfaches gesprochen. Es soll eine neue Klasse von Netzwerkterminals ermöglicht werden, die ohne umfangreiche Architektur, leistungsstarke CPUs, große Hintergrundspeicher und andere Komponenten auskommt. Somit entfallen sehr viele Kosten: einerseits die Kosten für die Hardware selbst und andererseits die – meist noch größeren – Kosten für deren Unterhalt, da einfach ein neues Java-Programm über Netz ausgeführt werden kann [18, 29, 54].

11.2 Entwicklung von Java-Prozessoren bei Sun

Derzeit beschäftigt sich nur das Unternehmen Sun Microsystems mit der Entwicklung von Java-Prozessoren. Sun beabsichtigt, zur Unterstützung von Java im Laufe der kommenden zwei Jahre Java-optimierte RISC-CPUs auf den Markt zu bringen, die jedoch ausschließlich für Java-Applets geeignet sein sollen. Die aus Software, Prozessoren und Systemen bestehende Java-Technologie

schätzt, daß bei einem Marktvolumen für Mikroprozessoren und Mikrokontroller von über 60 Milliarden Dollar bis zum Jahr 1999 die Java-Prozessoren nicht nur einen erheblichen Anteil für sich erobern, sondern auch gleichzeitig als Katalysator für die Entwicklung neuer Bausteintypen dienen werden, welche sich derzeit mit herkömmlichen Mitteln zu keinem vernünftigen Preis realisieren lassen. Der Markt für Java-Chips bis zum Jahr 1999 wird auf 15 Milliarden Doillar geschätzt. Den Kernmarkt bildet dabei das Internet mit seinen rund 40 Millionen Benutzern mit bis zu 6,6 Stunden Wochennutzung, das immer noch 75% jährliche Wachstum bei derzeit 6,6 Millionen angeschlossenen Hosts und einer Verdoppelung der monentan 27.000 Web-Servern innerhalb von 53 Tagen. Das Wachstum des Marktes ist hauptsächlich auf zwei Ursachen zurückzuführen:

- Das zunehmende Vordringen von High-End Embedded Prozessoren in die Bereiche der Automatisierung, Kommunikation und Konsumelektronik, sodaß in den Unternehmen der Zukunft Hunderte oder sogar Tausende von Mikrokontrollern Verwendung finden werden. Es wird geschätzt, daß 1999 bereits 90% aller Mikrokontroller im Fabrikmanagement eingesetzt werden, wo es besonders auf Plattformunabhängigkeit ankommt. Aber auch im Haushaltsbereich sollen ca. 100 Mikrocontroller ihren Dienst tun. Unter anderem werden bis zum Jahr 2000 auch 145 Millionen Handies mit Mikrokontrollern erwartet, die auch von Java-Prozessoren profitieren könnten.
- Der Markt für bestehende Systeme ist keineswegs gesättigt, somit teilen sich in den USA durchschnittlich 2 Mitarbeiter einen PC während es in Europa schon 4, in Japan 6, in China 1000 und in Indien sogar 2000 sind [61].

Insgesamt sind für die nächsten zwei Jahre zunächst drei Chips geplant: „Pico-Java“, „Micro-Java“ und „Ultra-Java“, welche im folgenden noch genauer erläutert werden [49].

11.3 Die Implementierung von Java in Hardware

Der Java-Prozessor wird im Mikrocode große Vielfalt ausweisen. Einfache und oft gebrauchte Grundbefehle werden ohne ihn auskommen, wobei aber die komplexeren Befehle auf den Mikrocode zurückgreifen werden müssen, um entsprechende Folgen von Aktionen, wie Auswerten der Konstanten-Tabelle, Methodensuche und das permanente Abarbeiten der Entscheidungstabellen, effizient ausführen zu können. Viele Java-Befehle wie lokales Laden/Speichern, Konstanten-Addition, Vergleichen/Verzweigen und Befehle für skalare Berechnungen können in vergleichbarer Weise wie bei heutigen Hochleistungsprozessoren implementiert werden. Um den bei Operanden-Speicher-Konflikten entstehenden Leistungsengpaß zu beheben, finden bei der Lokalisierung im Werte-Stack ebenfalls konventionelle und leistungsstarke Techniken der Registerumbenennung Verwendung. Bei den Stack-bearbeitenden Befehlen ist es möglich, die Superskalartechnik einzusetzen. So kann z.B eine Gruppe von Befehlen wie „Load, Load, Add, Store“ erkannt und in eine einzelne interne Drei-Adreß-Operation ungeformt werden. Weiterhin kann man in einer leistungsstarken Implementierung das aktuelle Stack-Fenster (oder auch mehrere) in einem Prozessorregister zwischenspeichern. Dadurch ist es möglich, obige Befehlsfolge als eine einzige interne Register-Register-Operation zu implementieren. Andere leistungssteigernde Techniken wie ein Befehls-/Data-Cache, Verzweigungsvorausberechnung, Reservation Stations, Umordnungs-Puffer für spekulative Befehlsausführung werden alle in einem Java-Prozessor Verwendung finden können. Alles in allem stellt die Stack-Architektur der virtuellen Java-Maschine und das dynamisch gebundene, objektorientierte Design die bisherigen Mikropozessordesigner

Mikropozessordesigner ist es jedoch immer noch schwierig, einen günstigen Weg zu finden, um eine leistungsstarke Implementierung zu finden, die sowohl zur Auswertung der Konstantenmenge (Constant-pool-Resolution) als auch für den Methoden-Aufruf gut geeignet ist. Das Problem besteht in der Tatsache, daß Auswertung und Aufruf ihrem Wesen nach seriell ablaufen und viele Clock-Zyklen und Speicherzugriffe benötigen werden. In der gegenwärtigen Implementierung des Java-Prozessors ziehen die meisten Befehlscodes nur eine Abhängigkeit auf der nächsten Ebene nach sich, was eine Vorausberechnung schwierig macht. Im Vergleich zu traditionellen Mikroprozessoren, bei denen durch Übersetzer und Binder alle symbolischen Referenzen aufgelöst und durch eine einfache Adressierung ersetzt werden, sodaß durch einen einfachen Speicherzugriff ein Wert vom Speicher geladen werden kann, muß das Java-System infolge seines dynamischen Bindens alle symbolischen Referenzen während der Laufzeit auflösen. Gesteht man dem Chip für die letzten Ergebnisse einen Hardware-Cache zu, so können die Auswertung und das Methodenaufrufen beschleunigt werden. Dieser Cache wird jedoch nie eine Treffer-Rate von 100% erreichen und den Java-Prozessor im Vergleich zu seinen konventionellen Kontrahenten verteuern. Im Grunde läuft alles darauf hinaus, daß die sonst ebenbürtigen Java-Prozessoren vergleichsweise mehr Hardware besitzen müssen, um eine vergleichbare Leistung zu erbringen. Da eine Java-Virtual-Maschine automatisch Garbage-Collections ausführt und während dieser Zeit keine Ausführung der Applets möglich ist, versucht Sun, seine Java-Chips mit einer Software-Garbage-Collection zu verbinden. Normalerweise ist die Programm-Abarbeitung zwar eingeschränkt oder gar angehalten, aber eine zwar schwierige, aber nicht unmögliche fortlaufende Garbage-Collection in kleinen Speicherblöcken könnte eine Weiterarbeit der Java-Chips außerhalb dieser kritischen Bereiche weiterhin erlauben und somit ein Anhalten weitestgehend vermeiden. Trotz all dieser Probleme profitiert die Java-Virtual-Maschine von der Stack-Orientierung und vom dynamischem Binden der symbolischen Referenzen. So hat z.B. der Stackcode einen hohen Informationsgehalt und verbessert die Portierbarkeit durch einfache Implementierung, da ein Stackcode-Interpreter sehr gut zu unseren traditionellen Mikroprozessor-Architekturen, wie der des X86, mit wenigen Registern paßt. Da die Java-Virtual-Maschine keine Befehle für Systemmanagement und Resourcenverwaltung (wie Supervisormode-Bits, virtual Adresstranslation, etc.) kennt, obwohl solche bei der Verwendung von konventionellen 32-bit Mikroprozessoren typischerweise vorhanden sind, müssen hierfür erst Alternativen geschaffen werden, bevor der Java-Chip als selbstständiges System gut arbeiten kann. Weiterhin existiert in jedem modernen System ein Betriebssystem-Kern, der die Verwaltung der Betriebsmittel, Prozesse und Threads übernimmt, sodaß ein echter Java-Prozessor zusätzlich solche Features enthalten muß, die über die derzeitige Java-Spezifikation hinausgehen. Beispiele dafür sind eine Unterbrechungsstruktur, ein Prozessorstatus-Register, geschützte Speichersegmente und Adreßsumrechnung [29].

11.3.1 „Pico-Java“

Der Chip besteht aus einem Low-cost-Kern auf RISC-Basis, der für Java optimiert wird. Es wird ein optimales Preis-Leistungs-Verhältnis für den industrieweiten Einsatz angestrebt. Der Chip soll voraussichtlich eine Größe von 25mm^2 haben und ab Mitte 96 für unter 25$ verfügbar sein. Er soll seinen Einsatz in Peripheriegeräten und Mobiltelefonen finden [49, 7].

11.3.2 „Micro-Java“

Der Mikro-Java basiert auf dem Kern von Pico-Java und wird durch applikationsspezifische I/O-Komponenten zur Speicherverwaltung, Kommunikation und Kontrollfunktionen ergänzt. Der

zwischen 25$ und 50$ verfügbar sein. Der Einsatz den Chips liegt im Bereich von Netzwerkgeräte wie Buchungssysteme und Mailterminals oder Spielekonsolen. In seiner Leistung soll der Chip heutige Pentium-Prozessoren schlagen [7, 49].

11.3.3 „Ultra-Java“

Der Ultra-Java soll die drei- bis fünffache Leistung des Mikro-Java-Chips erreichen. Er soll die VIS-Technologie unterstützen und seine Anwendung in Netzwerk-Computern, 3D-Graphik und Multimedia-Applikationen finden. Der Chip soll im vierten Quartal 1997 zu einem Preis von mindestens 100$ verfügbar sein. Die VIS-Technologie, das Visual-Instruktion-Set, unterstützt 2D- und 3D-Bildbearbeitung, Video-Kompression und Dekompression, Audio, Netzwerk, Entschlüsselung und andere Algorithmen. Sie wurde von Sun auf der UltraSPARC eingeführt und ermöglicht bis zu 10 VIS-Operationen pro Takt-Zyklus, d.h. z.B. 2 Milliarden Operationen pro Sekunde auf einem 200Mhz Prozessor. Der Ultra-Java ist als Gegengewicht zu entsprechenden Internet-PC-Entwicklungen bei IBM und Oracle gedacht [49, 7].

11.4 Die Reaktion der Industrie

Derzeit beschäftigt sich Sun als einziges Unternehmen mit der Entwicklung entsprechender Java-Prozessoren. Andere Unternehmen, wie zum Beispiel Silicon Graphics, distanzieren sich von solchen Plänen. Es werden folgende Gegenargumente angeführt:

- Die heutigen Java-Programme sind so klein, daß sie selbst auf leistungsschwachen Internet-PCs mühelos laufen können und den Extraaufwand eines Java-Prozessors nicht rechtfertigen. Es ist anzunehmen, daß die fortwährende Weiterentwicklung traditioneller Prozessoren das Wachstum der Java-Applets kompensieren kann, sodaß auch in Zukunft kein großer Bedarf bestehen wird [45].

- Da sich Java schlecht als Hardwarebasis eignet, wird es voraussichtlich mit einiger Arbeit verbunden sein, Java-Funktionen wie das Verändern einzelner Datenbytes auf dem Chip durchzuführen. Durch ihre wesentlich umfangreichere und kompliziertere Hardware werden daher Java-Prozessoren teurer sein als konventionelle Prozessoren mit vergleichbarer Leistung. (Bei genauerer Betrachtung muß man jedoch feststellen, daß die beiden Prozessoren nicht vergleichbar sind, weil der Java-Prozessor einige weitere Aufgaben wie das dynamische Binden symbolischer Referenzen durchführt, die bei traditionellen Prozessoren gar nicht implementiert sind. Andererseits bietet der Java-Prozessor keine Mechanismen zum Systemmanagement und zur Ressourcenverwaltung [29].)

- Wenn die kompakten Java-Programme über das Internet im Hauptspeicher eingelagert sind, müssen die Anweisungen erst umfangreich dekodiert werden. Im Vergleich dazu können die Anweisungen bei Verwendung eines konventionellen, redundanten RISC-Befehlssatzes leicht und schnell ausgeführt werden.

- Allgemein steht man programmiersprachenspezifischen Prozessoren (wie sie bereits in den siebziger und achtziger Jahren erforscht wurden) eher ablehnend gegenüber, da sich einerseits die kurzen Entwicklungszyklen in Compiler und Programmiersprachen nicht in die Hardware übernehmen lassen und andererseits neue Chip-Konzepte eine Anpassung der entsprechenden Software erfordern.

für Java-Software. In entsprechenden Java-Endgeräten werden sie wohl die Rolle eines Hochleistungs-Java-Coprozessors übernehmen. Daneben werden sie aufgrund ihrer Kompliziertheit bei gleicher Leistung teurer ausfallen als ihre traditionellen Konkurrenten. Nicht zuletzt soll natürlich Suns Java durch die Einführung von Java-Prozessoren zum Standard gemacht werden. Das Gelingen dieses Vorhabens wird jedoch noch in Frage gestellt.

- Es wird vermutet, daß nach dem Erscheinen des ersten Prototypen der Prozessoren noch bis zu einem Jahr vergeht, bis eine umfangreiche Serienfertigung von Endgeräten mit Java-Chips erfolgen kann. Weiterhin rechnet man bei Sun mit Produkt-Verzögerungen oder -Streichungen aufgrund von Marktbeobachtungen zur Rentabilität von Java-Chips [12, 18, 29].

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau.* Addison-Wesley, Reading, Mass., 1992.

[2] Jonathan Amsterdam. Building a computer in software. *byte*, 10(10):113–118, October 1985.

[3] S. Back. Heißer Kaffee – Programmieren in Java. *c't Magazin für Computertechnik*, pages 138–142, February 1996.

[4] D. Bank. *The Java Saga.* Wirded Ventures Ltd., 1995.

[5] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, June 1973.

[6] John Caspers and Anton Eliës. Using CORBA for internet-based workgroup applications. University of Amsterdam.

[7] Casting Java in silicon. Sun Microsystems Inc., Mountain View, CA, February 1996. Report STB-0008.

[8] *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, July 1995.

[9] Bernhard Davignon and Georg Edelmann. *Java oder: Wie steuere ich meine Kaffeemaschine.* tewi Verlag GmbH, München, 1996. ISBN 3-89362-459-7.

[10] Drew Dean, Edward W. Felton, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *IEEE Symposium on Security and Privacy*, 1996. `http://www.princeton.edu/sip/pub/secure96.html`.

[11] John December. *JAVA - Einführung und Überblick.* Markt und Technik Verlag, München, 1995.

[12] Developments to watch: Chips steeped in Java. Business Week – Industrial Edition, no. 12, 1996.

[13] J. Rumbaugh et.al. *Objektorientiertes Modellieren und Entwerfen.* Carl Hanser Verlag, München, Prentice-Hall Internat., London, 1993.

[14] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates, 1996. ISBN 1-56592-183-6.

[15] Peter Glasmacher. FORTH prozessor Novix-4000. *c't Magazin für Computertechnik*, 1987.

[16] J. Gosling and H. McGilton. The Java language environment - a white paper. Sun Microsystems Inc., Mountain View, CA, October 1995. `http://java.sun.com/whitePaper/java-whitepaper-1.html`.

[17] James Gosling. Java intermediate bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations*, 1995.

[18] Java chips boost Applets speed. byte, April 1996.

[19] Java – the inside story. Sun Microsystems Inc., Mountain View, CA. `http://www.sun.com/sunworldonline`.

CA, October 1995.

[21] FAQ - applet security v1.0.2. Sun Microsystems Inc., Mountain View, CA, May 1996. http://java.sun.com/sfaq/.

[22] HotJava(tm): The security story. Sun Microsystems Inc., Mountain View, CA, May 1995. http://java.sun.com/1.0alpha3/doc/security/security.html.

[23] The Java Virtual Machine specification, 1995. http://www.javasoft.com.

[24] Javasoft. Sun Microsystems Inc., Mountain View, CA, 1996. http://www.javasoft.com.

[25] Paul Klint. Interpretation techniques. *Software—Practice and Experience*, 11:963–873, November 1981.

[26] Ralf Kühnel. *Die Java-Fibel.* Addison-Wesley, Reading, Mass., 1996. ISBN 3-8273-1024-5.

[27] Doug Lea. Concurrent programming in Java, tutorials and design patterns. preliminary version.

[28] Laura Lemay and Charles Perkins. *Teach Yourself Java in 21 Days.* Sams.net, 1996. ISBN 1-57521-030-4.

[29] Mark Lentczner. Java's virtual world. *Microdesign Ressources – Microprozessor Report*, March 1996.

[30] Mapping IDL to Java. Sun Microsystems Inc., Mountain View, CA, February 1996.

[31] Chuck McManis. Synchronizing threads in Java. *JavaWorld*, April 1996. http://www.javaworld.com/jw-04-1996/jw-04-synch.htm.

[32] Chuck McManis. Using threads in Java. *JavaWorld*, May 1996. http://www.javaworld.com/jw-05-1996/jw-05-mcmanis.html.

[33] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[34] Step by step - how to implement native code in java. JavaWorld, Nuts & Bolts, June 1996. http://java.sun.com/books/Series/Tutorial/native/index.html.

[35] Implementing native methods in Java release 1.0 alpha. Sun Microsystems Inc., Mountain View, CA, December 1994.

[36] Ross P. Nelson. *Microsoft's 80386/80486 Programming Guide.* Microsoft Press, 1991.

[37] NEO frequently asked questions. Sun Microsystems Inc., Mountain View, CA. http://www.sun.com/sunsoft/neo/.

[38] P. Niemeyer and J. Peck. *Exploring Java.* O'Reilly & Associates, 1996.

[39] Martin Odersky. Pizza - a tasty complement to Java. University of Karlsruhe, Germany, 1996.

[40] Martin Odersky, Michael Philippsen, and Christian Kemper. EspressoGrinder Quelltext. University of Karlsruhe, Germany, 1995.

of Karlsruhe, Germany, 1996.

[42] Douglas C. Schmidt. *Object-Oriented Network Programming: An Overview of CORBA.*

[43] U. Schneider. Applets, schöne Applets. *iX*, pages 62–68, May 1996.

[44] U. Schneider. Vorgemahlen – Packages: Arbeit mit Javas Klassenbibliothek. *iX*, pages 142–152, June 1996.

[45] Silicon Graphics kritisiert Suns Pläne für die Java-Prozessoren. Computerwoche 18, 1996.

[46] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992.

[47] M. Stal. Frisch aus der Fabrik: JFactory 1.0. *iX*, pages 74–77, May 1996.

[48] Sun fine-tunes Java's future. Communications Week, no. 19, 1996.

[49] Sun kündigt eine Serie von Java-optimierten Chips an. Computerwoche 6, 1996.

[50] SunNews. Sun Microsystems Inc., Mountain View, CA. `http://www.sun.de`.

[51] Sun präsentiert auf Java optimierte low-cost prozessoren und embedded mikrokontroller. Sun GmbH, Deutschland, 1996. Presseinformation PR 01/96.

[52] Sun prepares a Java invasion. New Electronics, no. 13, February 1996.

[53] Pressemitteilungen. Sun GmbH, Deutschland.
`http://www.sun.de`.

[54] Sun to develope low-cost, fast chips for Java applications. Communications Week International, no. 159, 1996.

[55] Sunworld online news. Sun Microsystems Inc., Mountain View, CA.
`http://www.sun.com/sunworldonline`.

[56] J. Templ. Schwarzes Loch – Wie sicher ist Java wirklich? *iX*, pages 70–73, May 1996.

[57] Russell Rector und George Alexy. *THE 8086 BOOK.* McGraw-Hill Book Company, New York, 1980.

[58] The Unicode standard; worldwide character encoding, version 1.0.

[59] Peter van der Linden. How exceptions work. *JavaWorld, Nuts & Bolts*, June 1996.
`http://www.javaworld.com/javaworld/jw-06-1996/jw-06-exceptions.html`.

[60] Steve Vinoski. Distributed object computing with CORBA. *C++ Report Magazine*, July/August 1993.

[61] Wachstumsmarkt Java-prozessoren, Backgrounder. Sun GmbH, Deutschland, 1996.

[62] Michael Weiss, Andy Johnson, and Joe Kiniry. Distributed computing: Java, CORBA, and DCE. OSF Research Institute, February 1996.

[63] Frank Yellin. Low-level security in java. In *WWW4 Conference*, December 1995.
`http://java.sun.com/sfaq/security.html`.