Berichte zur Rechnerarchitektur
Band 3 Nr. 9 (1997)

Technischer Bericht

Java: Konzepte und Möglichkeiten

Michael M. Gutzmann
María Caridad González Nicolau
Andreas Klein René Scholz

Friedrich-Schiller-Universität Jena
Institut für Informatik
Lehrstuhl für Rechnerarchitektur und -kommunikation

Hrsg.: Prof. Dr. Ing. W. Erhard

FSU

Java: Konzepte und Möglichkeiten

Michael M. Gutzmann **María Caridad González Nicolau**
Andreas Klein **René Scholz**

Lehrstuhl für Rechnerarchitektur und -kommunikation, Universität Jena
D-07740 Jena, Germany
{gutzmann,cary,mckinley,mrz}@informatik.uni-jena.de
☎: +49/3641/946375 fax: +49/3641/946372

Juli 1997

Kurzfassung

Dieser Bericht beschreibt die Eigenschaften und Sprachkonstrukte der Programmiersprache Java. Die historische Entwicklung von Java wird ebenso beleuchtet wie die Unterschiede zu anderen Programmiersprachen.

1 Java Konzepte und wichtige Eigenschaften

Die wichtigsten Eigenschaften von Java gegenüber anderen Programmiersprachen werden von Sun Microsystems selbst mit folgendem Satz beschrieben:

> *Java ist: Eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutrale portable, hochleistungsfähige, Multithread-fähige und dynamische Sprache.*

1.1 Portabilität

Die Unabhängigkeit eines Java-Programms von der jeweils benutzten Rechnerarchitektur beruht hauptsächlich auf dem Konzept der *Java Virtual Machine*, kurz JVM.
Der Quelltext eines Java-Programms wird durch Aufruf des Java-Compilers `javac` in einen *Bytecode* (auch: J-Code) übersetzt, der aus Befehlen in Form von standardisiertem Programmcode für eine virtuelle Rechnerarchitektur besteht, der JVM [LY97].
Java ist daher sowohl eine Interpretersprache, als auch eine Compilersprache. Die JVM verhält sich für ein Java-Programm so wie ein normaler Prozessor, stellt dabei aber eine virtuelle und sichere Umgebung bereit, siehe Abschnitt 4.1. Diese Umgebung wird meist durch Software wie dem Java-Interpreter `java` realisiert. Da der Java-Interpreter relativ klein ist (zwischen 40 kB und 200 kB Speicherbedarf), werden daher in Zukunft verstärkt echte Java-Prozessoren [Jav97] eingesetzt werden, die den Bytecode direkt ausführen können (z. B. Sun's JavaStation [Sun97e]).
Da der Java-Bytecode von jedem Java-Interpreter ausgeführt werden kann, ist er daher von der jeweils darunter liegenden Rechnerarchitektur (Windows, Unix) unabhängig,

vorausgesetzt, die JVM wurde bereits auf diese Architektur portiert. Durch die Java-Sprachspezifikation ist die Größe der Basis-Datentypen und deren Arithmetik eindeutig festgelegt, es gibt keine implementierungsabhängigen Aspekte innerhalb der Sprachspezifikation. Java-Programme sind daher äußerst portabel, was den Aufwand und die Kosten für die Erstellung und Wartung im Vergleich zu herkömmlichen Software-Technologien senkt.

1.2 Sicherheitsmechanismen

Ein besonders wichtiger Punkt beim Entwurf der Java-Sprachspezifikation bestand in der Schaffung weitgehender Sicherheitsmaßnahmen bei der Ausführung von Java-Programmen. Diese sind insbesondere aufgrund der Möglichkeit der verteilten Ausführung von Java-Programmen im Internet notwendig, z. B. wenn Applets über das WWW geladen und vom Webbrowser ausgeführt werden. Es werden verschiedene Sicherheitsstufen bereitgestellt, um den Benutzer vor der Ausführung von „bösartigem" Programmcode zu schützen:

- Auf der untersten Ebene können Java-Programme keine Zeiger auf den Arbeitsspeicher des Rechners anlegen, die Grenzen von Arrays überschreiten oder Daten außerhalb der Array- oder String-Grenzen lesen. Durch die Unterbindung des direkten Zugriffs auf Systemdateien oder Speicherbereiche wird eine große Klasse von Sicherheitsattacken von vornherein ausgeschlossen.

- Als zweite Schutzmaßnahme wird vom Java-Interpreter beim Laden von unbekanntem Programmcode eine Überprüfung des Bytecodes durchgeführt, das *Bytecode Verifying*, s. a. Abb. 5. Dabei wird sichergestellt, daß der Bytecode richtig geformt ist, d. h. daß er keinen Über- oder Unterlauf des Stacks verursacht, keine illegalen Bytecodes enthält usw. Bei negativem Überprüfungsergebnis wird das Programm nicht gestartet und eine Exception ausgelöst.

- Eine weitere Sicherheitsschicht ist die "Sandbox" („Sandkasten"). Dabei wird unbekannter Programmcode oder ein Applet innerhalb einer geschützten Umgebung ausgeführt und kann nicht negativ auf die JVM einwirken. Dabei gelten eine Reihe von Einschränkungen, um diesen Schutz zu gewährleisten:
  - Der Programmcode hat keinerlei Zugriff auf das lokale Dateisystem.
  - Die Kommunikation mit fremden Rechnern kann beschränkt und überwacht werden.
  - Weitere Beschränkungen werden durch die `SecurityManager`-Klasse umgesetzt: Alle Java-Klassen, die sensitive Operationen durchführen, holen vorher vom Security Manager eine entsprechende Erlaubnis ein. Wenn die Operation direkt oder durch unbekannten Programmcode aufgerufen wird, wird vom Security Manager eine Exception generiert und die Operation nicht erlaubt.

- Ab Java 1.1 ist es möglich, den Java-Code mit einer digitalen Signatur zu versehen. Damit kann überprüft werden, ob der Code von einer Person oder Organisation stammt, der man genügend vertraut, um diesen Java-Code aus dem Netz zu laden und auszuführen. Der Programmcode wird dann ohne die Einschränkungen des Sandkasten-Modells ausgeführt.

1.3 Schnelligkeit und Multithreading

Da der Java-Compiler direkt binären Bytecode erzeugt, muß der Java-Interpreter (wie z. B. ein Webbrowser) kein Parsing und keine lexikalische Analyse durchführen. Wenn der Bytecode mittels eines zusätzlichen Schrittes noch weiter in Maschienencode umgesetzt wird (z. B. durch einen Just-in-Time–Compiler), erreicht man Ausführungsgeschwindigkeiten die mit denen normaler C++-Programme vergleichbar sind.

Da beim Entwurf von Java die modernen Fähigkeiten des Multithreadings implementiert wurden, kann Java grundsätzlich mehrere Operationen quasi-parallel ausführen, und bietet zur Kontrolle besondere, in der Syntax verankerte Keywords wie `synchronized` sowie die Klasse **thread**. Besonders bei grafischen Anwendungen (z. B. innerhalb von Webseiten) wird so das interaktive Verhalten für den Benutzer verbessert.

1.4 Verteilte Programmausführung

Java ist auch eine verteilte Sprache, die Netzwerke auf einer sehr hohen Ebene unterstützt. Beispielsweise können entfernte Dateien oder Ressourcen ähnlich einfach gelesen werden wie lokale Dateien. Ein Java-Programm kann dynamisch weitere Objektklassen aus dem WWW nachladen (wenn dieses erlaubt wurde). Mittels *Remote Method Invocation* (RMI) können verteilte Java-Anwendungen erzeugt werden, in denen die Methoden von entfernten Java-Objekten von anderen JVM's aufgerufen werden, die sich auf verschiedenen Rechnern befinden können. Weiterhin werden traditionelle Netzwerk-Mechanismen wie Sockets und Datagrams unterstützt.

1.5 Dynamisches Laden und Binden

Eine Java-Klasse kann jederzeit von einem laufenden Java-Interpreter geladen werden. Diese dynamisch geladenen Klassen können dann dynamisch instantiiert werden. Auch maschinenspezifische Codebibliotheken ("native code") können dynamisch geladen werden. Informationen über eine Klasse können ebenfalls dynamisch während der Laufzeit ermittelt werden. Zusammen mit der Möglichkeit der verteilten Programmausführung kann ein Java-Interpreter damit Programmcode aus verschiedenen Stellen des Internets laden und ausführen, dabei werden verschiedene Sicherheitsmaßnahmen beachtet.
Ein solches Programm kann sich an die Bedürfnisse des Benutzers anpassen und dabei die verteilten Ressourcen des Netzwerkes einsetzen.

1.6 Objektorientiertheit

Java ist eine (fast) vollständig objektorientierte Programmiersprache, die auf Klassen beruht und die grundlegenden Eigenschaften *Kapselung, Vererbung* und *Polymorphie* des OO-Konzeptes auf moderne und möglichst sichere Weise verwendet. Die *Klasse* ist die Basiseinheit der Übersetzung und Ausführung von Java-Programmen (welche ebenfalls Klassen sind).
Diese Klassen bilden eine gemeinsame Hierarchie, innerhalb der eine Unterklasse Eigenschaften und Funktionen von ihrer Oberklasse erben und diese erweitern und modifizieren kann. Dadurch wird die Wiederverwendbarkeit und die Strukturierung des Programmcodes wesentlich erleichtert.

Ein Objekt (eine konkrete Instanz einer Klasse) beinhaltet sowohl Daten als auch Methoden (Funktionsaufrufe), die mittels der Schlüsselworte `public, protected, package` und `private` als öffentliche oder geschützte Komponenten anderen Objekten zugänglich gemacht bzw. vor diesen verborgen werden können.
Im *Java Development Kit* (JDK) wird bereits eine Vielzahl von vorgefertigten Klassen mitgeliefert, die in sogenannten *Packages* bzw. Paketen zusammengefaßt sind (z. B. ist `java.util.zip` das Paket, welches die Klassen für verschiedene Kompressionsalgorithmen enthält).

1.7 Modularität, Erweiterbarkeit und Klassenhierarchie

Als objektorientierte Programmiersprache verfolgt Java ein strenges Klassenkonzept, in der alle Klassen in einer Baum-Hierarchie angeordnet sind, bei der die jeweils übergeordnete Klasse als Oberklasse bezeichnet wird und die untergeordneten, abgeleiteten Klassen entsprechend als Unterklasse. Der Vorgang des Ableitens einer Klasse wird auch als Vererbung bezeichnet, bei der die abgeleitete Klasse Methoden und Felder von ihrer Oberklasse erbt (mittels der `extends`-Klausel bei der Definition einer Klasse) und diese erweitert. Die Wurzel dieser Hierarchie ist in Java die Stammklasse `Object`.
In Abb. 1 ist ein Ausschnitt dieses Baums aller Java-Klassen des JDK zu sehen, u. a. auch die Klasse `ZipEntry` aus dem Paket `java.util.zip`, die Klassen `String` und `System` aus dem Paket `java.lang` und die Klasse `Applet` aus dem Paket `java.applet`.

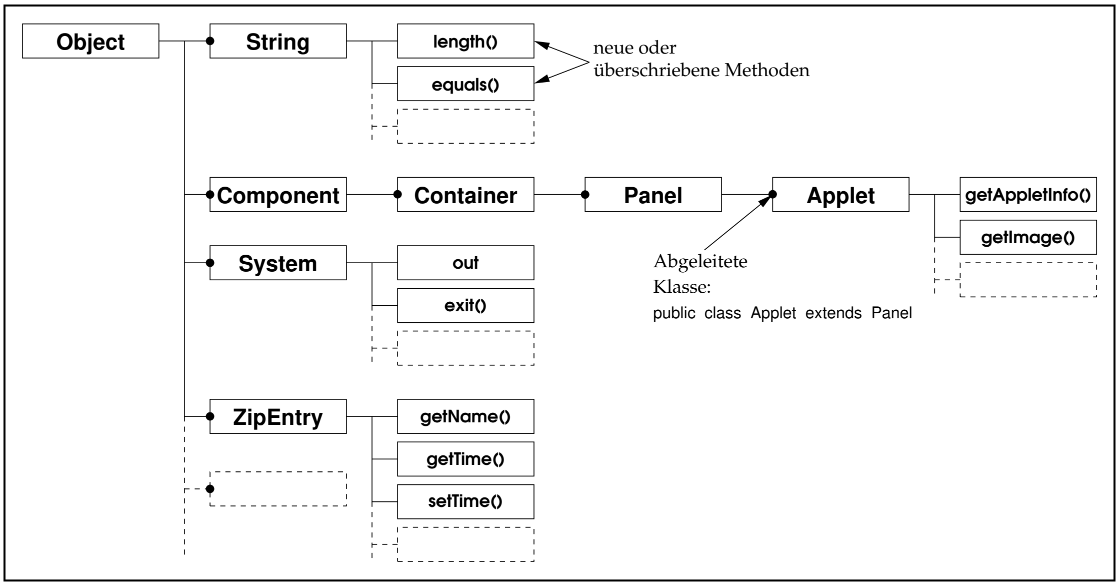

Abbildung 1: Ausschnitt des Klassenhierarchiediagramms von Java

Alle mit Java ausgelieferten Klassen sind übersichtlich in einer überschaubaren Anzahl von Paketen angeordnet, siehe dazu Tabelle 1.
Es ist aber auch möglich, selbsterstellte Klassenbibliotheken oder Pakete von anderen Herstellern einzusetzen und damit Java beliebig zu erweitern. Dazu muß lediglich die Umgebungsvariable `CLASSPATH` entsprechend erweitert werden, Beispiel: `CLASSPATH=/local/java/java/lib/classes.zip:.:~/java`.

Tabelle 1: **Vollständige Liste der Pakete von Java 1.1**

Paketname	Inhalt
`java.applet`	Applet-Klassen
`java.awt`	Grafik-, Fenster und GUI-Klassen
`java.awt.datatransfer`	Datenübertragungsklassen
`java.awt.event`	Klassen und Interfaces zur Ereignisbehandlung
`java.awt.image`	Bildverarbeitungsklassen
`java.awt.peer`	GUI-Schnittstellen zur Plattformunabhängigkeit
`java.beans`	JavaBeans-Komponentenmodell-API
`java.io`	Eingabe- und Ausgabeklassen
`java.lang`	Klassen der Kernsprache von Java
`java.lang.reflect`	Klassen der Reflection-API
`java.math`	Arithmetik beliebiger Genauigkeit
`java.net`	Netzwerkklassen
`java.rmi`	Klassen zum entfernten Aufruf von Methoden
`java.rmi.dgc`	RMI-verwandte Klassen
`java.rmi.registry`	RMI-verwandte Klassen
`java.rmi.server`	RMI-verwandte Klassen
`java.security`	Sicherheitsklassen
`java.security.acl`	Sicherheitsverwandte Klassen
`java.security.interfaces`	Sicherheitsverwandte Klassen
`java.sql`	JDBC+SQL-API für Datenbankzugriffe
`java.text`	Klassen zur Internationalisierung
`java.util`	Verschiedene nützliche Datentypen
`java.util.zip`	Klassen zur Komprimierung/Dekomprimierung

Alle Klassen innerhalb desselben Paketes enthalten in ihrem Quelltext als ersten Befehl ein `package`-Statement, z. B. für das Paket `java.util.zip` den Befehl `package java.util.zip;`.
Zur Vermeidung von Konflikten im Namensraum werden alle Elemente einer Klasse (die Datenfelder und Methoden) über einen eindeutigen, vollständig qualifizierten Namen angesprochen. Dieser besteht aus dem Paketnamen, dem Klassennamen und dem Namen des Elements, die alle durch einen Punkt voneinander getrennt sind. Die Paketnamen bestehen ebenfalls aus einer Reihe von durch Punkten getrennten Komponenten, welche die Verzeichnisstruktur abbilden, in der die Pakete vorliegen. Diese Paket-Hierarchie zur Bezeichnung der Klassen darf nicht mit der Klassen-Hierarchie selbst verwechselt werden.

Beispiel: In Java sind verschiedene Klassen zur Kompression von Daten in dem Paket `java.util.zip` enthalten, z. B. die Klasse `ZipEntry`, die verschiedene Informationen über die Datei-Einträge innerhalb eines ZIP-Archivs zurückgegen kann. Der vollständig qualifizierte Name für die Methode `getName()` dieser Klasse zur Anzeige des jeweiligen Dateinamens des ZIP-Archivs lautet: `java.util.zip.ZipEntry.getName()`.
Die in den Bytecode übersetzte Klasse `ZipEntry.class` befindet sich in dem Verzeichnis `java/util/zip`.
Um die Verwendung von Klassen zu vereinfachen, kann man nach Angabe des `import`-Statements (`import java.util.zip.*;`) in einem Java-Quelltext diese Klassen mit ihrem Kurznamen (`ZipEntry` anstelle von `java.util.zip.ZipEntry`) ansprechen.

Abb. 2 enthält alle Klassen aus dem Paket `java.util.zip` und zeigt für jede dieser Klassen aus welcher Oberklasse diese abgeleitet werden. Die dazu benötigten Oberklassen befinden sich in den Paketen `java.lang` und `java.io`. Die Klasse `Objekt` nimmt dabei eine Sonderstellung ein, da alle Klassen wie z. B. auch `IOException`, `InputStream`, `FilterInputStream` usw. direkt oder indirekt von `Object` abgeleitet werden. Die Pakete `java.lang` und `java.io` sind nur auschnittsweise dargestellt, insoweit sie für die Klassen aus `java.util.zip` benötigt werden.

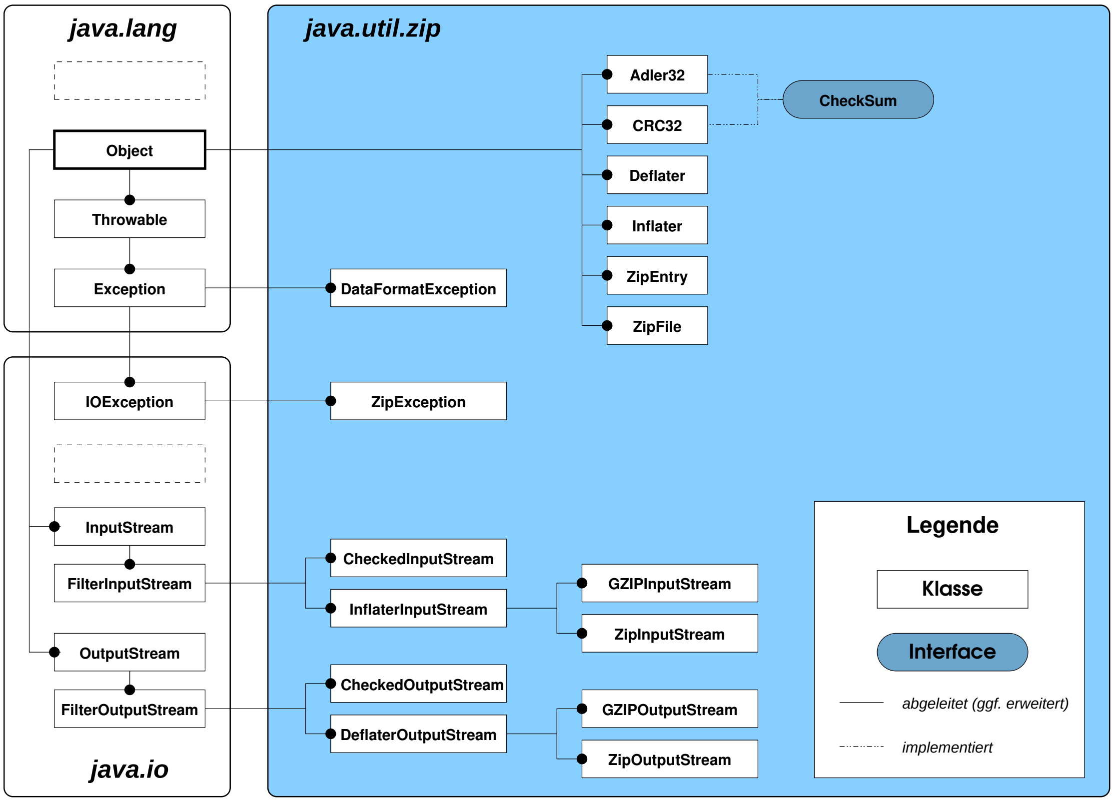

Abbildung 2: Aufbau des Java-Paketes `java.util.zip`

2 Weitere Java Charakteristika

Neben den oben aufgeführten Konzepten und wichtigen Eigenschaften besitzt Java noch weitere Charakteristika.

2.1 Einfachheit

Java ist in seiner Syntax an das weit verbreitete C++ angelehnt, was vielen Programmierern den Übergang zu Java erleichtert. Gleichzeitig wurde die Anzahl der Sprachkonstrukte relativ klein gehalten, wodurch Java schnell erlernt werden kann. Zusätzlich wurde auf einige Funktionen und Sprachkonstrukte, wie aus C/C++ bekannt, verzichtet. Dies betrifft insbesondere: Alle Pointer, die mehrfache Vererbung, das Überladen von Operatoren, den

C-Präprozessor, alle Header-Dateien sowie Befehle wie `goto`, `typedef`, `enum`, `union` und `struct`. Damit existieren viele der potentiellen klassischischen Fehlerquellen in Java nicht.

2.2 Robustheit

Ein weiteres Entwurfsziel von Java war die Entwicklung möglichst zuverlässiger und robuster Software. Bestimmte Arten von Programmierfehlern können so von vornherein vermieden werden. Java ist strenger typisiert als C++; der Compiler führt eine strenge Typprüfung beim Übersetzen des Quellcodes durch, weitere Überprüfungen erfolgen zur Laufzeit. Implizite Deklarationen von Methoden wie bei C sind in Java nicht erlaubt.
Da es keine Pointerarithmetik wie in C gibt, sondern nur *Objekte*, sind viele (mitunter schwer zu findende) Fehler mit Java gar nicht erst möglich. Die Grenzen von Arrays werden automatisch überprüft; das Überschreiben von Speicherbereichen ist nicht möglich.
Die automatische *Garbage Collection* gibt selbständig nicht mehr benötigten Speicher frei, wodurch die Entstehung von Speicherlecks und Fehler durch Speicherallokation und -freigabe vermieden werden.

2.3 Fehlerbehandlung durch Exceptions

Durch das *Exception Handling* kann die Fehlerbehandlung gegenüber herkömmlichen Programmiersprachen verbessert werden. Eine Exception signalisiert eine Ausnahmebedingung (z. B. „Datei nicht gefunden“). Durch die Anweisungen `try`, `catch` und `finally` können die entsprechenden Fehlerbehandlungsroutinen an einem Ort gruppiert werden, was die Behandlung und Wiederherstellung bei Fehlern stark vereinfacht und den Quelltext leichter lesbar macht, da der Programmcode nicht durch ständige Fehlerüberprüfungen unterbrochen wird. Weiterhin können Fehler an übergeordnete Methoden weitergereicht und an anderen Stellen behandelt werden, als sie ursprünglich aufgetreten sind.

2.4 Applets und Applikationen

Grundsätzlich gibt es zwei verschiedene Klassen von Java-Anwendungen: Applets und Applikationen. Dabei sind Applets „Mini-Anwendungen“, die von einem Browser o.ä. ausgeführt werden und strengen Sicherheitsbeschränkungen unterliegen, während eine Applikation als normales Java-Programm innerhalb der installierten Java-Umgebung auf dem Rechner ausgeführt wird.
In Tabelle 2 sind die wesentlichen Unterschiede zwischen Applets und Applikationen genauer aufgeführt.

Applet	**Applikation**
Ausführung durch Browser oder Appletviewer über eine HTML-Datei	Ausführung des Programms auf Shell-Ebene
Einschränkung durch Sandkasten-Modell; weniger bzw. keine Einschränkung für lokale bzw. signierte Applets	Keine Einschränkungen bei der Ausführung
Start über eine `start()`-Methode o.ä., die vom Browser aufgerufen wird	Start über die `main()`-Methode
Jedes Applet ist eine Unterklasse von `java.applet.Applet`	Definition einer eigenen „Hauptklasse“

Tabelle 2: Unterschiede zwischen Applets und Applikationen

2.5 Client-Server-Modelle

Java bietet mittels der Pakete **java.net** und **java.rmi** weitgehende Unterstützung zum Aufbau von Client-Server-Modellen an. Hierbei können verschiedene Kommunikationsmechanismen verwendet werden, insbesondere:

- Das Common Gateway Interface (CGI)
- Remote Methode Invocation (RMI)
- Sockets
- Servlets.

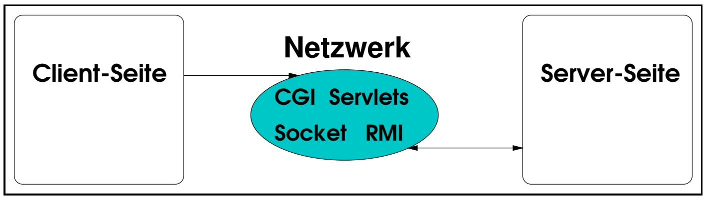

Abbildung 3: Client-Server-Modelle mit Java

Zu beachten ist dabei, daß Applets aufgrund der strengen Sicherheitsmaßnahmen („Sandkasten“) keinen direkten Zugriff auf Ressourcen des Servers erhalten dürfen und daher beispielsweise mit einer laufenden Applikation kommunizieren müssen.
Ein denkbares Szenario ist ein Applet, welches in einem Browser gestartet wird und eine Verbindung zu einer Applikation auf einem Datenbankserver herstellt und so die Dienste dieses Servers benutzen kann.

3 Entwicklung und Aufbau von Java

Die Entwicklung von Java erfolgte bei der Firma *Sun Microsystems* unter der Leitung der beiden bekannten Software-Experten James Gosling und Bill Joy. Dabei wurden verschiedene Konzepte aus den Programmiersprachen C, C++ und Objective C berücksichtigt.
In der Zeit von 1990 bis 1994 entstanden bei Sun Microsystems verschiedene Projekte mit der Zielrichtung Unterhaltungselektronik, insbesondere:

- das *Green*-Projekt,
- die Programmiersprache *Oak*,
- der Browser *WebRunner*.

Aus dem Browser WebRunner entwickelte sich schließlich der Browser *HotJava*, und aus Oak wurde Java, welches Ende 1995 offiziell eingeführt wurde [Sun97d, Par97b].
Mehr zu den Hintergründen der historischen Entwicklung ist in [Har97, Ban97] und [O'C97] zu finden.

Seit seiner Entstehung wurde Java und die damit verbundenen Produkte konsequent weiterentwickelt. Inzwischen liegt die Version 1.1 des *Java Development Kits* (JDK) vor, die

neue Schnittstellen für entfernte Objekte, Datenbankzugriffe, Kryptographie, Internationalisierung, Objektserialisierung und viele weitere Technologien enthält.
Java wird von allen wichtigen Hard- und Softwareherstellern sowie einigen Webbrowsern unterstützt[1]. Die JDK wird von Sun Microsystems für folgende Plattformen angeboten:

> SPARC Solaris 2.3-2.6, Intel x86 Solaris 2.5-2.6, Windows 95 / NT, Apple Macintosh.

Inzwischen existieren auch Portierungen von Java auf viele weitere Plattformen:

> Linux, AIX, DIGITAL OpenVMS, DIGITAL Unix, HP-UX, IRIX, NetWare, OS/2, OS/390, OS/400, SCO, UnixWare.

Positiv ist vor allem, daß Java *frei* ist, d. h. der Compiler `javac` wird kostenlos und mit kompletten Quellen ausgeliefert (für nichtkommerzielle Nutzung), ebenso der Rest des Java Development Kits: der Java-Interpreter `java`, der Debugger `jdb` und der `appletviewer`. Auch für die benötigten plattformabhängigen Bibliotheken ist der Quelltext zu den Bedingungen der *Java Internal Noncommercial Source License* [Sun97b] verfügbar, allerdings nur für SPARC Solaris und Windows. Es gibt aber auch bereits komplett freie Java-Implementierungen, die unter der GPL[2] stehen: *Kaffe OpenVM* [Tra97] und *Guavac* [Eng97].

Eine gute Einführung in die aktuelle Java-Version findet man in [Fla96].
Für die normale Arbeit mit Java ist die Binär-Version des JDK bereits ausreichend, die kommerzielle Verwendung von Java und JDK ist ebenfalls erlaubt [Sun97a].

3.1 Bestandteile des JDK

Das von Sun Microsystems ausgelieferte JDK umfaßt folgende Bestandteile:

- Verschiedene Programme wie den Compiler und Debugger,
- Plattformabhängige Bibliotheken zum Zugriff auf das Betriebssystem,
- Die Klassen der mitgelierten Pakete (siehe Tabelle 1) im Java-Bytecode,
- Den Java-Quelltext aller mitgelieferten Klassen,
- Die Unix-Manseiten der Programme,
- Weitere JDK-Dokumentation ist in einem extra Paket enthalten.

3.1.1 JDK-Werkzeuge

Die wesentlichen Werkzeuge, die mit dem JDK mitgeliefert werden, sind dabei:

- `javac`: Der *Compiler*. Übersetzt den Java-Quellcode in Bytecode, wobei jede im Quelltext definierte Klasse in eine eigene `class`-Datei übersetzt wird.

[1] Es ist jedoch zu beobachten, daß Microsoft eine etwas feindliche Haltung gegenüber Java bzw. Sun einnimmt.
[2] GPL: GNU Public License, s. a. [Fou97].

- `java`: Der *Java-Interpreter*. Interpretiert und verifiziert den Java-Bytecode der `class`-Dateien und führt die Java-Programme aus. Die angegebenen Parameter werden an die `main()`-Methode übergeben.
- `appletviewer`: Der *Applet-Viewer*. Lädt HTML-Dateien und startet die darin referenzierten Java-Applets. Der Applet-Viewer ermöglicht auch das Debuggen von Applets.
- `jdb`: Der *Java-Debugger*. Der `jdb` ist ein textbasierter Debugger ähnlich den bekannten Unix-Debuggern `dbx` und `gdb`. Beim Starten des in Java geschriebenen `jdb` wird eine weitere Kopie des Java-Interpreters `java` geladen. Dabei kann sich der `jdb` auch in bereits laufende Interpreter einbinden.

3.1.2 Weitere Java-Werkzeuge

Neben den bereits aufgeführten essentiell notwendigen Werkzeugen wie dem Java-Compiler `javac` gibt es noch eine Reihe weiterer Hilfsmittel zur Entwicklung von Java-Applikationen, die im JDK enthalten sind:

- `jar`: Das *Java Archive Tool*. Mit `jar` lassen sich komprimierte Archivdateien erzeugen, in denen Klassen und weitere Ressourcen abgelegt werden können. Die Verwendung von JAR-Archiven verringert die Ladezeit und ermöglicht das digitale Signieren von Applets.
- `javadoc`: Der *Java-Dokumentationsgenerator*. Mit `javadoc` kann man auf einfache Weise die benötigte API-Dokumentation im HTML-Format für Java-Quellcodes erzeugen. Dazu wird der spezielle Doc-Kommentar `/** */` von Java benutzt.
- `javakey`: Key-Management und digitale Signaturen. Mit dem Programm `javakey` können verschiedene Möglichkeiten zum Generieren von Keys und Zertifikaten, Verwalten der Key-Datenbank, Unterschreiben von JAR-Archiven bzw. Applets und dem Überprüfen von Zertifikaten bzw. signierten JAR-Archiven eingesetzt werden.
- `javap`: Der *Javaklassen-Disassembler*. Mit `javap` können die wesentlichen Informationen der `class`-Dateien ausgegeben und eine zusätzliche Klassenverifikation durchgeführt werden[3].
- Weitere, seltener benötigte Werkzeuge sind: `rmic, rmiregistry, serialver, jre, javah` und `native2ascii`.

Die vollständige Dokumentation aller Werkzeuge ist in den Unix-Manseiten oder in [Fla96] zu finden.

4 Die Komponenten von Java

In Abb. 4 sind die wesentlichen Komponenten bei der Anwendung von Java dargestellt. Mittels der verschiedenen APIs (Application Programming Interface) können die Basisklassen in Java-Programmen eingesetzt und die JVM angesprochen werden. Die Basisklassen

[3] Es gibt auch Disassembler, die beinahe den vollständigen Quellcode wiederherstellen können [Kou97].

und die JVM greifen über Betriebssystemfunktionen auf die darunter liegende Hardware zu, wobei auch Native-Code eingesetzt wird (Programmcode, der nicht als Bytecode vorliegt, sondern als plattformabhängiger Maschinencode).

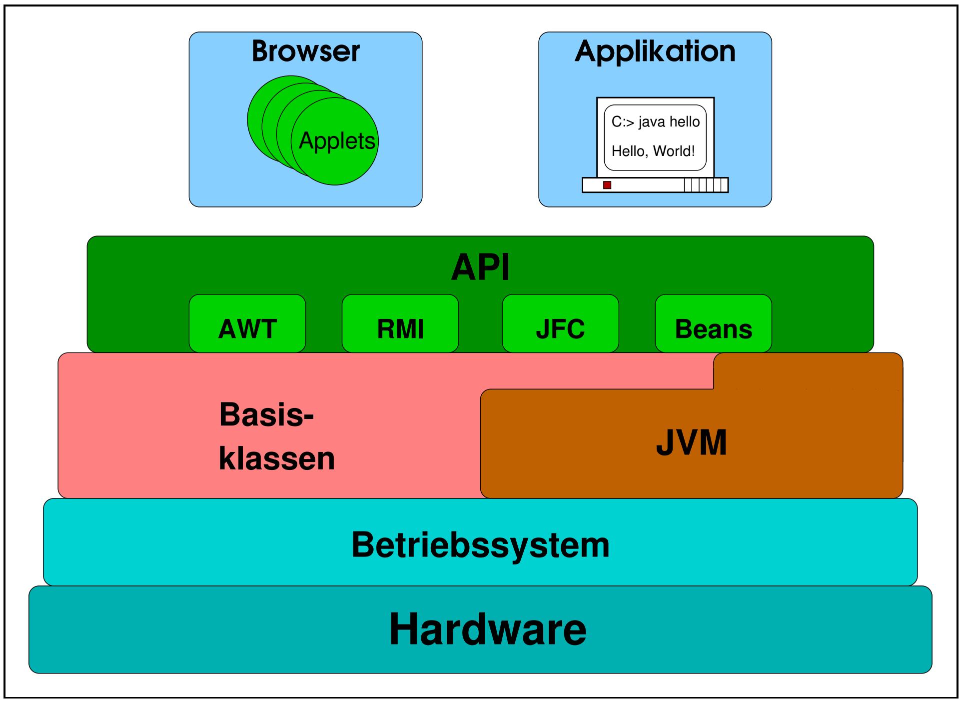

Abbildung 4: Prinzipielle Java-Komponenten

Eine fertige Java-Anwendung kann entweder als Applet innerhalb eines Browsers ausgeführt werden oder als eigenständige Applikation.[4] Dabei werden durch den Browser die JVM und die Basisklassen bereitgestellt, während eine Applikation direkt die vorhandene Java-Umgebung des JDK auf dem Rechner benutzt. Zur Ausführung werden alle notwendigen Klassen geladen (unbekannte Klassen müssen über den `CLASSPATH` verfügbar gemacht werden).

4.1 Die Java Virtual Machine

Wie bereits erwähnt, verhält sich die JVM für ein Java-Programm so wie ein normaler Prozessor, stellt dabei aber eine virtuelle und sichere Laufzeit-Umgebung für den compilierten Bytecode bereit. Von der JVM wird der stack-basierte Befehlsvorrat ausgeführt, die Speicheranforderungen bedient, die Basis-Datentypen werden erzeugt und behandelt sowie neuer Programmcode geladen und gestartet.
Die Spezifikation der JVM ist streng definiert und offengelegt, so daß eine eigene Implementierung dieser JVM von jedem vorgenommen werden kann. Es gibt keine undefinierten oder implementationsabhängigen Teile innerhalb von Java. Insbesondere ist die Größe und Funktionalität aller Basis-Datentypen genau vorgegeben.

[4] Dieser Unterschied muß im Quelltext entsprechend berücksichtigt werden.

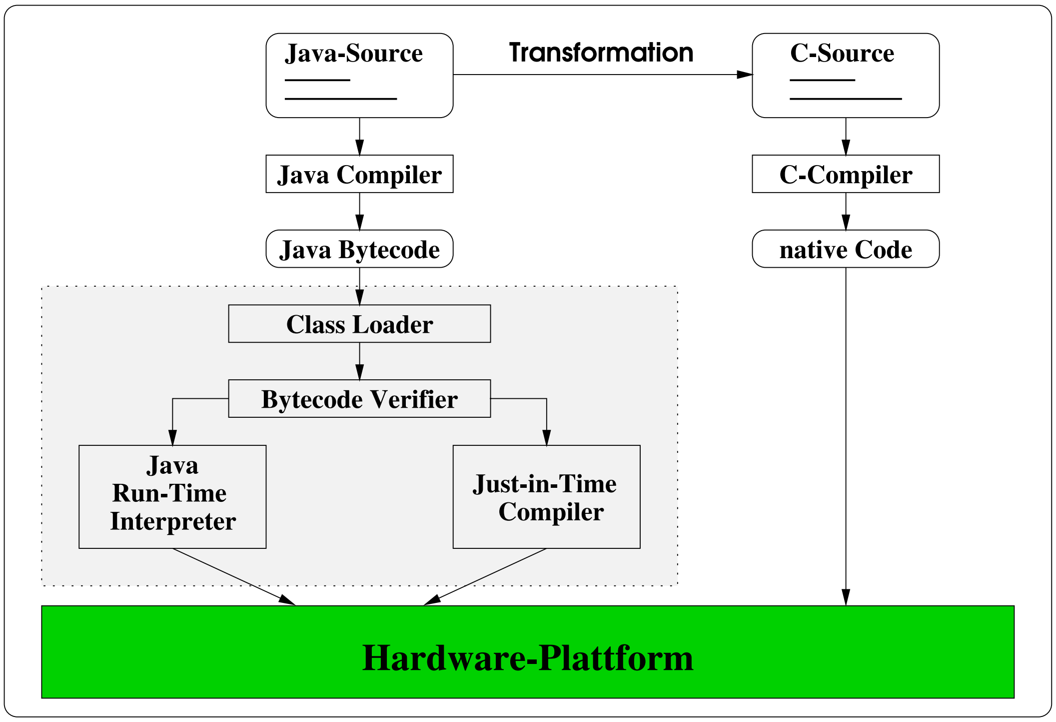

Abbildung 5: Übersetzung und Ausführung von Java-Programmen

4.1.1 Der Java-Interpreter

Der Java-Interpreter ist relativ klein und kann bei der Implementierung an die jeweilige Hardware-Plattform angepaßt werden. Auf den meisten Systemen ist dieser Interpreter in einer Sprache wie C oder C++ geschrieben, um schnelle, maschinen-spezifische Binaries des Compilers zu erhalten.
Der Interpreter kann als eigenständige Anwendung entwickelt werden oder auch innerhalb anderer Programme wie z. B. einem Browser eingesetzt werden.
Die Verwendung der JVM führt direkt zu garantiert portablem Programmcode. Eine Java-Applikation kann unverändert auf jeder Plattform ausgeführt werden, auf der die Umgebung der JVM bereitgestellt wird, s. a. Abb. 5.
Ein Programmierer ist also nicht gezwungen, seine Programme ständig an andere Plattformen anzupassen und muß auch nicht seine Quelltexte an die Endnutzer verteilen.

4.1.2 Der Java Just-In-Time–Compiler

Im allgemeinen sind Interpretersprachen im Vergleich zu Compilersprachen viel langsamer bei der Programmausführung. Da Java jedoch übersetzten Bytecode ausführt, ist Java eine sehr schnelle Interpretersprache. Darüberhinaus erlaubt Java die Implementierung von JVM-Umgebungen, die zusätzliche Steigerungen der Ausführungsgeschwindigkeit durch die interne Übersetzung des Bytecodes in maschinen-spezifischen Programmcode der jeweiligen Plattform erreichen [Cyg97]. Diese speziellen Übersetzer werden *Just-in-Time–Compiler* genannt und sind momentan vor allem in Webbrowsern anzutreffen.

Eine weitere Entwicklung sind Compiler, die aus bereits vorliegendem Java-Bytecode durch einmalige Übersetzung plattform-spezifischen, optimierten Maschinencode erzeugen, der noch schneller ist als bei Just-in-Time–Compilern [IBM97]. Aufgrund des Sprachdesigns von Java sind außerdem besondere Optimierungsmöglichkeiten während des Übersetzens gegeben, die in C oder C++ nicht möglich sind. Laut Angaben von Sun Microsystems ist die Geschwindigkeit dann fast so hoch wie bei traditionellen Sprachen wie z. B. C++.

4.2 Eine Alternative zur JVM

Eine weitere Möglichkeit ist die Transformation von Java-Quellcode in einen anderen Quellcode, wie z. B. C oder C++. Dieser neue Quellcode kann dann mittels normaler Compiler für diese Programmiersprache auf Plattformen übersetzt werden, für die noch keine Implementierung der JVM existiert (siehe Abb. 5).
Weiterhin kann so durch hochoptimierende Compiler die jeweilige Hardware-Plattform besser ausgenutzt werden. Nachteilig ist dabei allerdings, daß die Möglichkeit des dynamischen Ladens von Klassen verlorengeht.

4.3 Struktur eines Java-Programms

Die Erstellung des Quelltextes erfolgt entweder mit einem gewöhnlichen Editor oder durch Einsatz einer integrierten Entwicklungsumgebung für Java. Dabei bestehen folgende Vorgaben:

- Der Quelltext muß in einer Datei mit der Endung `.java` gespeichert werden.
- Die daraus erzeugten Dateien erhalten die Endung `.class`, zu jeder definierten Klasse wird eine `class`-Datei erzeugt (es findet damit kein Linken statt).[5]
- Der Compiler betrachtet eine Quelltextdatei als eine eigenständige Übersetzungseinheit, d. h. es ist in jeder `.class`-Datei genau eine Klasse vom Typ `public` erlaubt.
- Der Dateiname der Quelltextdatei muß den selben Namen erhalten wie die `public`-Klasse.

Die wichtigsten Punkte der Java-Sprachspezifikation (siehe [Fla96, Sun97c]) sind:

- Ein Java-Programm besteht aus einem oder mehreren der folgenden Teile:
  - einem `package`-Statement:

    Zusammenfassen von Klassen zu Paketen, innerhalb der bestimmte Sichtbarkeitstypen (`public, protected, package, private`) gelten.
  - `import`-Statements:

    Ansprechen der Klassen der Java-Pakete unter einem Kurznamen
  - `class`-Deklarationen:

    Definition von Klassen (deren Methoden und Elemente) und von lokalen Klassen.

[5] Dies ist abhängig vom verwendeten Compiler; inzwischen existieren auch Compiler, die darauf verzichten, jede Klasse in eine `class`-Datei zu überführen und stattdessen einen Link-Schritt durchführen.

- `interface`-Deklarationen:

  Zur Vererbung mehrerer `abstract`-Methoden an eine Unterklasse. Interfaces werden mittels der `implememts`-Klausel in einer Klassendeklaration implementiert, wobei die `abstract`-Methoden durch die Unterklasse selbst implementiert werden müssen.[6]

- **Datentypen:** In Java gibt es einfache Datentypen (`integer, boolean, char, floats`) und aus diesen zusammengesetzte komplexe Datentypen (Arrays, Klassen, Interfaces).

- **Syntax:** Der Compiler zerlegt den Quelltext in verschiedene Einheiten:

  - Bezeichner: Benennen Variablen, Funktionen, Klassen und Objekte. Dabei kann 16-Bit-Unicode verwendet werden.
  - Schlüsselwörter: Spezielle Bezeichner wie `for, break, import, public, while,` ...
  - Literale: (feste) Zahlen, Zeichen und String-Werte.
  - Operatoren: Numerische Operatoren, Vergleichsoperatoren usw.
  - Trennzeichen: Zeichen wie `? : , ;` usw.
  - Kommentare: Werden durch `/* */` oder `//` angegeben.

- **Statements:** Deklarationen, Ausdrücke, Exceptions (Ausnahmen), Verzweigungen.

- **Sichtbarkeit:** In Java existieren verschiedene Sichtbarkeitstypen von Klassenelementen: `public, protected, package` und `private`, s. a. Tabelle 3.

	Sichtbarkeit			
Erreichbar für	**public**	**protected**	**package**	**private**
Gleiche Klasse	ja	ja	ja	ja
Klasse in gleichem Paket	ja	ja	ja	nein
Unterklasse in anderem Paket	ja	ja	nein	nein
Keine Unterklasse, anderes Paket	ja	nein	nein	nein

Tabelle 3: Erreichbarkeit von Klassenelementen

In Abb. 6 ist der Java-Quelltext zur Ausgabe des Textes „Hello, World!“ dargestellt. Hier eine kurze Erläuterung der benutzten Befehle:

- Die `import`-Anweisung ist hier überflüssig, da das Paket `java.lang` von Java implizit *immer* importiert wird.

- Definition einer public-Klasse `HelloWorld`.

- Der Quelltext muß in einer Datei mit dem gleichen Namen (`HelloWorld.java`) gespeichert werden.

[6] Dies ersetzt die Mehrfachvererbung von C++.

- Diese Klasse enthält eine Methode `main()`, welche der Startpunkt für eine Java-Applikation darstellt.
- Im Array `args[]` sind die angegebenen Aufrufparameter des Java-Programms enthalten (insgesamt `args.length()` Parameter).
- Aus der Klasse `System` im Paket `java.lang` wird die Methode `println` aufgerufen (die allerdings in dem Objekt `out` existiert), um den String `"Hello, World!"` auf dem Bildschirm auszugeben. Ohne den Import des Paketes `java.lang` müßte man hier `java.lang.System.out.println` schreiben.

```
import java.lang.*;

/* A class with the first Java program */
public class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello, World!");  // the main program
  }
}
```

Abbildung 6: Java Applikation, Teil 1: Der Java-Quelltext

4.4 Übersetzung des Quelltextes

Anschließend an die Erstellung des Quelltextes erfolgt dessen Übersetzung durch den Java-Compiler `javac`. Dieser kann im einfachsten Fall als Java-Programm vorliegen, also als Bytecode der vom Java-Interpreter `java` ausgeführt wird (und ist daher wieder portabel).

```
$ javac HelloWorld.java
$
$ dir -l HelloWorld.*
-rw-r--r--   1 user       473 Jul  7 20:16 HelloWorld.class
-rw-r--r--   1 user       181 Jul  7 20:14 HelloWorld.java
```

Abbildung 7: Java Applikation, Teil 2: Die Übersetzung des Java-Quelltextes

Normalerweise existieren aber für die meisten Plattformen bereits Java-Compiler, die auf die jeweilige Rechnerarchitektur portiert wurden und daher wesentlich schneller arbeiten. Für die Übersetzung der Quelltexte spielt dies keine Rolle, nur die Ausführungszeit ist wesentlich kürzer. Der erzeugte Bytecode für die JVM ist in beiden Fällen identisch und von der jeweiligen Plattform *unabhängig*. Auf verschiedenen Plattformen wird also beim Übersetzen desselben Java-Programms ebenfalls identischer Bytecode erzeugt.
Bei herkömmlichen Compilern, wie z. B. C, ist das Ergebnis der Übersetzung an die gerade benutzte Hardware-Plattform gebunden und kann nur auf genau dieser Plattform ausgeführt werden. Die Abbildung 8 verdeutlicht diesen wesentlichen Unterschied beider Möglichkeiten.

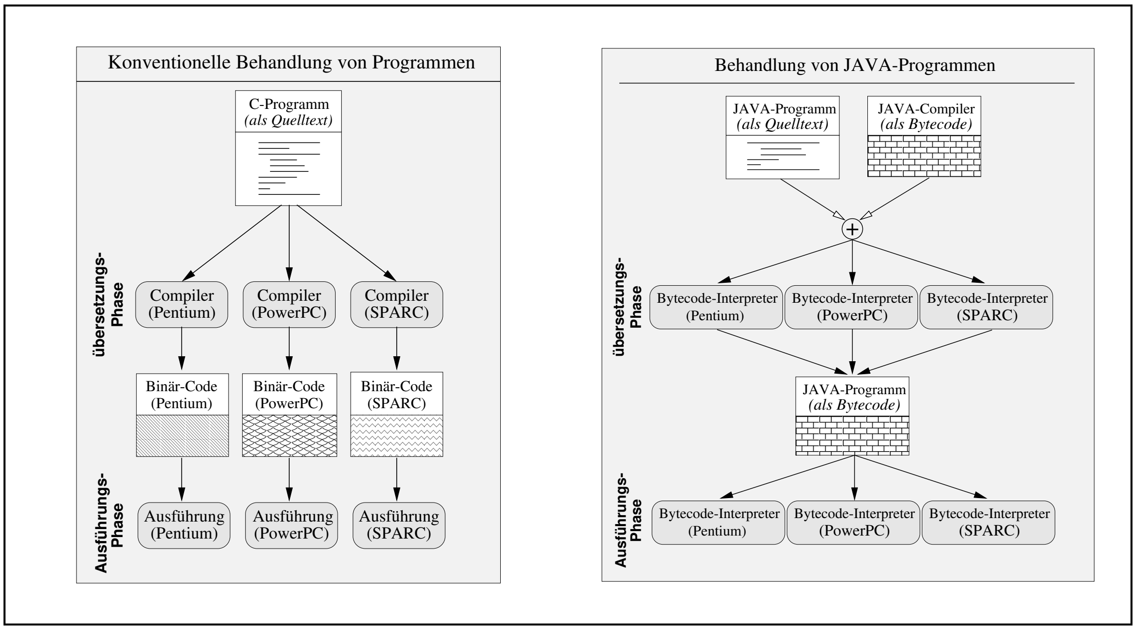

Abbildung 8: Übersetzung von Quelltexten

4.5 Ausführung des Bytecodes

Nachdem das Java-Programm in den Bytecode übersetzt wurde und als `class`-Datei vorliegt, kann diese mittels des Java-Interpreters `java`, durch welchen die JVM zur Verfügung gestellt wird, ausgeführt werden, indem die ausführbare Datei, wie in Abb. 9 gezeigt, an den Interpreter übergeben wird (ohne die Endung `.class`): `java program`.

```
$ java HelloWorld
Hello, World!
```

Abbildung 9: Java Applikation, Teil 3: Ausführung des Java-Bytecodes

Solche direkt ausführbaren und eigenständigen Java-Programme werden als Applikation bezeichnet.

Eine zweite Möglichkeit der Programmausführung ist das Erstellen von Applets. Dabei handelt es sich um Java-Programme, die innerhalb eines Webbrowsers oder eines Applet-Viewers ausgeführt werden. Diese Applets besitzen keine `main()`-Methode und können nur ausgeführt werden, indem sie in einer HTML-Datei mittels des `<APPLET>`-Tags referenziert werden und diese von einem Webbrowser oder Applet-Viewer angezeigt wird. Dabei wird dann die Applet-Klasse in den laufenden Java-Interpreter des Browsers geladen und ausgeführt, s. a. Abb. 10.
In Abb. 11 ist der Quelltext eines einfachen Applets zu sehen, der die gleiche Funktion hat wie die HelloWorld-Applikation.
Hier eine kurze Erläuterung der benutzten Befehle:

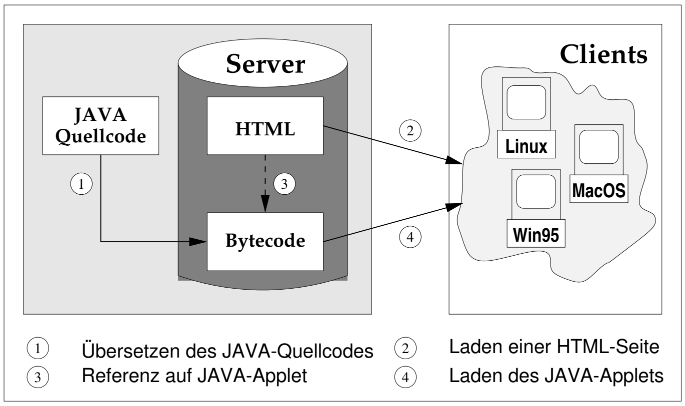

Abbildung 10: Einbettung eines Applets in HTML

- Die `import`-Anweisungen sind notwendig, um die Kurznamen `Applet, Graphics` usw. verwenden zu können.
- Jedes Applet ist eine Unterklasse von `java.applet.Applet`, die hier mit `public class HelloApplet extends Applet` erzeugt wird.
- Der Methode `paint()` wird vom Browser aufgerufen und ist die einzige Aktion in diesem Applet.

```
import java.applet.*;   // find Applet class
import java.awt.*;      // find Graphics class

/* A class with the first Java applet */
public class HelloApplet extends Applet {
    private Font font;

    /* hook to initialize the applet */
    public void init () {
        font = new Font("Serif", Font.BOLD, 30);
    }

    /* redraw the Applet */
    public void paint (Graphics g) {
        this.setBackground(Color.gray);
        g.setFont(font);
        g.drawString("Hello, World!", 30, 50);
    }
}
```

Abbildung 11: Java Applet, Teil 1: Der Java-Quelltext des Applets

- Die Methode `init()` wird ebenfalls vom Browser aufgerufen, allerdings nur einmalig beim Start des Applets. In dieser Methode wird ein größerer Textfont für die Ausgabe erzeugt.

- Die `paint()`-Methode verwendet die `Graphics`-Klasse, um die Hintergrundfarbe und den aktuellen Textfont zu ändern und den Text „Hello, World!“ an einer bestimmten Position innerhalb des Appletbereichs im Browser auszugeben.

Die Übersetzung des Quelltextes erfolgt in derselben Weise wie bei einer Applikation, siehe Abb. 12.

```
$ javac HelloApplet.java
$
$ dir -l HelloApplet.*
-rw-r--r--   1 user       762 Jul  7 20:55 HelloApplet.class
-rw-r--r--   1 user       484 Jul  7 20:30 HelloApplet.java
```

Abbildung 12: Java Applet, Teil 2: Die Übersetzung des Java-Quelltextes

Das Ergebnis der Ausführung des HelloWorld-Applets im Browser Netscape (v.4) ist in der Abb. 13 zu sehen.

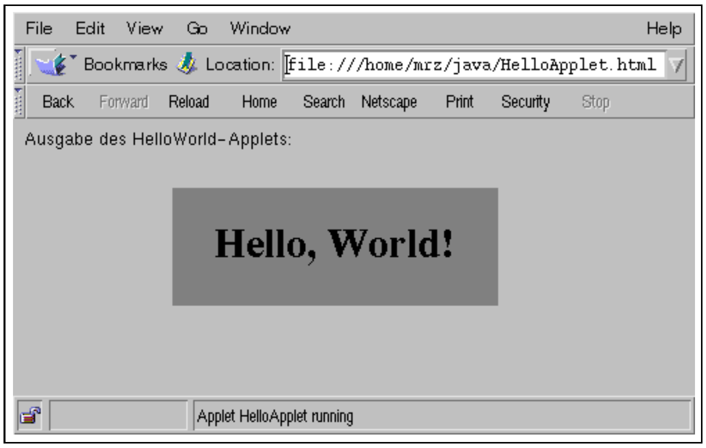

Abbildung 13: Java Applet, Teil 3: Ausführung des Applets

5 Java im Vergleich mit HTML

Die Darstellung von Information im WWW erfolgt über die Markup-Sprache HTML, welches eine Abkürzung von *Hypertext Markup Language* ist [Par97a]. Durch dieses Markup wird lediglich die logische Bedeutung der verwendeten „Befehle“ festgelegt. Die konkrete Darstellung bleibt dem Browser überlassen.
Im Zuge der zunehmend kommerziellen Nutzung des WWW reichte diese einfache Inhaltsbeschreibung nicht mehr aus. Infolgedessen wurden in den späteren Versionen HTML-3.2

und HTML-4.0 neue Möglichkeiten zur Verfügung gestellt, um wirkungsvolle Layouts zu realisieren. Eine weitere Möglichkeit ist die Verwendung von *Style Sheets* [Lie97].
Doch auch mit diesen Neuerungen bleibt ein ernsthafter Mangel: Die dargestellten Informationen sind rein statisch und ermöglichen keine Interaktion mit dem Benutzer. Daher wurden verschiedene Möglichkeiten entwickelt, um dynamische HTML-Seiten bereitzustellen:

- CGI-Skripte, die meist in Skript-Sprachen wie Perl oder Python geschrieben werden,
- *JavaScript*, *Active-X* u. a. zur Ausführung von bestimmten Aktionen im Browser,
- Erweiterung der WWW-Server durch Möglichkeiten zur automatischen Veränderung von HTML-Seiten bzw. zur Veränderung der Seiten in Abhängigkeit von Benutzeraktionen, z. B. *Server-Side Includes* (SSI) oder *Dynamisches HTML*,
- Benutzung von (proprietären) *Plugins* zum Verarbeitung und Anzeige spezieller Daten (z. B. Videos, Audio-Dateien usw.).

Die weitaus größten Möglichkeiten um das Layout zu beeinflussen, Benutzereingaben zu verarbeiten und die Webseiten dynamisch zu gestalten bleiben jedoch Java vorenthalten. Der Einsatz von Java innerhalb von Webbrowsern erfolgt in Form von *Applets*, die im Gegensatz zu den Plugins auf jeder Hardware-Plattform von jedem Java-fähigen Browser ausgeführt werden können.
Die Vorteile gegenüber reinem HTML sind hauptsächlich:

+ frei programmierbare multimediale Anwendungen,
+ beliebige Textdarstellungen, auch bewegt,
+ zeit-dynamische Darstellungen bei Grafiken und Bildern,
+ Darstellung von Animationen und 3D-Objekten,
+ das Layout kann vom Applet beliebig festgelegt und verändert werden,
+ verbesserte Interaktion mit dem Benutzer,
+ Rechenlast kann vom Server auf den Client verlagert werden,
+ einfache Integration von Audio- und Video-Anwendungen.

Demgegenüber existieren folgende Nachteile:

– längere Ladezeit der Applets,
– ein Java-fähiger Browser wird benötigt,
– viele Browser verfügen derzeit meist nur über die ältere Java-Version 1.0,
– die Sicherheitsmaßnahmen schränken die möglichen Anwendungen ein.

Java ist eine plattformunabhängige und objektorientierte Programmiersprache, die im Gegensatz zu CGI-Skripten die Ausführung von Aktionen auch auf dem Client-Rechner gestattet (unter Beachtung strenger Sicherheitsmechanismen).

6 Java im Vergleich mit C und C++

Java erinnert in seiner Syntax stark an C und C++, um eine leichte Erlernbarkeit zu ermöglichen. Die wichtigsten Ähnlichkeiten und Unterschiede sollen hier kurz angesprochen werden.
Die Gemeinsamkeiten mit C bzw. C++ sind:

- lokale Variablendeklarationen sind erlaubt: `for(int i=0, ...) { ...; }`,
- mehrdimensionale Arrays sind möglich: `int A[][]=new[10][4]`,
- fast alle standardmäßigen Operatoren werden von Java unterstützt (mit gleichem Vorrang und Assoziativität): `++, !, <<, ==, &, &&, ? :` usw.,
- die meisten Steueranweisungen sind gleich oder ähnlich: `do/while, for, if/else, switch, break, continue`...,
- Methoden können rekursiv aufgerufen werden,
- das Überladen von Methoden ist möglich:

```
public Draw(int x, int y) { ...; }
public Draw(int x, int y, int z) { ...; }
```

Die Unterschiede zu C bzw. C++ sind:

- in Java existieren keine globalen Variablen oder Funktionen (etwas ähnliches sind *statische* Klassenvariablen- oder methoden),
- in Java existieren kein Präprozessor und kein `#include`, sowie keine Templates,
- in Java ist keine Mehrfachvererbung möglich,
- in Java wird nicht mehr zwischen der Deklaration und der Definition von Variablen oder Prozeduren unterschieden,
- die einfachen Datentypen in Java sind genau definiert bzgl. Größe, Standardwert und Maximalwerten,
- in Java ist eine Dereferenzierung von Datentypen nicht möglich (und auch nicht notwendig),
- in Java werden einfache Datentypen immer "by value" übergeben, Arrays und Objekte immer "by reference",
- in Java gibt es keine Zeiger und keine Zeigerarithmetik, die Referenzierung und Dereferenzierung von Objekten geschieht automatisch,
- Es gibt kein `free()` oder `delete()` zur expliziten Freigabe von Objekten, dieses wird automatisch durch eine Garbage Collection erledigt,
- Strings sind keine Zeichen-Arrays, sondern Instanzen der Klasse **java.lang.String**,

- durch die Verwendung der *Unicode*-Kodierung in Java ist eine Internationalisierung (Anpassung an andere Sprachen) leicht möglich,
- das Überladen von Operatoren ist nicht möglich,
- es existieren verschiedene Modifikatoren für Variablen-, Klassen-, Interface- oder Methodendeklarationen: `final, native, synchronized, transient, volatile`,
- es gibt keine Typen `struct`, `union` oder `enum`. Diese Datenstrukturen können über Klassen und Subklassen simuliert werden, die darüberhinaus über mehr Features verfügen.

Der Vergleich macht deutlich, daß sich Java durch den Verzicht auf fehleranfällige Sprachfeatures von C und C++ und die Verwendung moderner Konzepte des Compilerbaus zu einer sicheren, leicht zu erlernenden Sprache entwickelt hat. Aufgrund der Plattformunabhängigkeit werden die Anwendungsbereiche von Java und die Unterstützung durch die Softwareindustrie mehr und mehr zunehmen.

7 Zusammenfassung

In diesem Bericht wurden die grundlegenden Konzepte und Charakteristika und die fortschreitende Entwicklung der Programmiersprache Java umfassend vorgestellt. Dabei wurde näher auf die Vorteile von Java zur Erstellung von Client-Server-Anwendungen eingegangen. Die komplexen Sicherheitsmechanismen und die wesentlichen Komponenten von Java sowie die Entwicklung von Java-Anwendungen wurden umfassend demonstriert. Anhand von Beispielen zur Erstellung von WWW-Applikationen und auf der Basis eines Vergleichs von Java mit herkömmlichen Programmiersprachen wurde die Leistungsfähigkeit von Java auf mehreren Ebenen verdeutlicht.

Literatur

[Ban97] David Bank. The Java Saga, July 1997. URL: `http://www.wired.com/wired/3.12/features/java.saga.html`.

[Cyg97] Cygnus Solutions. Cygnus Implementation Plans for the JavaTM Platform, 1997. URL: `http://www.cygnus.com/news/gcj-980907.html`.

[Eng97] David Engberg. Guavac home page, July 1997. URL: `http://http.cs.berkeley.edu/~engberg/guavac/`.

[Fla96] David Flanagan. *Java in a Nutshell*. O'Reilly, 1996.

[Fou97] Free Software Foundation. GNU General Public License, July 1997. URL: `http://http://www.fsf.org/copyleft/gpl.html`.

[Har97] Elliotte Rusty Harold. Where did Java come from?, July 1997. in: The comp.lang.java FAQ. URL: `http://sunsite.unc.edu/javafaq/javafaq.html#xtocid190291`.

[IBM97] IBM. alphaWorks Technologies by IBM, 1997. URL: `http://www.alphaworks.ibm.com/formula/`.

[Jav97] Inside Java. The microJava 701 chip, 1997. URL: http://www.inside-java.com/articles/javachips/microjava.htm.

[Kou97] Pavel Kouznetsov. Jad - the fast Java Decompiler, 1997. URL: http://web.unicom.com.cy/~kpd/jad.html.

[Lie97] Håkon Wium Lie. Web Style Sheets, July 1997. URL: http://www.w3.org/Style/.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[O'C97] Michael O'Connell. Java: The inside story, July 1997. URL: http://www.sunworld.com/swol-07-1995/swol-07-java.html.

[Par97a] Hubert Partl. HTML-Einführung, July 1997. URL: http://www.boku.ac.at/htmleinf/hein.html.

[Par97b] Hubert Partl. Was ist Java?, July 1997. URL: http://www.boku.ac.at/javaeinf/jein1.html#was.

[Sun97a] Sun Microsystems, Inc. Frequently Asked Questions About Licensing Java, 1997. URL: http://java.sun.com/nav/business/license-faq.html.

[Sun97b] Sun Microsystems, Inc. Java Development Kit Source License Agreement Form, 1997. URL: http://java.sun.com/nav/business/source_form.html.

[Sun97c] Sun Microsystems, Inc. Java Documentation, 1997. URL: http://java.sun.com/docs/.

[Sun97d] Sun Microsystems, Inc. Java Technology Home Page, 1997. URL: http://java.sun.com/.

[Sun97e] Sun Microsystems, Inc. JavaStation - An Overview, 1997. URL: http://www.sun.de/Produkte/Whitepapers/JavaStation/javast_ch6.html.

[Tra97] Transvirtual Technologies, Inc. Kaffe, 1997. URL: http://www.transvirtual.com/kaffe.html.