Einführung in Java

Dieter Korus
Philipps-Universität Marburg
FB Mathematik, FG Informatik
Version 1.0.2, April 1997

1. Was ist Java?

Java ist eine imperative, objekt-orientierte Programmiersprache mit vielen weiteren Eigenschaften.

Java ist eine von Grund auf objektorientierte Programmiersprache. Objektorientierte Konzepte wurden nicht wie z.B. in C++ oder Borland-Pascal nachträglich zu einer bestehenden Programmiersprache hinzugefügt, sondern sind integraler Bestandteil der Sprache selbst. Auf die wichtigsten Bestandteile objektorientierter Sprachen wird kurz in Abschnitt 2 eingegangen.

Java bietet darüber hinaus noch weitere Merkmale wie Ausnahmebehandlung, Nebenläufigkeit, Netzwerkfähigkeit und die Möglichkeit sog. Applets, d.h. Anwendungen für entsprechend ausgestattete HTML-Browser, zu schreiben. Hierauf wird in Abschnitt 3 hingewiesen.

Java ist plattformunabhängig, d.h. in Java geschriebene Programme werden in sogenannten *Bytecode* (Stackcode) übersetzt, der dann vom Java-Interpreter der jeweiligen (Ziel-)Plattform (egal ob Unix-Workstation, Macintosh oder PC mit Microsoft-Betriebssystem) interpretiert werden kann. Der Bytecode enthält Maschinenbefehle eines virtuellen Prozessors. Die Entwicklung von Java-Programmen auf verschiedenen (Entwicklungs-)Plattformen wird in Abschnitt 4 beschrieben.

In Abschnitt 5 erfolgt eine Kurzbeschreibung der Syntax von Java. Java hat teilweise ähnliche oder gleiche Syntax bzw. Semantik wie C(++), es gibt aber einige, zum Teil erhebliche Unterschiede. Einige Unterschiede von Java zu C++ werden in Abschnitt 6 aufgezählt.

2. Objektorientierte Prinzipien und ihre Ausprägung in Java

2.1 Abstrakter Datentyp

Der objektorientierten Programmierung liegt als ein wesentlicher Bestandteil der *abstrakte Datentyp* zugrunde. Ein abstrakter Datentyp besteht aus (generischen) *Typen*, *Funktionen*, *Bedingungen* und *Axiomen* den Datentyp betreffend. Im folgenden sei ein Beispiel für einen abstrakten Datentyp Stack gegeben:

Typ	Stack[ElemTyp]	*parametrisierbarer, generischer Typ*
Funktionen	Empty: → Stack[ElemTyp] Is_empty: Stack[ElemTyp] → boolean Push: ElemTyp × Stack[ElemTyp] → Stack[ElemTyp] Pop: Stack[ElemTyp] → Stack[ElemTyp] Top: Stack[ElemTyp] → ElemTyp	*Konstruktor* *Anfrage* *Modifikatoren*
Bedingungen	PRE Pop(St)=NOT Is_empty(L) PRE Top(St)=NOT Is_empty(L)	*Vorbedingungen*
Axiome	Is_empty(Empty()) NOT Is_empty(Push(Elem, St)) Top(Push(Elem,St))=Elem Pop(Push(Elem,St))=St	

Funktionen werden weiterhin unterschieden nach *Konstruktoren* zur Erzeugung (und gegebenenfalls Initialisierung) eines Datentyps, nach *Anfragefunktionen*, die einen Zustand des Datentyps abfragen, und nach *Modifikatoren*, die den Zustand des Datentyps ändern können.

2.2 Objekte, Klassen, Attribute und Methoden

In objektorientierten Sprachen wie Java werden abstrakte Datentypen als *Klassen* (engl. *classes*) implementiert. Die *Ausprägungen* (*Instanzen*, engl. *instances*) einer Klasse heißen *Objekte* (engl. *objects*).

Je nach Programmiersprache (Smalltalk, Eiffel, Java, C++,...) wird der abstrakte Datentyp unterschiedlich umgesetzt. In Java bestehen Klassen aus *Attributen* (Datenfeldern, engl. *fields*) und Funktionen (oder Prozeduren). Die Attribute beschreiben den Zustand eines Objekts. Funktionen (bzw. Prozeduren) werden unterschieden nach *Konstruktoren* und sog. *Methoden*, einer etwas unglücklichen Bezeichnung für Anfragen und Modifikatoren. Bedingungen und Axiome sind in Java (leider!) nicht implementiert.

Objekte werden in Java i.a. durch Aufruf eines *new-Operators* instanziert. Der benötigte Speicher wird dabei automatisch allokiert (Gott sei Dank: kein `malloc(...)`!).

2.3 Kapselung

Ein Vorteil objektorientierter Programmiersprachen ist die Möglichkeit der *Kapselung* von Attributen und Methoden, d.h. es sollte die Möglichkeit bestehen Daten und einige Methoden vor dem Anwender der Klasse bzw. des Objekts zu verstecken. Dies ist je nach Programmiersprache unterschiedlich realisiert. In Java gibt es die folgenden vier Möglichkeiten (Syntax s.u.):

- Ist ein Attribut oder eine Methode als *privat* (engl. *private*) deklariert, so kann es/sie nur in der Klasse selbst benutzt werden.
- Ist ein Attribut oder eine Methode als *geschützt* (engl. *protected*) deklariert, so kann es/sie in der Klasse selbst und im gleichen Paket (s.u.) benutzt werden. Außerdem kann auf ein geschütztes Attribut einer Klasse durch solche Objektreferenzen zugegriffen werden, die mindestens vom selben Typ wie die aufrufende Klasse sind. Geschützte Methoden kommen insbesondere bei der Vererbungen zur Geltung und sollen den Entwickler unterstützen.
- Hat ein Attribut oder eine Methode keine der oben angegebenen Zugriffdeklarationen, so ist es/sie in der Klasse selbst sowie in allen Unterklassen des gleichen Pakets benutzt werden.
- Ist ein Attribut oder eine Methode als *öffentlich* (engl. *public*) deklariert, so kann es/sie von außerhalb der Klasse direkt angesprochen werden (insoweit die Klasse dort selbst sichtbar ist).

Die Verwendung dieser Modifier hat weiterhin Einfluß auf die Erweiterung von Klassen sowie auf Objektreferenzen innerhalb der Klassenhierarchie (s.u.).

Es gehört zu einem guten Programmierstil, sämtliche Attribute zu verstecken und nur über Methoden auf diese zuzugreifen. Dadurch kann sichergestellt werden, daß die Zustände der Objekte nur in kontrollierter Art und Weise geändert werden.

2.4 Klassenattribute und -methoden

Attribute gehören normalerweise zu einem Objekt, da sie dessen Zustand beschreiben, und sollten nur von und in diesem geändert werden können. Es gibt aber Fälle, in denen ein Attribut nicht nur ein Objekt betrifft, sondern die ganze Klasse von Objekten (eigentlich eine Menge von Objekten, aber hier wird die Klassen-Objekt-Beziehung etwas aufgeweicht). Man denke sich z.B. eine Firma, die ihre Produkte (die einzelnen Objekte) bei der Herstellung der Reihe nach mit Seriennummern versieht. Die zuletzt vergebene Seriennummer ist sicherlich kein Attribut der einzelnen Objekte. Um nicht eine eigene Klasse für die Menge aller Produkte - mit den Attributen aktuelle Seriennummer und Liste der Produkte - zu erzeugen, wird in Java das sogenannten *Klassenattribut* (in Java auch etwas verwirrend *statisches Attribut* genannt, engl. *static field*) eingeführt. Dieses Attribut gilt global für alle Objekte der Klasse und ist besonders gekennzeichnet (Syntax s.u.).

Aber nicht nur Attribute, sondern auch Methoden können zu *Klassenmethoden* deklariert werden (Z.B. die Klassenmethoden, die auf die Klassenattribute zugreifen). Diese sind dann vergleichbar mit Prozeduren wie z.B. aus Pascal bekannt.

In einer Objektmethode ist es stets möglich die Klassenmethoden und -attribute der zugehörigen Klasse anzusprechen. Umgekehrt ist es aber nicht möglich in einer Klassenmethode, Methoden und Attribute von Objekte der Klasse direkt anzusprechen. Klassenmethoden besitzen nur Zugriff auf Klassenattribute und -methoden.

2.5 Zuweisung, Kopie und Gleichheit von Objekten

Je nach objektorientierter Programmiersprache haben *Zuweisung*, *Kopie* und *Gleichheit* von Objekten bzw. Objektvariablen eine andere Semantik. In Java liegt die *Referenzsemantik* vor, d.h. Objektvariable in Java sind genau genommen Referenzen (Zeiger, Pointer) auf einen Speicherbereich, in dem die eigentlichen Objekte liegen. Das hat gewisse Konsequenzen für die genannten Operationen.

Weist man in Java einer Objektvariablen O1 eine andere Objektvariable O2 (gleichen Typs) zu, so wird nicht der Inhalt (Zustand) kopiert, sondern die beiden Objektvariablen haben die gleiche Referenz! Ändert man daher ein Attribut in O2, so ändert sich automatisch das entsprechende Attribut in O1, und umgekehrt. (Eigentlich ändert sich dann gar nichts mehr, denn es ist ja dasselbe Objekt.) Zum Kopieren eines Objektes bedarf es also einer speziellen Operation. Dies ist anders in objektorientierten Programmiersprachen mit Wertesemantik (wie z.B. Borland-Pascal).

Ebenso wird beim Vergleich von zwei Objektvariablen in Java nachgeprüft, ob es sich um dieselbe Referenz handelt (und nicht, ob es sich um den gleichen Zustand der referenzierten Objekte handelt).

2.6 Klassenhierarchien und Vererbung

In vielen Anwendungen kommt es vor, daß man ähnliche Klassen (Typen) von Objekte hat, die sich nur in gewissen Eigenschaften unterscheiden. Diese lassen sich durch sog. *Klassenhierarchien* organisieren. Dabei ist es möglich die Eigenschaften (Attribute und Methoden) einer Klasse (der *Oberklasse*) in eine andere Klasse (die *Unterklasse*) zu übernehmen und letztere nach Bedarf zu erweitern. Man sagt, die Oberklasse *vererbt* ihre Eigenschaften an die Unterklasse, oder m.a.W. sie wird zur Unterklasse *erweitert*. Zu einer Klasse können mehrere Unterklassen definiert werden. Eine Unterklasse kann wieder Oberklasse einer weiteren Unterklasse sein. Es ist nicht nötig, die Attribute und Methoden der Oberklasse in der Unterklasse nocheinmal zu definieren. Ziel der Klassenerweiterung sollte es sein *Is-a-Beziehungen* zu modellieren.

Nun kann es natürlich passieren, daß gewisse Konstruktoren oder Methoden in einer Unterklasse eine andere Implementierung oder Aufrufsform (z.B. andere Parameter) benötigen. In diesem Fall besteht die Möglichkeit, die Methoden der Oberklasse zu *überschreiben* (engl. *overriding*).

Ein Objekt einer Klasse kann auch in die Rolle eines Objeks einer Oberklasse seiner Klasse schlüpfen. Insbesondere kann eine Objektvariable der Oberklasse auf ein Objekt der Klasse verweisen. Man nennt dies auch *Polymorphie* (dt. *Vielgestaltigkeit*). Für ein Beispiel in Java-Syntax siehe Abschnitt 5.7.4.

Mehrfachvererbung, d.h. die Möglichkeit Attribute, Konstruktoren oder Methoden mehrerer Oberklassen gleichzeitig zu erben, gibt es in Java (im Gegensatz zu C++) nicht. Es gibt aber in Java die Möglichkeit mittels sog. *Schnittstellen* (engl. *interfaces*) die Mehrfachvererbung eingeschränkt doch noch zu realisieren. (Mehr zu Schnittstellen s. Abschnitt 2.8.)

2.7 Abstrakte Klassen

Durch sogenannte *abstrakte Klassen* hat man in Java die Möglichkeit, Gemeinsamkeiten von Klassen in einer (abstrakten) Oberklasse zusammenzufassen. Da jedoch in den Klassen die Methoden womöglich unterschiedlich implementiert sind, müssen diese Methoden auch in der Oberklasse *abstrakt*, d.h. ohne konkrete Implementierung, definiert werden. Man spricht in diesem Zusammenhang vom *Bottom-Up-Entwurf* einer Klassenhierarchie, an deren Spitze die abstrakten Klassen stehen. Abstrakte Klassen können nicht instantiiert werden! Eine Unterklasse einer abstrakten Klasse kann wieder eine abstrakte Klasse sein.

Nicht alle Methoden einer abstrakten Klasse in Java müssen abstrakt sein, mindestens aber eine. Die abstrakten Methoden einer abstrakten Klasse sind erst in den Klassenerweiterungen durch Überschreiben mit konkretem Inhalt gefüllt.

Abstrakte Klassen sind von Schnittstellen (s. Abschnitt 2.8) zu unterscheiden, welche vor der Implementierung der Klassen spezifiziert werden sollten.

2.8 Schnittstellen

Schnittstellen (engl. *interfaces*) sind ein Konstrukt in Java, das aus modularen Programmiersprachen übernommen wurde.

Eine Schnittstelle ist ein Programmabschnitt, indem man Konstanten und Methoden deklarieren kann ohne die Methoden zu implementieren. Erst in einer davon abgeleiteten Klasse können die in einer Schnittstelle deklarierten

Methoden dann implementiert werden. Man hat so eine klare Trennung von Schnittstelle und Implementierung. Es empfiehlt sich daher, **vor** der Implementierung von Klassen deren Attribute und Methoden in Schnittstellen zu spezifizieren. Somit unterscheidet sich eine Schnittstelle von einer abstrakten Klasse (Abschnitt 2.7).

Eine Schnittstelle ist keine Klasse, und auch insofern von abstrakten Klassen zu unterscheiden. Im Gegensatz zu abstrakten Klassen können jedoch die Methoden einer Schnittstelle in jeder beliebigen Klasse einer beliebigen Klassenhierarchie verwendet und implementiert werden (insofern sichtbar).

Auch Schnittstellen können wie Klassen erweitert werden. Bei Schnittstellen ist jedoch weiterhin die Erweiterung mehrerer Oberschnittstellen in einer Unterschnittstelle erlaubt. Auf diese Art und Weise ist es in Java möglich, Klassen zu definieren, die Signaturen erben, welche in mehreren Oberklassen spezifiziert sind. Dadurch läßt sich eingeschränkt *Mehrfachverbung* in Java realisieren.

2.9 Freispeicherverwaltung

In Java muß sich der Programmierer nicht um das Aufräumen nicht mehr benötigter Objekte kümmern. Stattdessen gibt es eine *automatische Speicherbereinigung* (Freispeicherverwaltung, engl. *garbage collection*), d.h. Objekte werden automatisch aus dem Speicher entfernt, wenn keine Referenzen mehr auf sie bestehen. Dies vermeidet Fehler, die in anderen Programmiersprachen auftauchen können, wo durch konkretes Löschen der Speicherplatz eines Objektes freigegeben werden kann, obwohl noch Referenzen darauf bestehen.

3. Sonstige Sprachkonstrukte in Java

3.1 Pakete

Modularität ist keine Erfindung objektorientierter Sprachen, aber in diese (jedenfalls in Java) integriert. Dabei werden Klassen und Schnittstellen, die in einem gewissen Sinnzusammenhang stehen (diesen definiert der Softwareingenieur, oder auch nicht), zu einem sogenannten *Paket* (engl. *package*) - eben Modul - zusammengefaßt. In Java können Pakete hierarchisch aufgebaut sein.

Auf diese Pakete kann in Java auf zweierlei Art zugegriffen werden. Entweder wird das komplette (Teil-)Paket importiert, so daß deren Klassen und Schnittstellen direkt ansprechbar sind, oder einzelne Klassen bzw. Schnittstellen eines Pakets werden direkt über den Paketnamen als Zusatz angesprochen (Syntax s.u.). Hierbei ist darauf zu achten, daß bei der Benutzung Klassennamen und Paketnamen nicht verwechselt werden.

3.2 Überladen

In einer Klasse können in Java mehrere Methoden mit gleichem Namen deklariert sein. Diese müssen sich aber in ihrer Parameterliste unterscheiden. Es genügt, wenn andere Typen verwendet werden. Man nennt dieses *Überladen* (engl. *overloading*) von Methoden. Das Überladen von Operatoren ist in imperativen Programmiersprachen üblich: Man kann den Additions-Operator z.B. sowohl auf ganze als auch Gleitkomma-Zahlen anwenden. In Java können Operatoren aber durch den Benutzer nicht weiter überladen werden (wohl aber Methoden).

Im Unterschied zum Überschreiben von Methoden bedeutet *Überladen* von Methoden die Verwendung einer Methode mit unterschiedlichen Parametern. Diese können durchaus innerhalb einer Klasse definiert sein. *Überschreiben* von Methoden bedeutet dagegen die Reimplementierung einer Methode mit gleichen Parametern aus der Oberklasse.

3.3 Ausnahmebehandlung

Die *Ausnahmebehandlung* (engl. *exception handling*) ist ein Vorgehen, um das Auftreten "außergewöhnlicher" Situationen, eben Ausnahmen, die sonst gegebenenfalls einen Fehler hervorrufen würden, abzufangen.

Wird eine Ausnahme ausgelöst, so wird sie an das die Methode aufrufende Konstrukt weitergegeben. Dort wird die Ausnahme abgefangen und kann entsprechend behandelt werden. Außer vordefinierten Ausnahmen besteht die Möglichkeit, eigene Ausnahmen zu definieren. Ausnahmen sind in Java Objekte mit Attributen, Konstruktoren und Methoden.

3.4 Nebenläufigkeit

In Java besteht die Möglichkeit des *Multithreadings*. Ein Thread kann eine Aufgabe unabhängig von anderen Threads durchführen. Dabei können Threads auf gemeinsame Objekte zugreifen (engl. *shared access)*. Um dabei auftauchende

Fehlerquellen zu eliminieren, besteht in Java die Möglichkeit, Objekte oder Methoden für den gemeinsamen Zugriff zu sperren (*gegenseitiger Ausschluß*, engl. *mutual exclusion*). Durch diesen Mechanismus ist es weiterhin möglich, Threads zu *synchronisieren* und Nachrichten zwischen Threads auszutauschen (z.B. über gemeinsame Objekte oder deren Attribute). Die Probleme, die trotz oder aufgrund dieses Mechanismus auftreten können - wie z.B. Verklemmungen -, sollen hier nicht diskutiert werden. Es sei dazu auf die entsprechende Literatur bzw. Vorlesungen verwiesen.

3.5 Netzwerkfähigkeit

Java bietet ein umfangreiches Paket an, in dem Klassen zur Kommunikation zwischen Rechnern mit verschiedensten Protokollen wie HTTP oder TCP implementiert sind. Damit lassen sich ohne weiteres auch Client-Server-Programme verwirklichen.

4. Java-Entwicklung auf verschiedenen Plattformen

In diesem Abschnitt soll beschrieben werden, wie die zur Benutzung und Entwicklung von Java-Programmen auf den unterschiedlichen Plattformen des Fachbereichs vorhandenen Werkzeuge zu benutzen sind. Diese Dokumentation bezieht sich auf die Entwicklungsumgebung *Java Development Kit* (*JDK*) in der Version 1.0.2. Da im Fachbereich zur Zeit nur SUN-Workstations und PCs mit Windoofs 95/NT zur Verfügung stehen, beschränkt sich dieser Abschnitt auf diese beiden Plattformen. Wer auf dem Mac programmieren will, kann sich das Java Development Kit (JDK) 1.0.2 via WWW von Sun herunterladen unter der Adresse `http://www.javasoft.com/products/jdk/1.0.2/installation-macos-mac.html`. Die neueste Version JDK 1.1.1 steht bisher nur für Windoofs 95/NT und Solaris zur Verfügung.

Generell sollten Java-Quellprogramme die Endung "`java`" und ihre Übersetzung in Bytecode die Endung "`class`" besitzen.

4.1 SUN-Workstations

Das Java Development Kit 1.1 ist auf den SUNs verfügbar - die einzelnen Komponenten werden wie folgt aufgerufen:

4.1.1 Compiler

Der Java-Compiler `javac` übersetzt die Java-Quellprogramme in den plattformunabhängigen Bytecode.

Syntax:
```
javac [-g][-O][-debug][-depend][-nowarn][-verbose][-classpath path]
      [-nowrite][-deprecation][-d dir][-J <runtime flag>] <filename>.java...
```

Im Normalfall genügt der Aufruf von `javac <filename>.java`, also z.B. `javac test.java`. Es wird eine Datei mit der Endung `class` erzeugt, also im Beispiel `test.class`.

4.1.2 Laufzeitumgebung (engl. runtime system)

Zu Bytecode übersetzte Java-Programme können mit dem Java-Interpreter `java` zur Ausführung gebracht werden.

Syntax: `java [-options] <filename>.class`

Der Aufruf von `javac` ohne Parameter listet die möglichen Optionen und deren Bedeutung auf. Im Normalfall genügt der Aufruf von `javac <filename>.class`, also z.B. `java test.class`.

4.1.3 Debugger

Mit dem Java-Debugger `javadb` können Java-Programme debugt werden.

Syntax: `javadb`

Man gelangt nun in einen Interpreter-Modus ähnlich wie beim C-Debugger `dbx`. Der Modus wird verlassen mit dem Befehl `quit`.

4.1.4 Document generator

In Java besteht die Möglichkeit eine automatische Dokumentation mit Unterstützung entsprechend gekennzeichneter Kommentare (s. Abschnitt 5.4.1) im Quelltext zu erstellen. Dies geschieht durch Aufruf des document generators `javadoc`.

Syntax: `javadoc options* [class|package]`

Der Aufruf von `javadoc` ohne Parameter listet die möglichen Optionen und deren Bedeutung auf.

4.1.5 Applet viewer

Applets können außer im Java-fähigen WWW-Browser durch Aufruf des `appletviewer` gestartet werden.

Syntax: `appletviewer [-debug] [-J<javaflag>] [-encoding <char encoding type>] url|file ...`

Die sonstigen Tools sind unter `/app/java/bin` zu finden.

4.2 PCs mit Windows 95 / NT

Benutzt man das Java-Development Kit unter Windows95/ NT, so kann man innerhalb einer DOS-Shell die verschiedenen Tools wie Compiler und Laufzeitumgebung genauso aufrufen, wie unter Unix. Eventuell unterscheiden sich die Kommandozeilen-Parameter.

Im Programm-Manager von Windows befindet sich außerdem eine Programmgruppe namens Java. Öffnet man diese, so findet man dort Icons mit den Namen Symantec Cafe, Msdev, Jt, Guidgen und Ole2vw32. *Symantec Café 1.50* und *Msdev* (Microsoft Developer Studio for Visual J++) sind komfortable Entwicklungsumgebungen für Java.

Wir beschränken uns hier auf die Beschreibung von Symantec Café. Symantec C@fé 1.50 liegt das Java Development Kit 1.0.2 zugrunde..

Der Symantec Café zugehörige Pfad ist `v:\lang\javacafe\Bin\CAFE.EXE`

In Symantec Café gibt es verschiedene Ansichten (auch *Workspaces*) der aktuellen Programmentwicklung. Generell gelangt man zu den verschiedenen Ansichten durch Auswahl der gewünschten Ansicht bei Goto View im Menüpunkt Window. In den meisten Ansichten öffnet sich durch Betätigen der rechten Maustaste ein kleines ansichtsspezifisches Menü.

4.2.1 Projekte

Bei Symantec Café werden die zu einer Java-Anwendung gehörenden Dateien in sogenannten *Projekten* (engl. *projects*) zusammengefaßt. Die zugehörigen Dateinamen sollten die Endung "`prj`" besitzen.

Das Öffnen eines bestehenden Projektes erfolgt durch Aufruf Open unter dem Menü-Punkt Project. Anschließend gelangt man in die Projekt-Ansicht, bei der die zum Projekt gehörenden Dateien angezeigt werden. Es können nicht mehrere Projekte gleichzeitig geladen sein.

Ein neues Projekt wird durch Aufruf New unter dem Menü-Punkt Project erstellt. Hierauf öffnet sich ein Fenster, in dem u.a. der Name des neuen Projektes eingegeben wird (1. Name project), bereits bestehende Java-Quellcodes zum Projekt hinzugefügt werden können (3. Add files to project) und insbesondere festgelegt wird, ob es sich um ein Java-Applet (Target Type Applet) oder ein normales Java-Programm (Target Type Application) handeln soll (2. Set project type).

Weitere Quelldateien lassen sich durch Aufruf von Edit unter dem Menü-Punkt Project durch Auswahl der hinzuzufügenden Datei (`*.java`) dem Projekt hinzufügen.

4.2.2 Texteditor

Im *Texteditor* kann der Quellcode der Programme editiert werden. Bei einem bestehenden Projekt gelangt man in den Texteditor durch Doppelklick auf die zu editierende Datei der Projektansicht.

Eine neue Datei wird durch Aufruf New im Menü-Punkt File erstellt. Sie wird einem bestehenden Projekt hinzugefügt, indem sie im gleichen Verzeichnis abgespeichert wird wie die zugehörige `*.prj`-Datei.

Durch Klicken der rechten Maustaste im Editor öffnet sich ein Menü.

4.2.3 Hierarchieeditor

Zum *Hierarchieeditor* (*hierarchy editor*) gelangt man durch Auswahl von Goto View - Hierarchy Editor unter dem Menüpunkt Window. Es öffnet sich ein Fenster, in dem die aktuelle Klassenhierarchie in einem Graphen dargestellt wird. Bestehende Abhängigkeiten können verändert werden, indem man mit der linken Maustaste die Verbindungslinien markiert und verschiebt. Durch Klicken der rechten Maustaste öffnet sich ein kleines Menü.

4.2.4 Klasseneditor

Zum *Klasseneditor* (*class editor*) gelangt man durch Auswahl von Goto View - Class Editor unter dem Menüpunkt Window. Zum Klasseneditor gelangt man auch, indem man im Texteditor eine Methode markiert, die rechte Maustaste drückt und im sich daraufhin öffnenden Menü Go to - Definition auswählt.

Es öffnet sich ein drei-geteiltes Fenster. Im ersten Teil links oben erscheint eine Liste aller verwendeten Klassen (classes). Im zweiten Teil rechts oben erscheint eine Liste aller Methoden (methods) und Attribute (Data) der ausgewählten Klasse (Members of ...). Im dritten unteren Teil wird die Implementation der - durch Doppelklick - ausgewählten Methode bzw. die Definition des ausgewählten Attributes angezeigt.

Die Implementation im dritten Fenster kann ganz normal editiert werden. Wie im Texteditor öffnet sich nach dem Drücken der rechten Maustaste ein Menü.

In den anderen beiden Fenstern können nach Drücken der rechten Maustaste Klassen bzw. Methoden und Attribute zu den bestehenden Listen hinzugefügt oder aus diesen entfernt werden. Z.B. wird zu den Klassen eine erweiterte Klasse hinzugefügt, indem man im Teil Classes die rechte Maustaste drückt und im hierauf erscheinenden Menü Add Derived auswählt.

4.2.5 Compilieren

Ein Programmteil bzw. ein komplettes Java-Programm wird durch Aufruf von Build bzw. Rebuild All unter dem Menü-Punkt Project übersetzt. Es wird eine Datei erzeugt mit dem gleichen Stammnamen wie das Projekt und der Endung `class`. In der output-Ansicht werden entweder die erfolgreich ausgeführte Übersetzung oder eventuell aufgetretende Fehler angezeigt.

4.2.6 Programmausführung

Ein Java-Projekt wird in Symantec Café durch Aufruf von Execute Programm unter dem Menü-Punkt Project zur Ausführung gebracht. Existiert noch keine aktuelle Bytecode-Datei (Endung `class`), so wird zuvor das Projekt (neu) übersetzt.

5. Programmieren in Java

Die Syntax der Programmiersprache Java ist in vielen Teilen der Programmiersprache C entnommen. Viele Unartigkeiten, die in C möglich sind, lassen sich allerdings in Java nicht machen. Zum Vergleich von Java und C++ siehe Abschnitt 6.

5.1 Aufbau eines Programms in Java

Java-Programme bestehen aus Klassen und Schnittstellen. Diese können in Paketen zusammengefaßt sein. Ein Java-Programm besteht mindestens aus einer Klasse. Diese muß die Methode `main` (s.u.) besitzen. Will man ein Programm laufen lassen, so spezifiziert man genau diese Klasse, und der Java-Interpreter startet die zu dieser Klasse gehörende Methode `main`.

Bei der Methode `main` handelt es sich um eine Klassenmethode (also `static`), sie ist keine Funktion (sondern `void`) und öffentlich (`public`). Die Methode `main` besitzt genau einen Parameter vom Typ `String[]`:

```
public static void main (String[] args) { /* ... */ }
```

Das Array-Objekt `args` übergibt die Kommandozeilen-Parameter an das Programm.

5.2 Typen, Variablen und Konstanten

Java-Quellprogramme werden in sogenannten Bytecode übersetzt. Dies ist eine Maschinensprache eines virtuellen Prozessors. Ein Java-Interpreter übersetzt dann den Bytecode in die Maschinensprache der jeweiligen Plattform. Aus diesem Grunde hängt die Größe von Datentypen nicht von der aktuellen Plattform ab, sondern ist immer gleich groß. So sind z.B. ganze Zahlen vom Typ `int` in Java immer 32 bit lang.

5.2.1 Einfache Typen

`boolean`	entweder `true` oder `false`
`char`	16-bit-Unicode Zeichen

`byte`	8-bit-Integer
`short`	16-bit-Integer
`int`	32-bit-Integer
`long`	64-bit-Integer
`float`	32-bit-Gleitkommazahl
`double`	64-bit-Gleitkommazahl

Der Typ boolean ist kein Zahl-Typ und wird auch nicht implizit konvertiert. Insbesondere ist ein Wahrheitsausdruck (boolscher Ausdruck) vom Typ boolean und kein Zahl-Typ wie in C (dort ist x falsch, wenn x=0 und sonst wahr).

Die Werte der Typen `short` und `byte` werden vor ihrer Verwendung nach `int` umgewandelt.

Der Typ `char` enthält 2^{16} Zeichen. Die ersten 128 davon entsprechen dem ASCII-Zeichensatz, die ersten 256 dem ISO-Latin Zeichensatz. Der Rest ist i.a. unwichtig. Eine Reihe von Zeichen besitzt eine spezielle Notation:

`\n`	Zeilenvorschub
`\t`	Tabulator
`\b`	Backspace
`\r`	Wagenrücklauf
`\f`	Seitenvorschub
`\\`	Backslash
`\'`	Apostroph
`\"`	Anführungsstriche

Für Zeichenketten gibt es keinen eigenen Typ. Stattdessen sind Zeichenketten als `Objekte` der Klasse `java.lang.String` definiert und werden zwischen zwei Anführungszeichen geschrieben: `"mehrere Zeichen"`.

5.2.2 Variablendeklaration

Variable werden in Java durch Angabe des Typs deklariert:

```
<Typ> <Variablenname>;
```

Der Variablen kann bereits bei der Deklaration ein Anfangswert zugewiesen werden:

```
<Typ> <Variablenname> = <Wert>;
```

Dies ist zur Vermeidung von Fehlern unbedingt zu empfehlen. Z.B.

```
int i = 0;
float radius = 1.0f;
```

5.2.3 Zusammengesetzte Typen

Felder (*Arrays*) werden in Java wie folgt definiert:

```
<Typ> [] <Feldvar.name>;
```

Arrays werden in Java wie Objekte behandelt, insbesondere sind Felder Zeiger. Sie werden entweder durch Initialisierung bei der Deklaration oder durch Zuweisung eines anderen Arrays oder mit new instantiiert.

```
int [] Feld1 = {2,4,6,8,10};  // Klammerausdruck nur bei Initialisierung verwendbar
int [] Feld2;
Feld2 = Feld1;
int [] Feld3 = new int[5];
```

Felder können über ihre Indizes angesprochen werden, dabei ist der erste Index die 0:

```
// Fortsetzung
Feld2[3] = 5;  // Was ist jetzt der Wert an der Stelle Feld1[3]?
```

Es können auch *mehrdimensionale Felder* deklariert werden. Dabei müssen die Felder der zweiten Dimension nicht überall gleiche Länge besitzen. Im folgenden Beispiel ist ein Pascalsches Dreieck vom Level 5 definiert:

```
int [][] Feld2D = new int[5][];
Feld2D[0] = new int[1];
/* ... */
```

```
Feld2D[4] = new int[5];
```

In Java gibt es keinen Typ `Record` wie in Pascal (bzw. `struct` wie in C). Ebensowenig existieren in Java Aufzählungen.

5.2.4 Zeiger, Referenzen

In Java ist der Wert einer Objektvariable ein *Zeiger* (*Referenz*, engl. *pointer, reference*) auf den Speicherbereich, in dem das Objekte abgelegt ist. Ansonsten gibt es in Java keine Zeiger. Insbesondere gibt es keine Zeigerarithmetik wie in C.

5.2.5 Konstanten

Variablen werden zu Konstanten deklariert, indem vor der Variablendeklaration der Modifier `final` geschrieben wird. Konstanten müssen bei der Deklaration initialisiert werden.

```
final <Typ> <Konstantenname> = <Wert>;
```

5.3 Zuweisung und Operatoren

5.3.1 Zuweisung

Der Zuweisungsoperator ist in Java wie in C **ein** Gleichheitszeichen =, z.B.

```
i = 10;
```

Dies entspricht in Pascal der Zuweisung `i := 10;` und ist nicht zu verwechseln mit dem Vergleich, der aus **zwei** Gleichheitszeichen besteht: ==. Die Zuweisung ist in Java rechts-assoziativ.

In Java besitzt ein Ausdruck wie `i = 10` einen Wert, der dem Zuweisungswert (in diesem Falle `10`) entspricht. Es ist daher folgende Anweisung möglich:

```
j = i = 10;
```

Solche Anweisungen sind allerdings mit Vorsicht zu genießen!

5.3.2 Typkompabilität

Java hat eine strenge Typbindung. In einigen Fällen ist es trotzdem möglich, unterschiedliche Typen einander zuzuweisen. Dabei findet in folgenden Fällen eine *implizite* (automatische) Typumwandlung statt:

von	nach
`float`	`double`
ganze Zahl	ganze Zahl mit größerer Genauigkeit
ganze Zahl	Gleitkommazahl
`char`	`int`, `long`

Daneben sind in einigen Fällen auch *explizite* Typumwandlungen (engl. *cast*) möglich. Dabei wird der neue Typ der Variablen in Klammern vorweggestellt:

```
(<neuer Typ>) Variable
```

Folgende explizite Typumwandlungen sind erlaubt:

von	nach
`double`	`float`
ganze Zahl	ganze Zahl mit größerer Genauigkeit
Gleitkommazahl	ganze Zahl
`char`	`short`, `byte`

Dabei ist im allgemeinen mit Genauigkeitsverlust zu rechnen, eventuell kann sich sogar das Vorzeichen ändern! Zuweisungen mit expliziter Typumwandlung sollten also vermieden werden.

Ein `boolean` kann nicht in ein `int` umgewandelt werden!

5.3.3 Arithmetische Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo (Rest)

Wie in C sind auch Kombinationen von Zuweisungsoperator und arithmetischem Operator erlaubt. So bedeutet z.B. die Anweisung `i += 10;` das gleiche wie die Anweisung `i = i + 10;`, erfordert aber nur einen einmaligen Zugriff auf die Variable `i`. Bei Ausdrücken ist allerdings Vorsicht geboten!

5.3.4 Inkrement- und Dekrementoperatoren

Die Operatoren `++` und `--` dienen zum Inkrementieren bzw. Dekrementieren ihrer Operanden. Die Anweisung `++i;` ist z.B. gleichbedeutend mit `i += 1;` bzw. `i = i+1;`. Der Ausdruck `++i` ist hierbei vom Ausdruck `i++` zu unterscheiden. Im ersten Fall wird der Wert von `i` zuerst erhöht und dann der Ausdruck ausgewertet, im zweiten Fall umgekehrt. Bei der Anweisung `j = ++i;` erhält `j` den Wert von `i+1`, bei der Anweisung `j = i++;` den Wert von `i`. Auch solche Anweisungen sind daher sehr mit Vorsicht zu genießen!

5.3.5 Zeichenkettenoperatoren

Zeichenketten können mit + miteinander verkettet werden. Z.B.

```
String hallo = "Hello";
String welt = "world";
String gruss = hallo + " " + welt + "!";
```

Nützlich sind weiterhin die Methoden `substring` und `charAt`. Die Funktion `substring (int beginIndex)` z.B. liefert eine neue Zeichenkette (`String`) zurück, die eine Teilzeichenkette der ursprünglichen Zeichenkette ist, beginnend an der Position `beginIndex` bis zum Ende der ursprünglichen Zeichkette. Die Funktion `charAt (int index)` liefert den Buchstaben (`char`) an der Position `index` in der Zeichenkette. Der erste Buchstabe hat die Position 0. Weitere Methoden finden sich in der Klasse `String` im Paket `java.lang`.

5.3.6 Vergleichsoperatoren

>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	gleich
!=	ungleich

Man beachte, daß der Gleichheitsoperator hier im Unterschied zu Pascal aus **zwei** Gleichheitszeichen besteht! Dies kann leicht zu Fehlern führen, da **ein** Gleichheitszeichen die Zuweisung bedeutet!

5.3.7 Logische Operatoren

&&	logisches Und
||	logisches Oder
!	logische Verneinung (Negation)

5.3.8 Bitoperatoren

&	bitweises Und
|	bitweises Oder
^	bitweises Entweder-Oder (XOR)
~	bitweises Komplement
<<	Shift links (Auffüllen mit 0)
>>	Shift rechts (Auffüllen mit höchstem Bit, vorzeichentreu)

>>>	Shift rechts (Auffüllen mit 0)

5.3.9 Priorität der Operatoren

Für Operatoren gilt folgende Prioritäten-Reihenfolge bei der Auswertung:

Postfix-Operatoren	[] . (<parameter>) <ausdruck>++ <ausdruck>--
unäre Operatoren	++<ausdruck> --<ausdruck> +<ausdruck> -<ausdruck> ~ !
Erzeugung oder Typumwandlung	new (<typ>)<ausdruck>
Multiplikationsoperatoren	* / %
Additionsoperatoren	+ -
Verschiebeoperatoren	<< >> >>>
Vergleichsoperatoren	< > <= >= instanceof
Gleichheitsoperatoren	== !=
Bit-Und-Operator	&
Bit-Entweder-Oder-Operator	^
Bit-Oder-Operator	|
logischer Und-Operator	&&
logischer Oder-Operator	||
Bedingungsoperator	?:
Zuweisungsoperatoren	= += -= *= /= %= >>= <<= >>>= &= ^= |=

5.4 Kontrollstrukturen

5.4.1 Kommentar

Es gibt in Java drei Arten von Kommentaren

```
... // Kurzkommentar bis Ende der Zeile
... /* Kommentar auch ueber mehrere Zeilen hinweg */
... /** Kommentar zur Programmdokumentation */
```

Java-Kommentare können nicht verschachtelt werden.

5.4.2 Block

Ein Block beginnt wie in C mit dem Zeichen `{` und endet mit dem Zeichen `}`. Ein Block kann vorzeitig durch die Anweisung

```
break;
```

verlassen werden. Dies ist aber ein schlechter Programmierstil und sollte, wenn es geht, vermieden werden. (Leider geht es nicht immer - siehe `switch`-Anweisung.)

5.4.3 Bedingte Anweisung if-else

Die bedingte Anweisung hat in Java die Form

```
if (<Bedingung>) <Dann-Anweisung(sblock)>
else <Sonst-Anweisung(sblock)>
```

Der `else`-Teil kann auch entfallen. Bedingte Anweisungen können verschachtelt werden. Dabei bezieht sich der `else`-Teil immer auf den letzten vorhergehenden `if`-Teil. Um Verwirrungen zu vermeiden empfiehlt sich die Verwendung von Einklammerungen zu Blöcken.

5.4.4 Switch-Anweisung

Die `switch`-Anweisung ist ähnlich wie in C definiert und entspricht (fast!) der `case`-Anweisung in Pascal. Hier wird der `case`-Ausdruck dazu verwendet, um die Integergleichheit zu überprüfen. Bei Gleichheit wird der anschließende Anweisungsteil ausgeführt. Danach wird (im Unterschied zu Pascal!) der nächste `case`-Ausdruck ausgewertet und verglichen, es sei denn, der `switch`-Ausdruck wurde vorher explizit durch eine `break`-Anweisung abgebrochen. Ist

keine der Bedingungen erfüllt, so wird - falls vorhanden - der Anweisungsteil hinter dem `default`-Ausdruck ausgeführt.

```
switch (<int-Var>)
{  // BEGIN switch
   case <int-Konst1>:
      <Anweisung(sblock)1>    // wird ausgeführt, falls <int-Var>==<int-Konst1>
      // zum nächsten case-Ausdruck
   case <int-Konst2>:
      <Anweisung(sblock)1>
      break;
      /* hier wuerde abgebrochen werden, falls <int-Var>==<int-Konst1>
         oder <int-Var>==<int-Konst2> war */
   case <usw.>
   default:
      <default-Anweisung(sblock)>   // wird immer ausgeführt,
                                    // wenn vorher kein break !!
}  // END switch
```

5.4.5 For-Schleife

Die for-Schleife sieht analog zu C wie folgt aus:

```
for (<Initialisierung>; <Bedingung>; <In-/Dekrement>)
   <Anweisung(sblock)>
```

Man beachte dabei, daß die In- bzw. Dekrementierung nach dem Abarbeiten des Anweisungsblocks erfolgt.

Z.B. zur Berechnung der Fakultät von n

```
fak = 1;
for (i=1; i<=n; i++) fak *= i;   // n>=0 vorausgesetzt
```

Dies entspricht in Pascal der `for`-Anweisung

```
fak := 1;
for i:=1 to n do fak := fak * i;  { n>=0 vorausgesetzt }
```

In Java sind jedoch auch komplexere Ausdrücke für Initialisierung, Bedingung oder In-/Dekrement erlaubt. So ist z.B. auch die folgende for-Schleife möglich:

```
for (i=0, j=n-1; j>=0; i++, j--) <Anweisung(sblock)>
```

Wird die Bedingung leer gelassen, so wird `true` angenommen.

Es ist sogar möglich im Initialisierungsteil lokale Variablen zu deklarieren:

```
for (int i=0; i<k; i++) <Anweisung(sblock)>
```

Dies macht Sinn, da die Variable (`i` im Beispiel) nur lokal ist und somit nicht der Wert einer anderen gleichnamigen Variablen (außerhalb der Schleife deklariert) überschrieben werden kann.

Es ist jedoch auf Lesbarkeit des for-Konstruktes zu achten.

5.4.6 While-Schleife

Die While-Schleife gibt es in zwei Ausprägungen:

```
while (<Bedingung>) <Anweisung(sblock)>
```

Dies entspricht der `while`-Schleife in Pascal. Und

```
do <Anweisung(sblock)> while (<Bedingung>);
```

Dies entspricht vom Auswertungszeitpunkt her der `repeat`-Schleife in Pascal, allerdings wird in Pascal abgebrochen, falls die Bedingung erfüllt ist, während die Schleife in Java abgebrochen wird, falls die Bedingung **nicht** erfüllt ist.

5.4.7 goto-, break-, continue- und return-Anweisungen

Es gibt in Java glücklicherweise **keine** `goto`-Anweisung! Aber es gibt einige andere unschöne Anweisungen, mit denen sich ähnliche Effekte erreichen lassen (pfui!). Diese Unarten hat man z.T. leider aus der Sprache C

übernommen.

Die `break`-Anweisung haben wir ja bereits im Abschnitt über Blöcke (5.4.2) kennengelernt.

Die `continue`-Anweisung kann (sollte aber nicht) in Schleifen genutzt werden. Trifft das Programm auf die `continue`-Anweisung, so springt es ans Ende der Schleife und wertet die Schleifenbedingung erneut aus.

Die `return`-Anweisung beendet eine Funktion und kehrt an den Aufrufer zurück. Handelt es sich um eine echte Funktion mit Rückgabe, so steht hinter dem `return` der Rückgabewert. Die `return`-Anweisung ist in Java für Funktionen mit Rückgabe daher notwendig. Ein guter Programmierer implementiert die Funktion jedoch so, daß die `return`-Anweisung immer nur am Ende der Funktion steht.

```
int Betrag (int x)   // Funktion int -> int
{  int b;

   if (x>=0) b=x;
   else b=-x;

   return b;
}  // END Betrag
```

5.5 Klassen, Methoden und Objekte

5.5.1 Klassen

Klassen können in Java wie folgt deklariert werden:

```
class <klassenname>
{  <Deklaration der Attribute>

   <Deklaration der Konstruktoren>
   <Deklaration der Methoden>
}  // END class <klassenname>
```

5.5.2 Attribute

Ein Attribut wird ähnlich wie eine Variable durch Angabe von Typ und Name und gegebenenfalls einer Initialisierung deklariert. Wird ein Attribut nicht explizit initialisiert, so wird es - im Unterschied zu Variablen einer Methode - defaultmäßig initialisiert.

Es können außerdem noch sogenannte Modifikatoren davor geschrieben werden (s. Abschnitt 5.8). Klassenattribute werden durch den Modifikator `static` deklariert.

5.5.3 Konstruktoren

Konstruktoren werden in Java wie folgt deklariert:

```
<Konstruktorname> ( <Parameterliste> ) <Anweisungsblock>
```

Der Konstruktorname ist gleich dem Klassennamen!

Falls kein Konstruktor angegeben ist, wird automatisch ein Default-Konstruktor mit leerer Parameterliste und leerem Anweisungsblock definiert.

Konstruktoren kann ein Modifikator vorweggestellt werden.

5.5.4 Methoden

Eine Methode ist in Java wie folgt deklariert:

```
<Typ> <Methodenname> ( <Parameterliste> ) <Anweisungsblock>;
```

Ist der Typ explizit (z.B. als `int`) angegeben, so handelt es sich um eine Funktion. Für Prozeduren wird der "leere Typ" `void` verwendet. Für Funktionen erfolgt die Rückgabe durch die return-Anweisung (s. Abschnitt 5.4.7).

Zusätzlich können den Methoden noch Modifikatoren vorweggestellt werden (s. Abschnitt 5.8).

Das Überladen von Methoden ist in Java i.a. erlaubt (s. Abschnitt 3.2). Klassenmethoden werden durch den Modifikator `static` deklariert. Klassenmethoden sollten sich ausschließlich mit klassenbezogenen Operationen beschäftigen.

5.5.5 Objekte

Objekte können in Java ausschließlich durch Aufruf eines *Konstruktors* instanziert werden. Vor dem Konstruktor steht das Schlüsselwort `new`:

```
<klassenname> <objektname> = new <konstruktorname> (<parameterliste>);
```

Taucht ein Objektname in der Parameterliste einer Methode auf, so wird dabei eine Objekt**referenz** als **Wert**parameter an die Methode übergeben.

5.5.6 Zugriff auf Attribute und Methoden von Objekten bzw. Klassen

Auf Attribute und Methoden von Objekten kann innerhalb von Objekten und deren Methoden direkt durch Angabe des Attributnamens zugegriffen werden. In Konfliktfällen kann das Attribut des eigenen Objekts eindeutig identifiziert werden durch das `this`-Konstrukt:

```
class <klassenname>
{  /* Attribute */
   int x=0;    // initialisiert mit 0

   /* Methoden */
   boolean gleich_x (int x)
   {  boolean f;

      if (x == this.x) f = true;
      else f = false;

      return f;
   }  // END Funktion gleich_x

   void setze_x (int x)
   {  this.x = x;
   }  // END Prozedur setze_x

}  // END class <klassenname>
```

Auf Attribute eines Objektes bzw. einer Klasse von außerhalb kann wie folgt zugegriffen werden (falls sichtbar):

```
<Objekt- bzw. Klassenname>.<Attributname>
```

Dies sollte jedoch vermieden werden. Besser ist es, durch entsprechende Methoden auf die Attribute zuzugreifen (siehe obiges Beispiel).

Auf Methoden eines Objektes bzw. einer Klasse von außerhalb kann wie folgt zugegriffen werden (falls sichtbar):

```
<Objekt- bzw. Klassenname>.<Methodenname>
```

5.5.7 Initialisierung einer Klasse

Der Initialisierungsteil einer Klasse steht i.a. zwischen den Attributdeklarationen und Methodendeklarationen der Klasse und wird wie folgt definiert:

```
static <Anweisungsblock>
```

Die Initialisierung der Klasse erfolgt beim ersten Zugriff auf diese Klasse.

5.6 Schnittstellen

Eine Schnittstelle (s. Abschnitt 2.8) in Java ist wie folgt definiert:

```
interface <schnittstelle>
{  /* Deklaration von Konstanten */
   // ...
```

```
   /* Deklaration von Methoden-Koepfen */
   <typ> <methode_1> (<parmeterliste_1>);
   <typ> <methode_2> (<parmeterliste_2>);
   /* usw. */
}  // END interface
```

Zur Verwendung von Schnittstellen wird das `implements`-Konstrukt benutzt:

```
class <klasse> implements <schnittstelle>
{  /* eigene Attribute */
   /* Implementierung der Methoden der Schnittstelle */
   /* eigene Methoden */
}
```

Die Erweiterung von Schnittstellen erfolgt nach folgendem Schema:

```
interface <unterschnittst.> extends <oberschnittst_1>, <oberschnittst_2> /* usw. */
{  /* evtl. neue Methodendeklarationen */  }
```

Beispiel:

```
interface I { /* ... */ }
interface I1 extends I { /* ... */ }
interface I2 extends I { /* ... */ }
class C implements I1, I2 { /* ... */ }
```

5.7 Klassenerweiterungen

5.7.1 Ober- und Unterklassen

Eine Klasse wird in Java durch das `extends`-Konstrukt erweitert:

```
[<Modifier>] <Unterklasse> extends <Oberklasse>
{  // ...
}
```

Die Unterklasse erbt alle (nicht privaten) Attribute, Konstruktoren und Methoden der Oberklasse(n) und kann diese erweitern oder überschreiben (falls nicht in der Oberklasse als `final` deklariert).

Auf die Methoden und Attribute der direkten Oberklasse kann mit dem Konstrukt `super.<methode_bzw._attr>` zugegriffen werden. Der Aufruf `super.super.<methode_bzw._attr>` ist **nicht** erlaubt.

5.7.2 Die Klasse `Object`

Die Oberklasse für alle Klassen in Java ist die Klasse `Object` aus dem Paket `java.lang`. (Der Name `Class` wäre eigentlich passender, denn es handelt sich ja um eine Klasse und nicht um ein Objekt.) Die Deklaration einer neuen Klasse mit

```
class <klassenname> { /* ... */ }
```

wie in Abschnitt 5.5.1 beschrieben ist eigentlich eine abkürzende Schreibweise für

```
class <klassenname>  extends Object { /* ... */ }
```

5.7.3 Abstrakte Klassen

Eine abstrakte Methode wird durch den Modifikator `abstract` definiert und ist eine reine Deklaration ohne Implementation:

```
abstract <typ> <methodename> (<parmeterliste>);
```

Eine Klasse, die eine abstrakte Methode enthält, ist eine abstrakte Klasse und muß ebenfalls mit dem Modifikator `abstract` versehen werden. Sie kann nicht instanziert werden (s. Abschnitt 2.7).

5.7.4 Polymorphie

Im folgenden sei ein Beispiel für Polymorphie (s. Abschnitt 2.6) gegeben:

- Definition der Klasse `Kantenzug` mit der Methode `intersection`:

```
public class Kantenzug
{  public double winkel;

   public Point[] intersection() { /* ... */ }
   /* ... */

}  // END Kantenzug
```

- Erweiterung der Klasse `Kantenzug` zur Klasse `Polygon` und Verfeinerung der Methode `intersection`:

```
public class Polygon extends Kantenzug

{  public int winkel;

   public Point[] intersection() { /* ... */ }
   /* ... */

}  // END Polygon
```

- Definition von Objektvariablen:

```
Polygon P = new Polygon();
Kantenzug L = P;    // Ein Polygon ist auch ein Kantenzug
Object O = L;       // Ein Kantenzug ist auch ein Object
Kantenzug [] zweiLZ = {new Polygon(), new Kantenzug};
```

- Durch die beiden folgenden Aufrufe werden zwei verschiedene Implementierungen der Funktion `intersection` angesprochen:

```
zweiLZ[0].intersection();
zweiLZ[1].intersection();
```

- Im folgenden wird jedoch jedesmal das Attribut `winkel` (vom Typ `double`) aus der Klasse `Kantenzug` angesprochen:

```
zweiLZ[0].winkel

zweiLZ[1].winkel
```

5.8 Modifikatoren

Modifikatoren können auf Attribute, Methoden oder sogar ganze Klassen angewendet werden.

Der Modifikator `static` deklariert ein Attribut bzw. eine Methode in ein Klassenattribut bzw. eine Klassenmethode um (s. Abschnitt 2.4).

Der Modifikator `final` deklariert ein Attribut in eine Konstante um. Diese **muß** initialisiert werden.

Die Modifikatoren `static` und `final` können für Attribute auch kombiniert werden (*Klassenkonstante*).

Der Modifikator `public` deklariert ein Attribut bzw. eine Methode als öffentlich, der Modifikator `protected` als geschützt und der Modifikator `private` als privat (s. hierzu Abschnitt 2.3). Diese drei Modifikatoren sind exklusiv zu verwenden. Zu beachten ist der Einfluß der Modifikatoren bei Klassenerweiterungen.

	Attribute	**Methoden**	**Klassen**
`final`	Konstante	nicht überschreibbar	nicht erweiterbar
`static`	Klassenattribut	Klassenmethode	-/-
`public`	öffentlich	öffentlich	öffentlich
`protected`	geschützt	geschützt	-/-
`private`	privat, nicht vererbbar	privat, nicht vererbbar	-/-
`abstract`	-/-	abstrakte Methode	abstrakte Klasse

5.9 Pakete

Pakete (Module, s. Abschnitt 3.1) werden in Java wie folgt deklariert:

```
package <Paketnamenfolge>;
```

Eine Paketnamenfolge kann aus mehreren Paketnamen bestehen, getrennt durch einen Punkt ".". Jede Quelldatei, deren Klassen und Schnittstellen in ein bestimmtes Paket gehören, muß obige Deklaration enthalten. In einer Quelldatei kann nur ein Paket deklariert werden.

In einem Paket können mehrere Klassen bzw. Schnittstellen definiert sein. Jede Klasse (bzw. Schnittstelle) gehört zu genau einem Paket und kann (z.B. beim Zugriff) über die Paketnamenfolge eindeutig identifiziert werden:

```
<Paketnamenfolge>.<Klassenname>
```

Hier ist auf die Unterscheidung von Paketnamen, Klassen und Methoden zu achten. Z.B. bedeutet

```
java.awt.image.ImageFilter.setPixels(...)
```

die Methode `setPixels(...)` der Klasse `ImageFilter` im Paket `java.awt.image`.

Damit auf Klassen bzw. Schnittstellen eines Pakets von außerhalb eines Pakets zugegriffen werden kann, müssen diese als öffentlich (`public`) deklariert sein. Pakete, Pakethierarchien oder auch nur einzelne (öffentliche) Klassen bzw. Schnittstellen aus Paketen können in ein Programm importiert werden:

```
import <eine_Paketnamenfolge>;   // importiert ein ganzes Paket
import <andere_Paketnamenfolge>.*;   // importiert Pakethierarchie ab Knoten
import <noch_andere_Paketnamenfolge>.<Klassenname>  // importiert Klasse
```

Die Klasse (bzw. Schnittstelle) kann dann ohne Angabe der Paketnamenfolge verwendet werden. Unter Umständen ist auf doppelte Namensgebung zu achten.

5.10 Ausnahmebehandlung

5.10.1 Definition von Ausnahmen

Ausnahmen sind in Java Objekte. Sie gehören alle zur Standard-Klasse `Throwable` oder einer ihrer Unterklassen. Selbstdefinierte Ausnahmeklassen sind Erweiterungen der Unterklasse `Exception`.

Zur Klasse `Exception` gehören standardmäßig zwei Konstruktoren, einer ohne Parameter, der andere mit einem Parameter vom Typ `String`. Außerdem gehören zu ihr die folgenden beiden Methoden:

```
public String getMessage();  // Liefert den beim Konstruktor übergebenen String.
public void printStackTrace();   // Gibt aus, wo die Ausnahme ausgelöst wurde.
```

Eine neue Ausnahme wird wie folgt definiert:

```
public class <NeueAusnahme> extends Exception
{  /* Attributdeklaration */

   <NeueAusnahme> (<Parameterliste>)  //  Konstruktor
       <Anweisungsblock>
}
```

5.10.2 Auslösen von Ausnahmen

Ausnahmen werden in Java durch die `throw`-Anweisung ausgelöst:

```
<Typ> <Methodenname> throws <Ausnahme>
{  // ...

   throw <AusnahmeKonstruktor>;
   // ...
}
```

5.10.3 Behandlung von Ausnahmen

Ausnahmen werden in Java durch die `try-catch-finally`-Anweisung wie folgt behandelt:

```
try <Anweisungsblock>
   catch (<Ausnahme1>) <Anweisungsblock>
   catch (<Ausnahme2>) <Anweisungsblock>
   // usw.
   finally <Anweisungsblock>
```

Der Rumpf der `try`-Anweisung wird entweder bis zum Ende oder bis zum Auslösen einer Ausnahme ausgeführt. Beim Auslösen einer Ausnahme werden die `catch`-Konstrukte nacheinander geprüft, ob sie die Ausnahme behandeln können. Wird ein passendes `catch`-Konstrukt gefunden, so wird die Ausführung des Programms im zugehörigen Anweisungsblock fortgesetzt. Wird kein passendes `catch`-Konstrukt gefunden, so wird die Ausnahme an die aufrufende Methode weitergereicht. Der Anweisungsblock im `finally`-Teil wird entweder nach einem vollständigen Durchlauf des `try`-Teils (kein passendes `catch`-Konstrukt gefunden) oder nach Durchlauf eines `catch`-Teils ausgeführt. Der `finally`-Teil kann auch weggelassen werden.

5.11 Nebenläufigkeit

5.11.1 Threads

Ein Thread ist in Java als Klasse `Thread` definiert. Neue `Thread`-Objekte werden daher wie folgt instanziert:

```
Thread <Thread-Name> = new Thread();
```

Durch Aufruf der Methode `start` wird ein neuer Thread gestartet. Die Methode `start` ruft automatisch die Methode `run` des neuen Threads auf. Die Methode `stop` bricht den Thread ab, die Methode `suspend` unterbricht den Thread.

Um einen sinnvollen Thread zu implementieren, kann die Klasse `Thread` erweitert und deren Methode `run` überschrieben werden.

```
class <my thread> extends Thread
{  /* ... */
   public void run()
   {  /* was der Thread tun soll */  }
   /* ... */
}
```

Alternativ dazu kann man aber auch die Schnittstelle `Runnable` benutzt werden, um die Methode `run()` zu implementieren.

5.11.2 Sperren von gemeinsamen Objekten bzw. Methoden

Um Fehler beim gemeinsamen Zugriff auf Objekte zu vermeiden, ist es möglich Objekte direkt oder durch Aufruf von bestimmten Methoden in Java zu sperren.

Um Objekte über Methoden sperren zu können, müssen diese vorher als `synchronized` deklariert worden sein. Ruft nun ein Thread eine als `synchronized` deklarierte Methode auf, so ist dessen Objekt für andere Threads gesperrt. Ein anderer Thread der nun ebenfalls auf das Objekt zugreifen will, wird solange in den Wartezustand versetzt.

Objekte können aber auch direkt gesperrt werden. Dies geschieht durch folgendes `synchronized`-Konstrukt:

```
synchronized (<objekt>) <Anweisung(sblock)>
```

Sobald eine Methode auf das `synchronized`-Konstrukt trifft, wird das Objekt gesperrt bis die Anweisung bzw. der Block von Anweisungen abgearbeitet ist.

Durch das Sperren von Objekten ist es bereits möglich Threads zu synchronisieren. Ein Thread, das auf ein gesperrtes Objekt zugreifen will, wird solange in den Wartezustand versetzt, bis das Objekt oder die Methode wieder freigegeben ist.

5.11.3 Kommunikation zwischen Threads

Zur Kommunikation von Threads dienen die Methoden `wait()` und `notify()`, die in der Klasse Object definiert sind, in Verbindung mit einer `synchronized`-Methode. Das Schema im empfangenden Prozeß könnte wie folgt aussehen:

```
synchronized <empfaenger-methode>
{  while (!<bedingung>) wait();
   /* was bei erfüllter Bedingung zu tun ist */
}
```

Durch Aufruf der Methode `wait()` wird der Thread in einen Wartezustand versetzt. Das Schema im sendenen Prozeß könnte wie folgt aussehen:

```
synchronized <sender-methode>
```

```
{  /* Änderung von Werten, die in obiger Bedingung auftreten */
   notify();
}
```

Durch Aufruf der Methode `notify()` wird der am längsten wartende Thread aufgeweckt (aktiviert). Auf die verschiedenen Form der beiden Methoden wird hier nicht weiter eingegangen.

Für weitere Informationen über Threads und Beispiele in Java sei auf die Literatur verwiesen.

5.12 Netzwerkfähigkeit

Im Java-Paket `java.net` sind die Schnittstellen, Klassen und Ausnahmen zur Unterstützung von netzwerkfähigen Anwendungen enthalten. Unterstützt werden u.a. URLs, TCP sockets, UDP sockets und IP-Adressen.

5.13 Standardpakete

Zunächst eine Übersicht über einige Java-Standardpakete:

`java.applet`	Implementierung von Applets
`java.awt`	Grafik, Fenster und GUIs
`java.awt.image`	Bildverarbeitung
`java.awt.peer`	GUI Toolkit für versch. Plattformen
`java.io`	Ein-/Ausgabe-Ströme
`java.lang`	Vordefiniertes für die Sprache Java, z.B. die Klassen `Object` und `String`
`java.net`	Netzwerk-Unterstützung
`java.util`	zusätzliches Nützliches

Ein vollständiger Überblick über alle Java-Pakete der Versionen 1.0 und 1.1, ihre Klassen und Methoden findet sich unter den unten angegebenen Links (s. Abschnitt 7).

6. Unterschiede von Java zu C++

Java wurde mit Absicht C++-ähnlich gestaltet, um den Programmierern den Umstieg auf die neue Sprache zu erleichtern. Es gibt aber mehrere wichtige Unterschiede:

Java-Compiler erzeugen maschinenunabhängigen Bytecode, der auf der Zielmaschine vom Java-Interpreter ausgeführt wird. C und C++ Compiler generieren den direkten Maschinencode des Zielsystems. Dadurch sind Java-Programme sicherlich etwas langsamer als C++ - Programme. Andererseits sind sie aber auch besser portierbar. Für Benutzeroberflächen braucht man nicht unbedingt den schnellsten Code und zeitkritische Programmabschnitte kann man vor der Ausführung in direkten Maschinencode konvertieren. Die Bytecodes von Java werden etwas schneller interpretiert, als reine ASCII-Texte (Smalltalk, Perl etc.). Es ist ein Kompromiß zwischen Ausführungsgeschwindigkeit und Flexibilität.

C++ ist im wesentlichen ein C mit objektorientierten Erweiterungen - eine gemischte Sprache, die nicht unbedingt einen sauberen Programmierstil erzwingt. Java dagegen ist rein objektorientiert. Zwar existieren in Java auch primitive numerische Typen (int, long, float, char). Alles andere ist aber ein Objekt.

Zeichen in Java (`char`) sind nicht 8-Bit ASCII Zeichencodes, sondern 16-Bit Unicodezeichen. Java-Zeichen können in Arrays zusammengefaßt werden, aber sie sind deshalb noch keine Strings. Für Strings benutzt man die Java-Klassen `String` (Zeichenketten konstanter Länge und Inhalt) und `StringBuffer` (Zeichenketten variabler Länge). Java-Strings werden auch nicht durch ein NULL Zeichen terminiert.

Java hat keine Headerdateien und benötigt keinen Precompiler.

Java kennt keine impliziten Typumwandlungen, außer der Umwandlung von numerischen Datentypen in den selben Datentyp mit größerer Genauigkeit (z.B `int` nach `long`). Würde man jedoch bei der Konvertierung an Genauigkeit verlieren, so muß man eine explizite Typumwandlung vornehmen.

Java kennt keine Zeigerarithmetik. Es gibt nur Objekte und Arrays von Objekten. Alles, außer primitiven numerischen Typen, sind Objekte und die Variablen beinhalten Referenzen auf Objekte, nicht die Objekte selbst. Vor einem Zugriff

auf ein neues Objekt muß man dieses also vorher immer mit dem `new` Operator anlegen. Arraygrenzen werden zur Laufzeit überprüft. Ein Array kann nach dem Anlegen nicht mehr vergrößert oder verkleinert werden.

Alle Objekte und Arrays muß man explizit mit dem `new` Operator allokieren. Es gibt aber keinen delete-Operator. Speicher wird durch den Garbage-Collector zurückgefordert.

Alle primitiven Datentypen in Java haben einen festen Speicherbedarf und eine Genauigkeit, die nicht von der Zielplattform abhängig ist.

Es gibt keine Structs, Unions, Typedefs oder Enums, sondern nur Klassen.

Es gibt keine vorzeichenlosen Datentypen.

Wahrheitswerte lassen sich nicht in numerische (und umgekehrt) umwandeln.

Man kann die mathematischen Operatoren nicht überladen.

Java enthält direkten Support für Threads, kritische Programmabschnitte und Synchronisierung von Threads.

Java enthält keine goto-Befehle, kennt aber Labels. Mit einem `break`, `continue` oder `throw` Befehl wird aus einer verschachtelten Schleife herausgesprungen.

Java implementiert eine eigene Form der Mehrfachvererbung.

Java benutzt "late binding". Statt Methodenadressen werden Symbole benutzt, die erst während der Laufzeit durch direkte Adressen ersetzt werden.

7. Links

(ohne Anspruch auf Aktualität)

Die **Java-Homepage** von SUN ist:

```
http://java.sun.com
```

Ein *Mirror* findet sich an der Uni Aachen unter

```
http://sunsite.informatik.rwth-aachen.de/Mirror/java
```

Einige Seiten sind auch *lokal am Fachbereich* installiert unter:

```
http://www.mathematik.uni-marburg.de/~gasi/system/progs/java/
```

Hier finden sich Verweise zu:
- Java Development Kit
- JDK Dokumentation
- Java Tutorial
- Java Sprachspezifikation
- Java Paketbeschreibung (API)
- Java Klassenhierarchie
- Java Frequently Asked Questions (FAQ)

Es ist allerdings auf die verschiedenen Versionen (1.0/1.0.2, 1.1/1.1.1) zu achten.

Die Homepage von *Symantec Café* ist zu finden unter:

```
http://www.symantec.com
```

8. Literatur

[1] Arnold, Ken; Gosling, James "Java - Die Programmiersprache" Addison Wesley (Dtld.) 1996^2.

[2] Flanagan, David "Java in a Nutshell" O'Reilly & Associates Wesley, 1996.

[3] Seeger, Bernhard "Skript zur Vorlesung Informatik I" Universität Marburg, FB Mathematik, FG Informatik, WS 1996/97.

[4] Sommer, Manfred "Skript zur Vorlesung Rechnerkommunikation 2" Universität Marburg, FB Mathematik, FG Informatik, WS 1996/97.