Tips & Tricks zu Java

Markus Reitz

23. September 1999

Inhaltsverzeichnis

Kapitel 1

Allgemeines

1.1 Einleitung

Diese FAQ stellt allgemeine Hinweise und Antworten zu häufig gestellten Fragen zur Verfügung. Die FAQ lebt von der Beteiligung der User, deshalb sind Anregungen und Ergänzungen herzlich willkommen. Anregungen bitte per E-Mail an Markus_Reitz@yahoo.com. Die jeweils aktuelle Version dieser FAQ steht unter der Adresse

http://www.geocities.com/SiliconValley/Foothills/5270/JavaFAQ.html

zum freien Download bzw. als Online-Version zur Verfügung.

1.2 Copyright

Dieser Text wurde am 18.08.1999 von Markus Reitz erstellt. Er darf für den Privatgebrauch beliebig vervielfältigt und weitergegeben werden, solange dieser Hinweis bestehen bleibt und die Vervielfältigung und Weitergabe nicht im kommerziellen Rahmen erfolgt und inhaltlich keinerlei Veränderungen vorgenommen werden. Ansonsten bedarf es der ausdrücklichen Einwilligung des Autors. Auch die (teilweise) Aufnahme in andere Werke, in welcher Form sie auch vorliegen mögen, ist ohne ausdrückliche Genehmigung des Autors nicht zulässig. Ein Einrichten von Spiegelungen, d.h. die Abrufbarkeit von anderen als dem Geocities Server, dieses Dokuments bedarf der ausdrücklichen Genehmigung des Autors.

1.3 Wie stellt man Fragen in einer Newsgroup?

Newsgroups sind so etwas wie schwarze Bretter, an die jeder Informationen hängen kann, die er für andere wichtig hält. Newsgroups im Internet, die sich mit

Programmiersprachen oder Ähnlichem beschäftigen, enthalten häufig Fragen zu speziellen Themen der Programmiersprache. Jeder kann sich mit dem Problem beschäftigen, Kommentare und Lösungsvorschläge anbieten.

Postet man in eine Newsgruppe, so ist das Ziel, möglichst schnell möglichst effektive und klare Problemlösungen zu bekommen. Damit dies auch möglich ist, sollte man einige Regeln beachten:

- Nur in Newsgroups posten, die sich mit dem Thema beschäftigen. Es bringt nichts, eine Frage über HTML oder JavaScript in einer Java-Newsgroup zu stellen. Erstens müssen sich nicht unbedingt alle Leute, die die Java-Newsgroup lesen, damit auskennen und zweitens ist so ein Thema "off-topic", also in dieser Newsgroup am falschen Platz.

- Die Subject-Zeile des Newsgroup-Postings sinnvoll wählen. Subjects wie "Hilfe!!!!!!!", "Dringend!!!!!!" oder Ähnliches haben keine Aussagekraft und verhindern, daß sich viele Leute mit dem Thema beschäftigen. Subjects also so wählen, daß sie in wenigen Worten das Problem skizzieren.

- Postings sollten immer eine gültige E-Mail-Adresse haben. Im Zeitalter des Internet, in der praktisch jeder Zugang zu diesem neuen Medium hat, hat auch die Werbeindustrie das Netz als Möglichkeit kostenloser Werbung erkannt. Newsgroups werden gescannt und Werbe-E-Mails mit mehr oder minder (meist eher das Letztere) sinnvollen Inhalten versandt. Manche User versuchen sich dagegen zu schützen, indem sie in ihrer angebenen E-Mail-Adresse ungültige Zeichen angeben. Will man jemandem antworten, so muß man diese Zeichen erst entfernen. Dies ist alles andere als eine gute Methode, denn dadurch verhindert man, daß viele Antworten auf das Problem erhalten werden, weil sich kaum einer die Mühe machen will, die ungültige in eine gültige Mail-Adresse umzuwandeln.

- Kommentare wie "Antwort bitte per E-Mail an mich, da ich die Newsgroup nicht oder nur unregelmäßig lese", vermeiden. Erstens werden dann Antworten der Newsgroup-Gemeinde vorenthalten, die vielleicht von Interesse für alle sind und zweitens ist das Verhalten egoistisch - "kümmert euch um eine Lösung und schickt sie dann an mich, ich will mich mit der Suche nicht stressen"

- Komplette Sourcecodes vermeiden. "Ich habe folgende Applikation geschrieben und finde den Fehler nicht", sind vollkommen fehl am Platz und helfen auch nicht viel. Vielmehr den Sourcecode auf den Teil, in dem das Problem auftritt, beschränken und einen minimalen Code posten, der genau das Problem aufzeigt.

- Fragen konkret stellen. "Ich habe das Hello-World-Programm eingetippt und jetzt läuft es nicht" ist nicht aussagekräftig und wird dadurch zu wenigen oder gar keinen Antworten führen. Und die Antworten die kommen, können prinzipiell nur raten, weil zu wenig Informationen vorliegen.

Wenn man sich an diese Regeln hält, sollte man eigentlich relativ schnell eine Lösung für sein Problem erhalten - sofern die anderen eine Problemlösung finden können.

1.4 Inhalt der Newsgroup de.comp.lang.java

In der Newsgroup de.comp-lang.java sollen Probleme und Lösungen, die sich im Zusammenhang mit der Programmiersprache Java ergeben, diskutiert werden. Die Newsgroup beschäftigt sich nur mit der Sprache Java, Javascript oder herstellerspezifische Implementierungen (z.B. Microsoft J++) besitzen eigene Newsgroups, in denen diese spezifischen Probleme und Lösungen diskutiert werden. Neben der Newsgroup de.comp.lang.java gibt es noch weitere (englischsprachige) Newsgroups, die sich mit der Programmiersprache Java beschäftigen:

- comp.lang.java.3d
- comp.lang.java.advocacy
- comp.lang.java.beans
- comp.lang.java.corba
- comp.lang.java.databases
- comp.lang.java.gui
- comp.lang.java.help
- comp.lang.java.machine
- comp.lang.java.programmer
- comp.lang.java.security
- comp.lang.java.softwaretools

1.5 Tags - Konvention zu einem einheitlichen Aussehen von Postings

Neben den schon oben weiter angebrachten Ratschlägen, die allgemeine Anmerkungen zum Posten in Newsgroups darstellen, folgen nun einige spezifische Anmerkungen zur Schreibweise in der Newsgroup de.comp.lang.java. Vor einiger Zeit wurde die Einführung sogenannter Tags vorgeschlagen, die die Übersichtlichkeit steigern sollen. Hintergrund ist die einfache Idee, den Inhalt des Postings durch ein eindeutiges Schlüsselwort, das Tag, zu charakterisieren. Mit einem entsprechend konfigurierten Newsreader ist es dann möglich, sich solche Nachrichten hervorheben zu lassen, wodurch man direkt auf einen Blick Postings zum Beispiel zum Thema Swing überschauen kann.

Tags werden an den Anfang des Subjects in eckigen Klammern eingeschlossen geschrieben. Damit Einheitlichkeit gewahrt wird, sollten die folgenden Tags verwendet werden. Wenn jeder seine eigenen Tags spezifizieren würde, wäre der Sinn zerstört und genauso viel Chaos wie vorher, deshalb die Bitte sich an die Vorgaben zu halten. Bei Änderungsvorschlägen bitte in die Newsgroup zur Diskussion posten und sollten sich dann keine Widersprüche ergeben, werden die Vorschläge an dieser Stelle in die FAQ aufgenommen.

Liste der Tags in de.comp.lang.java:

- [INFO] - Allgemeine Informationen, z.B. Links auf Webseiten - *keine* Frage
- [DISCUSSION] - Diskussion zu einem Java-spezifischen Thema - *keine* Frage
- [ANNOUNCE] - Vorstellung neuer Software - *keine* Frage
- [LANG] - Frage bezüglich der Sprache Java, beispielsweise Vererbung
- [IO] - Frage bezüglich Eingabe/Ausgabe in Java
- [SWING] - Frage bezüglich Swing
- [BROWSER] - Frage zur Zusammenarbeit von Browser und JAVA-Programm
- [APPLET] - Frage zu Java-Applets
- [NONCORE] - Fragen zu Klassen/Packages, die über den Kern der Sprache hinausgehen, also Java3D etc.
- [TOOLS] - Frage zu einem Java-Tool, sei es IDE, java.exe o. ä.
- [VM] - Virtuelle Maschinen, alles über JREs & Co.
- [MISC] - Alles, was nicht in eine der anderen Rubriken paßt

1.6 Was ist Java?

Zuersteinmal: Java ist nicht Javascript. Javascript ist eine Sprache, die federführend von der Firma Netscape entwickelt wurde, um die Inhalte von Webseiten dynamischer und interaktiver zu gestalten. Javascript-Programme werden in den HTML-Quelltext der Seite eingebettet und vom Browser interpretiert und ausgeführt. Zwar macht Javascript viele Anleihen bei Java, ist aber bei weitem nicht so flexibel. Java wurde von der Firma Sun Microsystems mit dem Ziel entwickelt, eine moderne, objektorientierte Sprache zu schaffen. Durch das Ziel der Plattformunabhängigkeit ist Java vor allem im Zusammenhang mit dem

Entwickeln von Web-Applikationen im Internet eine der am häufigsten verwendeten Sprachen. Doch Java beschränkt sich nicht nur auf das Erstellen von Effekten für die Webseite, Java ist eine ausgewachsene Programmiersprache, mit der man alle anstehenden Probleme lösen kann. Die Syntax der Sprache ist an die von C++ angelehnt, Schlüsselwörter sind verändert bzw. in der Bedeutung erweitert worden, bestimmte Features von C++ wurden zu Gunsten der Übersichtlichkeit bzw. Sicherheit nicht in Java verwendet. Java Programme liegen im sogenannten Bytecode vor, der Maschinencode für einen fiktiven Prozessor, der von der VM, der virtuellen Maschine, ausgeführt wird. Durch die Verwendung des Bytecodes wird die Plattformunabhängigkeit von Java garantiert.

1.7 Erste Schritte in Java

Erfahrungen mit der Sprache Java zu sammeln ist nicht schwer. Sofern man einen Internetzugang hat, kann man sich die Sprache, Compiler etc. direkt von Sun besorgen. Es empfiehlt sich die Verwendung der Java-Implementierung von Sun, denn diese ist die Referenz für alle anderen Implementierungen und normalerweise die, die sich am aktuellsten Sprachstandard orientiert. Lösungen von Microsoft oder Symantec haben den Nachteil, daß Features modifiziert oder gar nicht implementiert werden oder Fähigkeiten hinzukommen, die der ursprünglichen Sprachimplementation fehlen. Außerdem bietet das JDK (Java Development Kit) von Sun den Vorteil, daß es kostenlos verfügbar ist. Nachteil: Compiler und Tools arbeiten kommandozeilenorientiert, d.h. man schreibt den Quellcode mit einem beliebigen Editor, speichert die Datei und führt dann den Compiler aus. Bei Fehlern lädt man die Datei, korrigiert den Fehler und das ganze Spiel beginnt von vorne. Abhilfe schaffen IDEs (Integrated Development Environments), die das JDK steuern und Fehler und Meldungen direkt anzeigen, ohne daß der Quelltexteditor verlassen werden muß. Ein weiterer Vorteil von IDEs ist das Feature *Syntax Highlighting*, daß die Schlüsselwörter der Sprache farblich hervorhebt und dadurch die Übersichtlichkeit steigert.

Bezugsquelle für das JDK:

- http://www.developer.java.sun.com

Anmerkung: Um die aktuellen Versionen zu erhalten, muß man sich zuvor registrieren lassen. Die Registrierung ist vollkommen kostenlos und nach erfolgreicher Registrierung erhält man Zugang zu den aktuellsten Versionen und Tools.

1.8 IDEs für Java

IDEs für Java findet man fast schon wie Sand am Meer und auch das Preisspektrum reicht von einigen zehn Mark bis in die Tausende. Um in die Sprache hineinzuschnuppern, empfiehlt sich die Verwendung des JDKs von Sun in Kombination mit einer IDE, die die Features direkt unterstützt. Eine solche IDE, die auch preislich ein nicht zu tiefes Loch in die Tasche reißt, ist KAWA von Tek-Tools. Man kann sich kostenlos die aktuellste Version zum Testen herunterladen, die Testphase ist jedoch begrenzt, sollte jedoch ausreichen, um sich für oder gegen die IDE bzw. für oder gegen Java zu entscheiden.

Anmerkung: Für Studenten, die an der Uni einen Java-Kurs belegen, ist es möglich, einen Freischaltungscode zu bekommen, der die kostenlose Nutzung der IDE für drei Monate gestattet.

Bezugsquelle für KAWA:

- http://www.tek-tools.com/ - Die Seite von Tek-Tools
- http://www.olabs.com/kawa/ - Eine Mirror-Seite, falls die Tek-Tools-Seite mal nicht funktioniert

Natürlich gibt es neben KAWA noch eine Reihe anderer IDEs, die aber teilweise den Nachteil haben, daß Testversionen nicht verfügbar sind. Auf den Seiten der Hersteller erfährt man jedoch gegebenenfalls näheres hierzu.

Quellen weiterer IDEs:

- Apptivity (Progress) -http://www.apptivity.com
- CodeWarrior Professional - http://www.metrowerks.com
- GRASP (frei verfügbar) - http://www.eng.auburn.edu/grasp
- Grinder - http://www.tpex.com
- Java WorkShop (Sun) - http://www.sun.com/workshop/java
- Javelin, Visual Object Development for Java - http://www.stepahead.com.au
- JBuilder (Inprise) - http://www.inprise.com/jbuilder
- JDE for emacs - http://sunsite.auc.dk/jde/
- NetBeans (Swing-based) - http://www.netbeans.com
- PARTS alpha (ObjectShare) - http://www.objectshare.com
- PowerJ (Sybase) - http://www.sybase.com/products/powerj
- SilverStream - http://www.silverstream.com
- Super Mojo (Penumbra) - http://www.penumbrasoftware.com
- SuperCede (Asymetrix) - http://www.supercede.com

- teikade (PFU Ltd) - http://www.pfu.co.jp/teikade
- Together/J (Object Intl Inc.) - http://www.oi.com
- Visaj (Imperial SW Tech) - http://www.imperial-software-tech.co.uk
- VisualAge (IBM) - http://www.software.ibm.com/ad/vajava
- Visual Cafe (Symantec) - http://cafe.symantec.com
- Xelfi - http://www.xelfi.com
- AnyJ - http://www.netcomputing.com

1.9 Literatur/Infos rund um Java

JDK und eine IDE sind vorhanden und was jetzt noch fehlt, ist Literatur, die sich mit der Java-Programmierung beschäftigt. Auch hierfür bietet das Internet eine fast unerschöpfliche Quelle an Informationen, man muß nur wissen, wo man suchen muß. Deshalb eine kleine Auflistung von Texten rund um die Java-Programmierung:

Online-Bücher

- http://www.developer.com/reference/r_java.html - verschiedene Bücher zum Thema Java
- http://www.mut.com/leseecke/default.htm - Java 1.2 in 21 Tagen
- http://rubens.math.uni-potsdam.de/%7eassmus/java/httoc.htm - ein Mirror zu Java 1.2 in 21 Tagen
- http://www.boku.ac.at/javaeinf/ - Java-Einführung von Hubert Partl
- http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/java/kurs/Kurs.html - Java-Kurs von Rene Grothmann
- http://java.sun.com/docs/books/tutorial/index.html - Das Java-Tutorial direkt von Sun
- http://www.addison-wesley.de/Service/Krueger/javabuch.htm - Java 1.1 lernen - Onlineversion des Buches von Guido Krüger
- http://www.gkrueger.com/books/k99a.html - GoTo Java 2 - das neueste Buch von Guido Krüger
- http://www.bruceeckel.com/javabook.html - Thinking in Java von Bruce Eckel

- http://sunsite.net.edu.cn/~cc/tutorials/se_java2e/httoc.htm - Using Java - Special Edition
- http://mindprod.com/jgloss.html - Java Glossary
- http://free-ed.net/fr03/fr030205.htm - Liste von kostenlosen Online-Büchern über Java
- http://infosoc.uni-koeln.de/akademie/java/ - Sammlung von Material zum Thema Java
- http://online-books.oreilly.com/ - Online Bücher von O'Reilly

Allgemeine Links zum Thema Java:

- http://www.jars.com/
- http://www.javacats.com/
- http://www.gamelan.com/
- http://www.rhein-neckar.de/~cetus/
- http://www.freecode.com/
- http://www.developer.com/
- http://www.progsource.com/
- http://www.devx.com/default.asp
- http://www.kawo1.rwth-aachen.de/~wormy/java.html

Java Ring:

- http://members.tripod.com/~mpTOOLS/ring.html

Online-Magazine:

- http://java.seite.net/ - Kaffe & Kuchen - Deutschsprachiges Java-Magazin

1.10 Erweiterungen der Sprache Java

Java ist noch eine relativ junge Sprache und immer noch in der Entwicklungsphase begriffen. Manche Features, die man von anderen Sprachen, wie zum Beispiel C++ kennt, wird man in Java schmerzlich vermissen. Hierzu zählen zum Beispiel Operatorenüberladung und Templates. Es gibt aber Preprozessoren, die die Sprache um diese Features erweitern, solange diese Features (noch?) nicht im offiziellen Sprachumfang enthalten sind.

- http://www.cis.unisa.edu.au/˜pizza/ - Templates und andere Features für die Sprache Java
- http://www.winternet.com/˜gginc/jfront/ - Operatorenüberladung für Java
- http://www.forestro.com/kiev/ - Kiev Java Compiler - Verbindung der Sprachen Java und Prolog
- http://theoretica.informatik.uni-oldenburg.de/˜jawa/ - Erweiterung von Java mit Assertions (analog zu Eiffel)
- http://www.reliable-systems.com/tools/iContract/iContract.htm - Erweiterung um "Design by contract" (ähnlich Eiffel)

1.11 Reverse-Engineering für Java

Wie für jede andere Sprache auch, so gibt es auch für Java Reverse-Engineering Tools, die aus dem vom Java-Compiler generierten Bytecode wieder ein Java-Programm erzeugen können. Damit ist es möglich auf die in dem Programm verwendeten Algorithmen Rückschlüsse zu ziehen.

Decompiler für Java:

- http://www.geocities.com/SiliconValley/Bridge/8617/jad.html - Ein De-Compiler in C++ geschrieben
- http://www.ahpah.com/ - SourceAgain - ein kommerzieller DeCompiler

Auf der anderen Seite gibt es sogenannte Code-Obfuscatoren, die eine DeCompilierung verhindern oder zumindest erschweren sollen.

Code-Obfuscatoren für Java:

- http://www.4thpass.com/SourceGuard - ein kommerzieller Obfuscator
- http://www.primenet.com/˜ej/ - freier Obfuscator

- http://www.math.gatech.edu/˜mladue/HoseMocha.java - freier Obfuscator
- http://www.monmouth.com/˜neil/Obfuscate.html -freier Obfuscator

1.12 Tools, die aus Java-Programmen native Programme erzeugen

Einer der Vorteile von Java ist die Plattformunabhängigkeit der damit erstellten Programme aufgrund des portablen Bytecodes. Dies muß allerdings mit einer verminderten Ausführungsgeschwindigkeit bezahlt werden. Braucht man aber eine hohe Ausführungsgeschwindigkeit, insbesondere dann, wenn man zeitkritische Programme entwirft, dann kann man den Bytecode in den Maschinencode des verwendeten Prozessors übersetzen lassen.

Native Übersetzungsprogramme:

- Symantec Visual Cafe - http://cafe.symantec.com/
- SuperCede - http://www.supercede.com/
- Tower Techology TowerJ - http://www.twr.com/java/java-products.html
- IBM Visual Age for Java - http://www.software.ibm.com/ad/vajava/
- Instantiations JOVE - http://www.instantiations.com/jove.htm
- Cygnus Sourceware for Java - http://sourceware.cygnus.com/java/

1.13 Code-Standards

Um die Übersichtlichkeit des Quelltextes zu garantieren, werden vor allem in größeren Firmen sogenannte Code-Standards festgelegt, die die schnelle Einarbeitung in vorhandene Quelltexte garantieren sollen. Dabei gibt es soetwas wie einen einheitlichen Standard nicht, fast jeder kocht hier sein eigenes Süppchen. Dementsprechend gibt es auch viele unterschiedliche Quellen, mit deren Hilfe man sich jedoch einen eigenen Standard festlegen oder an dem man sich direkt orientieren kann.

- http://v2ma09.gsfc.nasa.gov/coding_standards.html - Coding Standards für C, C++ und Java
- http://www.numeric-quest.com/news/NQ-comments.html - Anregungen für den Java Programmierstil

- http://gee.cs.oswego.edu/dl/html/javaCodingStd.html - Doug Lea's Draft Java Coding Standard
- http://www.redbooks.ibm.com/SG247006/codguide.htm - Java Beans Coding Guidelines
- http://www.ambysoft.com/javaCodingStandards.html - Java Coding Standards (AmbySoft)
- http://www.chimu.com/publications/javaStandards/ - Java Development Standards
- http://reality.sgi.com/sandvik/JavaGuidelines.html - Kent Sandvik's Java Coding Style Guidelines
- http://dubhe.cc.nps.navy.mil/~java/course/styleguide.html - Naval Postgraduate School Java Style Guide
- http://developer.netscape.com/library/technote/java/codestyle.html - Netscapes Software Coding Standards Guide für Java
- http://java.sun.com/docs/codeconv/index.html - SUNs Java Code Conventions

1.14 Verwandtschaft von Java mit anderen Sprachen

Java bietet eine Reihe von neuen Features, orientiert sich aber auch an etablierten Sprachen wie C++, Smalltalk oder Objective-C. Prinzipien aus diesen Sprachen werden übernommen und teilweise erweitert. Die folgende Tabelle gibt einen kleinen Überblick über Features von Java, die aus anderen Sprachen quasi "entliehen" wurden.

	C++	Smalltalk	Objective-C
Statische Typen	⋆		
Universelle Basisklasse Object		⋆	
Garbage Collection		⋆	
Konstruktoren	⋆		
Primitive Datentypen	⋆		
Bibliothek von Standardklassen		⋆	
Interfaces			⋆

Im Vergleich zu C++ besitzt Java folgende Unterschiede:

- keine Templates

- keine Operatorenüberladung
- kein prozeduraler Overhead
- keine Mehrfachvererbung

Kapitel 2

Häufig gestellte Fragen - FAQ

2.1 Java findet (scheinbar!) den Konstruktor nicht

```
class Test
{
  public void Test(int a, int b)
  {
    ...
  }

  public static void main(String[] Param)
  {

Test EinObjekt=new Test(12,13) //und hier die Fehlermeldung
    ...
  }
}
```

Versucht man die folgende Klasse zu übersetzen, so meckert der Compiler mit der Meldung "Wrong number of arguments in constructor ...". Die Lösung des Problems ist einfach: *Konstruktoren besitzen keinen Rückgabetyp, auch nicht void.* In obigem Beispiel hat man nämlich nicht den Konstruktor für die Klasse Test definiert, sondern eine Funktion, die den selben Namen wie die Klasse hat. Weiter unten versucht man nun, den vermeintlich definierten Konstruktor aufzurufen und bekommt die Fehlermeldung.

2.2 Die *equals(...)*-Methode funktioniert (scheinbar!) nicht

```
class Test
{
  public boolean equals(Test Second)
  {
    ...
  }
  ...
}
```

Verwendet man nun Objekte dieser Klasse in Containerklassen und hier insbesondere Methoden, die auf die Methode *equals(...)* des Objekts zurückgreifen, so funktioniert dies nicht. Der Grund liegt in der falschen Signatur der Methode *equals(...)*. Der Paramter muß vom Typ Object sein und nicht vom Typ der Klasse, zu dem die Methode gehört. Ansonsten existieren für die Klasse Test zwei Versionen der *equals(...)*-Methode. Eine, die von der Klasse Object geerbt wurde und die als Parametertyp auch Object besitzt und eine zweite, nämlich die oben definierte. Containerklassen verwenden aber die erste und da diese nicht verändert wurde, wird nicht das gewünschte Ergebnis erzielt.

2.3 Java rechnet (scheinbar!) falsch

Läßt man folgendes Programm laufen

```
class Test
{
  public static void main(String[] Param)
  {
    float Zahl=0.0f;
    int i;

    for(i=1;i<=100;i++)
    {
      Zahl+=0.1f;
      System.out.println(Zahl);
    }
  }
}
```

so ergibt sich folgende Ausgabe:

```
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.70000005
0.8000001
0.9000001
...
```

Scheinbar rechnet Java an einigen Stellen falsch und ist nicht fähig, zu einer Zahl den Wert 0.1 korrekt zu addieren. Dies scheint jedoch nur so und ist auch keineswegs typisch für die Programmiersprache Java, sondern ein allgemeines Problem.

Um dies zu verstehen, muß man sich klarmachen, daß Zahlen intern in Form von Binärzahlen und Fließkommazahlen mit Hilfe von Binärbrüchen dargestellt werden. Nun kann der (nicht gerade seltene) Fall eintreten, daß sich eine Zahl im Dezimalsystem zwar darstellen läßt, in Binärdarstellung jedoch zu einem unendlichen Bruch wird. Für die Speicherung einer Zahl steht aber nur ein beschränkter Speicherplatz zur Verfügung, d.h. die unendliche Binärdarstellung wird nur bei zu einer gewissen Stelle gespeichert. Das Resultat sind Ungenauigkeiten. Addiert man jetzt solche Zahlen (und 0.1 ist zum Beispiel eine solche) mehrfach auf, so addieren sich die unvermeidbaren Ungenauigkeiten immer mehr auf und führen zu dem obigen Verhalten.

Dies ist keineswegs charakteristisch für Java, sondern auch in jeder anderen Sprache, die Fließkommazahlen verwendet, reproduzierbar.

Eine Abmilderung des Problems besteht in einer "schlauen" Rundung von Zwischenergebnissen an geeigneten Stellen der Berechnung, sodaß Fehler kompensiert oder zumindest abgeschwächt werden.

2.3.1 Runden von Fließkommazahlen

Wenige Zeilen weiter oben wurde davon gesprochen, mögliche Rechenfehler auszugleichen, indem man an geeigneten Stellen rundet. Wie aber rundet man eine Fließkommazahl? Nun, das Prinzip hat sich seit den Tagen des C-64 nicht geändert:

- Verschiebe das Komma um die gewünschte Anzahl an Stellen nach rechts
- Addiere zu der neuen Zahl 0.5 hinzu und schneide die Nachkommastellen ab

- Verschiebe das Komma um die selbe Anzahl an Stellen wieder nach links

Berücksichtigt man, daß das Verschieben des Kommata um n Stellen nach rechts der Multiplikation der Zahl mit 10^n entspricht und die Verschiebung nach links in analoger Weise durch die Division durch 10^n erledigt wird, kann man die Rundungsfunktion wie folgt formulieren:

```
...
double runden(double z)
{
  return(((int)z*Math.pow(10,n))/Math.pow(10,n));
}
```

2.4 Ich kann mein Java-Programm nicht ausführen/compilieren, was mache ich falsch?

2.4.1 NoClassDefFoundError bei Ausführunsversuch mit java

Der Grund ist einfach: Man muß der Java-Laufzeitumgebung mitteilen, wo sich die *.class-Dateien des Programms befinden, damit die Ausführung erfolgen kann. Dies erreicht man bei SUN mit dem Aufrufparameter -cp für Classpath oder man setzt die Variable CLASSPATH auf das Verzeichnis, in dem sich die *.class-Dateien befinden.

2.4.2 Couldn't read <Name>

In Java gilt die einfache Regel, daß der Name der Datei, in der die Klasse gespeichert ist, identisch sein muß mit dem Klassennamen, wobei auf die korrekte Groß- und Kleinschreibung zu achten ist.

2.5 Die *equals(...)*-Methode der Klasse funktioniert nicht

Man hat eine eigene Klasse entworfen und möchte nun testen, ob zwei Objekte gleich sind. Fein, denkt man sich, dafür bietet die Klasse *Object* ja die Methode *equals(...)*. Doch das Programm mit Verwendung der *equals(...)*-Methode wird zwar korrekt übersetzt, der Vergleich funktioniert jedoch nicht so, wie er eigentlich sollte.

Die Methode *equals(...)* muß für jede Klasse neu überschrieben werden. Schließlich kann man nicht in allgemeiner Weise die Gleichheit zweier Objekte einer Klasse spezifizieren, theoretisch kann der Test auf Gleichheit bei jeder Klasse anders ablaufen. Deshalb muß die Routine überschrieben werden. Dies gilt auch bei der Verwendung von selbstgeschriebenen Objekten innerhalb von Containerklassen, denn viele Funktionen eines Containers werden mit Hilfe der *equals(...)*-Methode realisiert.

2.6 Wie vergleiche ich zwei Strings in JAVA?

2.6.1 Die Standardantwort

Man versucht zwei Strings in der Form

```
if (String1==String2)
```

zu vergleichen und erhält alles andere als ein richtiges Ergebnis. Der Grund ist der, daß mit dem "=="-operator nur die beiden Referenzen der Objekte miteinander verglichen werden, nicht jedoch der eigentliche Inhalt. Man erhält deshalb auch bei zwei gleichen Strings die Aussage, daß sie verschieden sind. Für den inhaltlichen Vergleich, nicht nur von Strings, sondern allgemein von Objekten, wird in Java die Methode *equals* verwendet, die nicht Referenzen, sondern Daten vergleicht. Obige Abfrage müßte also

```
if (String1.equals(String2))
```

lauten, damit das gemacht wird, was eigentlich gewünscht ist. Im Zusammenhang mit Strings ist noch eine Besonderheit zu erwähnen:

```
if ("JAVA".equals(String2))
```

ist zulässig, der Compiler erzeugt aus der Zeichenkette automatisch ein String-Objekt; man muß also nicht zuerst ein Objekt anlegen und den String *JAVA* dort hinein speichern.

2.6.2 Die Alternative - Verwenden der Methode *intern()*

Es gibt noch eine weitere Möglichkeit, zwei Strings miteinander zu vergleichen. Die Klasse String bietet die Methode *intern()* und in Zusammenarbeit mit dieser Methode ist es dann möglich, zwei Strings doch mit Hilfe des "=="-Operators zu vergleichen. Um dies zu verstehen, muß man wissen, was die Methode *intern()* eigentlich macht. Dazu nehmen wir an, daß s ein String sei.

```
s=s.intern()
```

Wird die Methode aufgerufen, so wird zuerst nachgeschaut, ob der String, den *s* enthält, schon im sogenannten String-Pool gespeichert ist. Ist dies der Fall, so liefert die Methode die Referenz auf den String in diesem Pool zurück und durch obige Zeile verweist nun auch *s* auf diesen String im Pool. Ist der String noch nicht im Pool vorhanden, so wird er zum Pool hinzugefügt und s zeigt auf diesen String, es ändert sich also nichts. Durch die Methode *intern()* wird also gewährleistet, daß gleiche Strings immer die gleiche Referenz haben, denn ein String existiert immer nur genau einmal im String-Pool und alle String-Referenzen verweisen auf diesen String.

Durch diese Tatsache kann man nun zwei Strings vergleichen, indem man die beiden Referenzen miteinander vergleicht. Diese Methode ist natürlich wesentlich schneller als ein herkömmlicher Vergleich mit der Methode *equals(...)*, denn hierbei muß nur eine Referenz verglichen werden, während bei der Methode *equals(...)* Zeichen für Zeichen der beiden Strings verglichen werden muß, was wesentlich zeitaufwendiger ist.

2.7 Gibt es in Java keine Zeiger wie in C++?

Im Prinzip gibt es ein Konstrukt, das den Zeigern anderer Programmiersprachen, wie zum Beispiel C++, sehr ähnlich ist, die sogenannten Referenzen. Referenzen können, im Vergleich zu Zeigern in C++, nicht manipuliert werden; die sogenannte Zeigerarithmetik, bei der man Zeiger auf beliebige Speicherinhalte zeigen lassen kann, wurde aus Sicherheitsgründen nicht in Java übernommen. Der Inhalt einer Referenz kann einer anderen Referenz zugewiesen werden, Referenzen können miteinander verglichen werden. Wird in Java eine Objektvariable angelegt, so ist dies nichts weiter als ein Speicherplatz für eine Referenz für ein Objekt des angegebenen Typs. Der *new*-Operator erzeugt das Objekt und liefert die Referenz darauf zurück, die dann in der Objektvariablen gespeichert wird.

2.8 Referenz $\neq$Referenz?

```
class Test
```

```
{
  public static void main(String Param[])
  {
    String a="A";
    String x;

    x=a;
    System.out.println(x);
    a="B";
    System.out.println(x);
  }
}
```

so sollte die Ausgabe doch eigentlich

```
A
B
```

lauten, denn die Variable x speichert eine Referenz auf den String a. Im ersten Fall hat a den Wert "A" und damit auch x, das ja auf diesen String verweist. Im zweiten Fall wird a geändert und damit müßte sich doch auch eigentlich der Wert von x ändern, weil x auf a verweist. Die Ausgabe, die das Programm liefert, ist jedoch

```
A
A
```

was eigentlich nicht in das Bild einer normalen Referenz paßt. Der Fehler liegt darin begründet, daß a *nicht* der eigentliche String ist, sondern nur eine Referenz auf diesen String. Durch $x{=}a$ referenzieren beide Variablen den selben String und durch $a{=}...$ verweist a auf einen anderen String. Dies ändert jedoch nichts an der Referenz, die in x gespeichert ist. Somit ist die Ausgabe völlig korrekt.

2.9 Werden in Java Funktionsparameter per Referenz oder per Wert übergeben?

Obwohl Java eine objektorientierte Sprache ist, existieren auch weiterhin sogenannnte elementare Datentypen wie Integerzahlen oder Fließkommazahlen, die sich vom Aufbau her an der Hardware orientieren, jedoch trotzdem prozessorunabhängig definiert sind. Man muß also bei der Übergabe an Funktionen zwischen Objekten und eben diesen elementaren Datentypen unterscheiden. Dabei gilt:

- Objektvariablen sind nichts anderes als Referenzen auf Objekte und werden deshalb im Prinzip als Referenz übergeben

- elementare Datentypen werden per Wert an die Funktion übergeben

Bei Objektvariablen steht deshalb "im Prinzip", da die Referenz an sich eine Art elementarer Datentyp ist und deshalb der Wert kopiert und übergeben wird. Betrachtet man das Ganze jedoch vom Standpunkt eines Objektes aus, so wird an die Funktion nur die Referenz übergeben und deshalb kann man sagen, daß Objektübergabe eine Referenzübergabe ist.

2.10 Was bedeutet das Schlüsselwort *final*?

Klassen oder Methoden, die das Schlüsselwort *final* tragen, können nicht mehr überschrieben werden, wenn von dieser Klasse abgeleitet wird. Die Verwendung dieses Schlüsselwortes bietet sich aus zwei Gründen an:

- Realisierung von Sicherheitsmaßnahmen

- Codeoptimierung

Sicherheitsmaßnahmen werden realisiert, weil es nicht möglich ist, die Bedeutung der Methode in abgeleiteten Klassen zu verändern und damit bestehende Konzepte zu durchbrechen. Codeoptimierung deshalb, weil der Compiler nun davon ausgehen kann, daß sich an den Methoden nichts mehr ändern wird und deshalb elegantere Codeoptimierungen möglich sind. In Anbetracht der Tatsache, daß *final*-Methoden nicht mehr verändert werden können, muß man sich beim Programmentwurf sehr sicher sein, daß das Feature des Überschreibens definitiv nicht für diese Methode benötigt wird, ansonsten kann es bei Verbesserungen des Codes zu Problemen kommen.

2.11 Warum gibt es in Java nicht Destruktoren so wie in C++?

In Java hat der Programmierer unter normalen Umständen keinen direkten Einfluß darauf, wann Objekte aus dem Speicher entfernt werden. So ist es unter anderem nicht möglich, ein Objekt per Befehl aus dem Speicher zu löschen. Dies erledigt der sogenannte Garbage-Collector, der Objekte aus dem Speicher entfernt, auf die keine Referenz mehr verweist. Dadurch wird verhindert, daß der Programmierer mehr oder minder mutwillig Speicherfehler erzeugen kann, die das Programm zum Absturz bringen könnten. Wann die Entfernung aus dem

Speicher erfolgt, liegt im Ermessen des Computers. Es gibt also keinen definierten Zeitpunkt, wann ein Objekt nicht mehr existiert und deshalb ist es in den meisten Fällen nicht sinnvoll, Operationen zu definieren, die beim Löschen des Objektes ausgeführt werden. Man könnte (ungültige) Annahmen voraussetzen - zum Beispiel bei verketteten Listen, daß das Nachfolgerelement noch existiert, obwohl dies gar nicht der Fall ist - und damit Fehler verursachen. Destruktoren wie in C++ existieren deshalb nicht. Ist es aus irgendeinem dringenden Grund dennoch nötig, Operationen beim Löschen des Objektes auszuführen, so kann man eine Methode *finalize* definieren, die bei der Speicherbereinigung abgearbeitet wird.

2.12 Gibt es soetwas wie den *const*-Modifizierer von C++ auch in Java?

Ab Java 1.1 können Argumente mit dem Modifizierer *final* als konstant definiert werden. Bei einer Objektreferenz bedeutet dies allerdings nur, daß die Referenz konstant bleibt, nicht aber unbedingt das Objekt, auf das die Referenz verweist.

2.13 Probleme mit final-Werten

Angenommen, eine Klasse A ist wie folgt definiert

```
class A {final static boolean test=true}
```

Aus irgendeinem Grund wird die Definition der Klasse A auf Folgendes geändert:

```
class A {final static boolean test=false}
```

Die Klasse A wird neu compiliert und eine neue Klassendatei erzeugt. Alle anderen Klassen, die das in Klasse A definierte Feld verwenden, müßten nun mit dem korrigierten Wert *false* arbeiten. Probiert man dies in der Praxis aus, so stellt man fest, daß dies nicht der Fall ist. Der Grund liegt in der Sprachdefinition der Sprache Java. Ein primitiver (elementarer) Datentyp, der das Modifiziererpaar *final static* trägt, wird vom Compiler als Konstante behandelt und kann überall, wo er im Programm auftritt, vom Compiler durch den Wert ersetzt werden. Um also zu erreichen, daß sich die in der Klasse A gemachte Änderung auf alle sie benutzenden Klassen auswirkt, müssen alle diese Klassen neu übersetzt werden.

2.14 Der dynamische Typ der Parameter wird nicht zum Auswählen der überladenen Funktion verwendet

Dies ist ein korrektes Verhalten gemäß der Java-Sprachspezifikation. Wird auf ein Objekt eine Methode angewendet, dann wird an Hand des Objekts entschieden, welche Methode aufgerufen werden soll. Dabei ist es egal, wie der Typ der Referenz ist, die auf das Objekt zeigt. Dieses Verhalten ist der sogenannte Polymorphismus (Vielgestaltigkeit). Zu beachten ist, daß diese dynamische Auswahl nur bei der Methodenauswahl geschieht. Bei polymorphen Funktionen entscheidet nicht der Typ des an die Methode übergebenen Objektes über die auszuwählende Methode, sondern der Typ der Referenz, die auf das Objekt verweist, wird zur Selektion der Methode herangezogen. Der Compiler muß zur Compilierzeit entscheiden, welche Methode ausgewählt wird und eine Aussage kann er deshalb nur über den Typ der Referenz machen, da dies das Einzige ist, was zur Compilierzeit definitiv feststeht.

Das Ganze demonstriert ein Beispielprogramm:

```
class BasisKlasse {}
class AbgeleiteteKlasse extends BasisKlasse {}

public class Test
{
  public static void Methode(BasisKlasse EineKlasse)
  {
     System.out.println("Basisklassenmethode!");
  }


public static void Methode(AbgeleiteteKlasse EineKlasse)
  {

System.out.println("Methode in der abgeleiteten Klasse!");
  }

  public static void TestMethode(BasisKlasse A)
  {

if (A instanceof AbgeleiteteKlasse)  System.out.print("Abgeleitet: ");
        else System.out.print("Basis: ");
```

```
Methode(A);   // Methode der Basisklasse oder der abgeleiteten Klasse?
  }

  public static void main(String Param[])
  {
    BasisKlasse A = new BasisKlasse();
    AbgeleiteteKlasse B = new AbgeleiteteKlasse();

    TestMethode(A);
    TestMethode(B);
  }
}
```

Das Programm erzeugt folgende Ausgabe:

```
Basis: Basisklassenmethode!
Abgeleitet: Basisklassenmethode!
```

2.15 Was bedeutet super(...)?

Werden Ableitungen von Klassen gebildet und dabei Funktionen redefiniert (überschrieben), so ist es in vielen Fällen nötig, auf die Funktionalität der Basisklasse zurückzugreifen. Mit *super.Methode(...)* teilt man mit, daß man die Methode der Basisklasse und nicht die Methode der aktuellen Klasse benutzen will. Innerhalb eines Konstruktors ist es möglich mit *super(...)* den Konstruktor der Basisklasse aufzurufen. Findet im Konstruktor kein expliziter Aufruf mit *super(...)* statt, so wird automatisch der parameterlose Konstruktor der Basisklasse aufgerufen.

2.16 Beim Compilieren erhält man Meldungen der Form "... is deprecated"

Diese Meldung erhält man normalerweise dann, wenn ältere Java-Programme mit einer neuen Version des Compilers übersetzt werden sollen. Die Meldung ist nur eine Warnung, also kein richtiger Fehler, die Funktionalität des Programms wird nicht beeinträchtigt. Die Methode, die bei der Meldung "... is deprecated" genannt wird, sollte aber nach Möglichkeit nicht mehr im Programm verwendet werden, da das Konzept der Klasse verändert wurde und deshalb die Methode unter Umständen in neueren Versionen nicht mehr implementiert sein wird. Treten also solche Meldungen beim Compilieren auf, so sollte man die entsprechenden Methoden durch die empfohlenen Varianten ersetzen, damit gewährleistet ist, daß das Programm auch mit späteren Versionen von Java einwandfrei übersetzt werden kann.

2.17 Programme die Serialisierung verwenden, sind langsamer

Java bietet mit dem Mechanismus der Serialisierung auf einfache Weise die Möglichkeit, den aktuellen Zustand des Objekts auf einem Datenträger zu sichern. Der zu Grunde liegende Mechanismus ist relativ kompliziert, damit auch alle auftretenden Möglichkeiten korrekt behandelt werden können. Sind die Daten der Objekte sehr groß, so empfiehlt es sich, eigene Prozeduren zur Speicherung zu entwickeln, die dann nicht mehr allgemeingültig, aber auf die aktuelle Verwendung angepaßt und damit schneller sind.

2.17.1 Initialisierung von Objekten

```
public class Test
{
  public static void main(String Param[])
  {
    B EineKlasse[] = new B[10];
    for (int i=0; i<10; i++)

EineKlasse[i].Text = "Test"; //NullPointerException
  }
}

class B
{
  String Text;
}
```

Führt man das Programm aus, so erhält man die im Programmtext erwähnte Exception, obwohl doch alles eigentlich ganz in Ordnung aussieht. Der Teufel steckt aber im Detail. Mit *B EineKlasse[]=new B[10]* wird eben nicht ein Array von zehn Objekten vom Typ B erzeugt, sondern nur ein Array, welches zehn *Referenzen* auf Objekte vom Typ B enthält, die eigentlichen Objekte werden damit nicht erzeugt.

Folgende modifizierte Version leistet dagegen das Gewünschte:

```
public class Test
```

```
{
  public static void main(String Param[])
  {
    B EineKlasse[] = new B[10];
    for (int i=0; i<10; i++)
    {
       EineKlasse[i]=new B();

EineKlasse[i].Text = "Test"; //keine Exception mehr
    }
  }
}

class B
{
  String Text;
}
```

2.18 Nicht initialisierte *final* Variablen

Folgender Java-Code ist gültig ab Java in der Version 1.1

```
public class Test
{
  final int x;

  public Test(int i)
  {

x = i;          // konstante Variable wird auf den Wert von i gesetzt
  }


public Test()          // Fehler - die Variable wird nicht initialisiert
  {
  }
}
```

Es ist also möglich, Objekte mit konstanten Daten zu erzeugen, deren konstanter Wert beim Erzeugen des Objekts individuell festgelegt werden kann. Allerdings kann dies nur innerhalb eines Konstruktors geschehen und es muß auf jeden Fall eine Initialisierung der Variablen erfolgen, denn ansonsten meldet der Compiler Fehler.

2.19 Anonyme Arrays

Ab Java 1.1 ist folgender Code gültig

```
int x[];
x = new int[] {1, 2, 3};
```

Es ist jetzt also möglich, Arrays auch außerhalb der Definition mit den gewünschten Werten zu initialisieren, indem man mit {...} einfach die Werte angibt, die das neue Array tragen soll.

2.20 Bedingte Übersetzung

In C++ existiert der sogenannte Pre-Prozessor, der es mit bestimmten Komandos erlaubt, Teile des Codes zu übersetzen und andere Teile bei der Übersetzung zu überspringen. Unter Umständen wäre es hilfreich, wenn auch Java eine solche Möglichkeit der Compilationssteuerung zuliese, doch einen Pre-Prozessor gibt es hier nicht. Man kann dies aber, zumindest ansatzweise, mit *if*-Statements nachbilden. Dazu definiert man eine boolsche Variable *DEBUG* und in Abhängigkeit von dieser Variablen sollen bestimmte Codeteile ausgeführt werden und andere nicht

```
class Test
{
  final static boolean DEBUG=true;

  public static void main(String Param[])
  {
    int x=12;


if (DEBUG) System.out.println("Der Wert von x ist "+x);
  }
}
```

Verwendet man nun noch Optimierer, die unbenutzten Code aus den Klassendateien entfernen, so hat man im Prinzip ein ähnliches Verhalten wie beim Pre-Prozessor von C++. In diesem Zusammenhang ist es noch erwähnenswert, daß das Verfahren nur bei der *if*-Abfrage möglich ist, denn das Java-System prüft normalerweise darauf, ob Codezeilen erreicht werden können oder nicht und gibt gegebenenfalls Fehlermeldungen aus. Das if-Statement ist jedoch wie oben beschrieben erweitert worden, um das gewünschte Verhalten simulieren zu können. Dahingegen wird folgender Code mit einer Fehlermeldung übersetzt

```
class Test
{
  final static boolean DEBUG=true;

  public static void main(String Param[])
  {
    int x=12;


while (DEBUG) System.out.println("Der Wert von x ist "+x);
  }
}
```

Denn es würde sich je nach dem Zustand von *DEBUG* eine Endlosschleife ergeben.

2.21 Operatorenprioritäten

```
public class Test
{
  public static void main(String Param[])
  {
    int x=10;
    int y=x++;

    System.out.println(y);
  }
}
```

Schaut man in der Priortätstabelle für Operatoren nach, so stellt man fest, daß der "++"-Operator eine höhere Priorität hat als der "="-Operator. Dann wäre es doch aber eigentlich ganz logisch, daß das obige Programmfragment den Wert 11 ausgibt. Ausgegeben wird aber der Wert 10. Tatsächlich wird der "++"-Operator vor dem "="-Operator ausgeführt. Der "++"-Operator ist jedoch nichts anderes als eine spezielle Art von Funktion, die natürlich auch einen Rückgabewert liefert und als Seiteneffekt den Wert der übergebenen Variable um eins erhöht. Und genau hier liegt der Grund, warum die Ausgabe 10 und nicht etwa 11 lautet: Der "++"-Operator liefert als Rückgabewert x und *nicht* x+1 (das ursprüngliche x!). Und da x den Wert 10 hat, bekommt auch y diesen Wert.

2.22 Wie wandle ich einen String in eine Integer-Zahl um?

Klassen können Methoden besitzen, die die Konvertierung eines Objekts dieser Klasse in ein Objekt einer anderen Klasse übernehmen. Zu dem elementaren Java-Datentyp *int* gibt es eine sogenannte Wrapper-Klasse *Integer*, die den elementaren Datentyp in einer Klasse kapselt. Diese Klasse stellt eine Methode (in diesem Fall eine Klassenmethode) zur Verfügung, die das Gewünschte leistet:

```
class Test
{
  public static void main(String Param[])
  {
    int Zahl;
    String ZahlString="50";

    Zahl=Integer.parseInt(ZahlString);

    ...
  }
}
```

2.23 Wie wandle ich eine Integer-Zahl in einen String um?

Dieses Problem ist genau das Gegenteil des vorherigen. Wie die Klasse Integer, so besitzt auch die Klasse String eine Methode, die das Problem löst, allerdings heißt die Methode nicht *parseString*, was man analog schließen könnte, sondern *valueOf*.

```
class Test
```

```
{
  public static void main(String Param[])
  {
    int Zahl=50;
    String ZahlString;

    try
    {
      ZahlString=String.valueOf(Zahl);
    }

    catch(NumberFormatException e)
    {

//Fehlerbehandlung für den Fall, daß der String keine Zahl war
    }

    ...
  }
}
```

2.24 Wie kann man ein Array mit einem bestimmten Wert belegen?

Häufig möchte man Arrays mit bestimmten Werten vorbelegen, doch leider gibt es in Java keine Standardfunktion, die diese Aufgabe erledigen könnte. Der naheliegenste Weg ist die Verwendung einer for-Schleife, die der Reihe nach alle Elemente durchgeht und den gewünschten Wert in jedes Array-Element schreibt, jdeoch ist dieser Weg insbesondere für große Arrays nicht unbedingt schnell.

Mit Hilfe der System-Routine *arraycopy* kann man aber einen Algorithmus aufstellen, der diese Aufgabe recht schnell erledigt. Dabei arbeitet der Algorithmus nach dem folgenden Prinzip:

Zuerst wird in das erste Feld der zu gewünschte Wert geschrieben. Mit Hilfe der *arraycopy*-Methode wird dann dieses Feld hinter die aktuelle Position kopiert. Dann wird dieser Bereich wieder hinten dran gehängt und nun sind schon vier Array-Einträge mit dem gewünschten Wert initialisiert. Bei erneutem Durchführen sind es acht Werte usw. D.h. die Anzahl der initialisierten Array-Einträge verdoppelt sich bei jedem Durchlauf. Handelt es sich bei der Anzahl der Elemente um eine Zweierpotenz, so ist das Array nach $\log_2 n$ Durchläufen mit dem gewünschten Wert initialisiert. Handelt es sich nicht um eine Zweierpotenz, so verbleibt noch ein uninitialisierter Bereich, welcher mit einem weiteren *arraycopy*-Aufruf initialisiert wird.

Das nachfolgende Programm zeigt die konkrete Implementation des beschriebenen Algorithmus

```
class Test
{
  public static void main(String Param[])
  {

int EinArray[]=new int[10000]; //Das zu initialisierende Array


int Startposition=0; //Der Index, ab dem initialisiert werden soll

int AnzahlElemente=10000; //Anzahl der zu initialisierenden Elemente
    int Wert=2; //Initialisierungswert

    int Zaehler=1;
    int AktuellePosition=Startposition+1;

    EinArray[Startposition]=Wert;
    AnzahlElemente--;

    while(AnzahlElemente>Zaehler)
    {

System.arraycopy(EinArray,Startposition,EinArray,AktuellePosition,Zaehler);
      AnzahlElemente-=Zaehler;
      AktuellePosition+=Zaehler;
      Zaehler*=2;
    }

    //Ist die Anzahl der Elemente keine Zweierpotenz,

//dann muessen im letzten Schritt noch die restlichen
    //Array-Eintraege initialisiert werden


System.arraycopy(EinArray,Startposition,EinArray,AktuellePosition,AnzahlElemente);
  }
}
```

2.25 Gibt es außer der Vererbung noch andere Beziehungen zwischen Objekten?

2.25.1 Vererbung

Vererbung ist in Java und natürlich auch in anderen Programmiersprachen nur eine Möglichkeit, um das konkrete Problem in den Computer zu übertragen. Das Prinzip der Vererbung ist dann anwendbar, wenn zwei Objekte in einer *ist-ein* oder *ist eine Art von* Beziehung zueinander stehen. Eine abgeleitete Klasse ist ein Subtyp der zugehörigen Oberklasse, besitzt also die Eigenschaften (Daten+Methoden) und erweitert diese bei Bedarf um neue. LKW und PKW sind zum Beispiel Subtypen der Oberklasse Automobil, wobei beim LKW zum Beispiel der Anhänger oder die Achsenanzahl hinzukommt.

Wendet man das Prinzip der Vererbung an, so gilt das sogenannte *Liskov'sche Substitutionsprinzip*:

Objekte der abgeleiteten Klasse können stets an die Stelle von Objekten der Oberklasse treten.

2.25.2 Aggregation

Das Prinzip der Aggregation besagt, daß ein Objekt aus mehreren Teilen (=Objekten) besteht, die wiederrum aus Teilen bestehen können usw. Die Klasse Computer könnte z.B. aus den Klassen Speicher, Festplatte etc. bestehen. In einem Pseudo-Java-Code etwa so formuliert:

```
class Computer
{
  Speicher FluechtigerSpeicher;
  Festplatte PermanenterSpeicher;
  ...
}
```

2.25.3 Assoziation

Mit der Assoziation wird die Verbindung des Objektes zu einem oder mehreren anderen Objekten beschrieben. Assoziationen können kurzfristig sein, zum Beispiel dann, wenn ein anderes Objekt an das aktuelle Objekt als Parameter der Objektmethode übergeben wird. Sie können aber auch langfristig sein, wenn das Objekt Referenzen auf die assoziierten Objekte speichert (Registrierung).

2.25.4 Benutzung

Das aktuelle Objekt benutzt eines oder mehrere andere Objekte um die anstehende Aufgabe erfüllen zu können. Das Objekt Backofen benutzt zum Beispiel das Objekt Thermostat, um die Aufgabe *backen* zu erfüllen, damit der Inhalt des Backofens nicht verkohlt.

2.25.5 Allgemein

Es muß angemerkt werden, daß man häufig aus dem Programmcode keine direkte Entscheidung treffen kann, ob Aggregation, Assoziation oder Benutzung vorliegt. Diese drei Beziehungen sind prinzipiell Designprinzipien, die in der späteren Implementierungsphase, zum Beispiel in Java, in relativ ähnliche oder sogar identische Konstrukte umgesetzt werden. Es ist deshalb wichtig, daß man das jeweils zugrunde gelegte Prinzip dokumentiert, damit die Funktionsweise besser und einfacher nachvollzogen werden kann.

2.26 Unterschiede zwischen abstrakten Klassen und Interfaces

Java fügt dem schon von C++ her bekannten Konzept der abstrakten Klasse noch ein weiteres, aus der Sprache Objective-C entliehenes Feature hinzu: Interfaces. Ein Interface ist zuersteinmal nichts anderes als eine Auflistung von Methoden. Eine Klasse implementiert ein Interface, wenn sie alle in der Interface-Deklaration angegebenen Methoden besitzt. Wird mindestens eine der Methoden des Interfaces nicht implementiert, so wird die Klasse zu einer abstrakten Klasse.

Mit dem Interface-Prinzip lassen sich in Java einfach *Benutzt*-Beziehungen modellieren. Eine Klasse benutzt eine andere Klasse in dem Sinn, daß es ganz spezielle Methoden dieser Klasse verwendet, um seine eigene Funktionalität zu realisieren - auf Daten der Hilfsklasse wird ja wegen dem Prinzip der Datenkapselung nicht direkt zugegriffen. Um ein Objekt zu benutzen, ist es nur wichtig, daß dieses Objekt die gewünschten Funktionen auch besitzt. Man definiert sich daher ein Interface, welches die benötigten Funktionen auflistet. Alle Objekte, die diese Funktionen benötigen, sprich, die dieses Interface verlangen, können nun all die Klassen verwenden, die dieses Interface implementieren.

Im Sinne eines guten Klassendesigns ist es daher wichtig, solche *Benutzt*-Beziehungen zu lokalisieren, um die Klassen flexibler zu machen und unnötigerweise angewendete Vererbung zu eliminieren.

Abstrakte Klassen sind im Gegensatz dazu Modellierungen des *Ist ein*-Prinzips und unterscheiden sich in dieser Hinsicht von Interfaces.

2.27 Wie kann man in Java Mehrfachvererbung simulieren?

Java kennt im Gegensatz zu C++ nicht das Feature der Mehrfachvererbung, eine Klasse kann nur genau einen Vorfahren haben, im Gegensatz zu beliebig vielen bei Mehrfachvererbung.

Häufig liest man, daß man die Mehrfachvererbung mit Hilfe von Interfaces nachbilden kann, indem man die Methodenschnittstelle der einzelnen Klassen als Interfaces definiert und diese Interfaces alle gleichzeitig von der fraglichen Klasse implementieren läßt, denn dies ist in Java möglich.

Mit Mehrfachvererbung hat dies aber nichts zu tun, betrachtet man den eigentlichen Sinn und Zweck von Vererbung. Vererbung ist eines der möglichen Prinzipien, um die Codewiederverwertung zu garantieren. Code, der in mehreren Klassen benötigt wird, ist nur in der Basisklasse aufgeführt und durch die Vererbung können die "Erben" auch diesen Code benutzen. Der Code steht also nur an einer Stelle. Bei der oben beschriebenen "Nachbildung" wird in keiner Weise Code gespart, denn die Methoden der Interfaces müssen ja von der Klasse noch implementiert werden.

In diesem Sinne kann man Mehrfachvererbung nicht in Java nachbilden.

2.28 Wie kann man eine variable Parameterliste realisieren?

Es gibt Anwendungsfälle, in denen es äußerst nützlich wäre, einer Methode eine variable Anzahl an Parametern übergeben zu können. Ein Anwendungsfall wäre eine Klasse beliebig-dimensionaler Vektoren. Ein Konstruktor dieser Klasse müßte die Initialisierung eines Vektors mit bestimmten Werten erlauben und da beliebig-dimensionale Vektoren von dieser Klasse verarbeitet werden, müßte der Konstruktor mit einer variablen Parameteranzahl arbeiten können. Mit einem Trick kann man eine variable Parameterliste realisieren: Man übergibt der Methode ein Array von Referenzen auf die Klasse *Object*. Da in Java alle Objekte von der Klasse *Object* abstammen, sie ist quasi die Klasse aller Klassen, kann das Array Referenzen auf beliebige Objekte aufnehmen. Innerhalb der Methode kann dann mit dem *length*-Feld des Arrays die Anzahl der übergebenen Parameter ermittelt werden. Mit dem *instanceof*-Operator von Java kann dann der Typ des Objekts ermittelt werden, auf den die Referenzen verweisen und an Hand dieser Informationen kann man dann festlegen, was getan werden soll. Ein Programmfragment verdeutlicht das bisher Gesagte:

```
class Test
```

```
{
  void VariableParameterliste(Object Liste[])
  {
    //Liste.length -> Anzahl der Argumente
    //Typ von Liste[x] mit instanceof
    ...
  }
}
```

Ein Nachteil ist offensichtlich: Es kann nur genau eine Methode mit variabler Parameterliste geben bzw. innerhalb dieser einen Methode müssen alle Variationen berücksichtigt und implementiert werden, was nicht unbedingt zur Übersichtlichkeit des Programms beiträgt.

Ein Aufruf der Methode gestaltet sich nun wie folgt:

```
...

VariableParameterliste(new Object[]{ParameterObjekt1, ..., ParameterObjektn})

...
```

Will man elementare Datentypen wie double oder int an die Methode übergeben, so muß man die zugehörigen Wrapper-Klassen verwenden, die diese elementaren Datentypen in Klassen "einpacken".

2.29 Wie kann man erreichen, daß Methoden nach den dynamischen Typen der Parameter ausgewählt werden?

Java wählt Methoden nach dem statischen Typ der Übergabeobjekte aus. Man kann jedoch eine dynamische Auswahl simulieren, indem man das oben angesprochene Prinzip der variablen Parameter hierauf überträgt. Man prüft die übergebenen Referenzen mit dem *instanceof*-Operator, der den dynamischen Typ des Parameters liefert. Anhand dieser Informationen kann man dann ein "dynamisches" Verhalten der Methodenaufrufe realisieren. Auch hier gilt, wie auch schon bei der variablen Parameterliste, daß eine einzige Methode alle Aufrufmöglichkeiten abdecken muß.

2.30 Sind Methoden in Java immer virtuell?

In Java sind alle Methoden virtuell, eine Unterscheidung zwischen virtuellen und nicht-virtuellen Methoden wie zum Beispiel in C++, gibt es in Java nicht. Ist es für die Funktionsweise eines Objektes wichtig, daß die vorhandenen Methoden nicht überschrieben werden können - etwa um zu verhindern, daß sich die Funktionalität des Objekts dadurch grundlegend ändern läßt - so muß man entsprechende Methoden mit dem Modifizierer *final* deklarieren, der ein Überschreiben verhindert.

2.31 Wie kann man in Java Referenzen auf konstante Objekte erzeugen?

C++ kennt das Schlüsselwort *const*, das es erlaubt, konstante Objekte zu definieren, deren Wert man nicht ändern kann. Auf ein solches Objekt kann man nur Methoden anwenden, die ebenfalls als *const* definiert sind, die also die Daten des Objektes nicht ändern können.

Java kennt keinen Modifizierer *const*, es ist aber recht einfach möglich, diesen nachzubilden. Alles, was man dazu braucht, ist ein Interface, das die konstanten Methoden enthält, die das Objekt nicht ändern können. Will man nun ein konstantes Objekt zurückgeben oder erzeugen, dann gibt man einfach eine Referenz vom Typ des Interfaces zurück und schon hat man das Gewünschte erzielt.

Folgendes Beispiel soll das Ganze etwas verdeutlichen:

```
class NichtKonstanteKlasse implements KonstanteKlasse
{
  public SetValue(...)
  {
    ...
  }

  public GetValue()
  {
    ...
  }
}

interface KonstanteKlasse
{
  public GetValue();
}
```

Anmerkung: Im Prinzip wird hier durch das Interface eine Art Untermenge der Klasse “NichtKonstanteKlasse” definiert. Bei C++ läuft dieser Prozeß im Prinzip auch so ab, nur wird hier vom Compiler automatisch diese Untermenge an Hand des Modifizierers *const* erzeugt und der Programmierer muß sich darum nicht kümmern. In Java muß man diesen Prozeß selbst von Hand durchführen.

2.32 Wie kann ich Referenzen von Übergabeparametern ändern?

Von Sprachen wie C++ oder Pascal kennt man die Möglichkeit, Referenzen, die an die Methode übergeben werden, innerhalb der Methode zu ändern, an die sie übergeben wurden, wodurch nach dem Aufruf der Methode die Referenzen auf andere Objekte verweisen.

In Java gibt es zwei Möglichkeiten, diesen Effekt zu erreichen:

- Einführen einer weiteren Ebene mit Wrapper-Klassen, die spezielle Referenzen zur Verfügung stellen
- Verwendung von eindimensionalen, einelementigen Arrays

Der erste Punkt dürfte klar sein, jedoch steht der Aufwand erst dann in einem vernünftigen Ergebnis, wenn diese Art der Parametermodifikation öfter innerhalb des Programms vorkommt, denn ansonsten lohnt sich das Design einer speziellen Klasse nicht unbedingt. Den zweiten Weg verdeutlicht das folgende Programmfragment:

```
public void ParameterModifikation(Object Parameter[])
{
  ...

  Parameter[0]="Neue Referenz";

  ...
}

Object Parameter[]=new Object[1];

Parameter[0]="Zu modifizierender Parameter";

ParameterModifikation(Parameter);

//Nach Ablauf steht in Parameter[0] eine neue Referenz
```

2.33 Wie erzeuge ich eine "tiefe" Kopie eines Objektes?

Weißt man einer Objektvariablen eine andere zu, so wird nur die Referenz kopiert und beide Objektvariablen können das Objekt modifizieren und beide Objektvariablen sehen auch diese Modifikation. Dies ist das Standardverhalten von Java und wird *flache* Kopie (*shallow* copy) genannt. Im Normalfall ist es eigentlich unerwünscht, daß nur eine flache Kopie erzeugt wird, denn dadurch können die beiden Objektvariablen nicht unabhängig voneinander mit dem Objekt arbeiten.

Ein sehr eleganter Weg, eine tiefe Kopie eines Objektes zu erzeugen, verwendet den Serialisierungs-Mechanismus. Objekte, die man mit diesem Verfahren kopieren möchte, müssen also das Interface *Serializable* implementieren.

```
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;

public class TiefeKopie
{

public static Object Kopiere(Object DasObjekt) throws Exception
  {

ByteArrayOutputStream ByteOutput = new ByteArrayOutputStream();

ObjectOutputStream ObjectOutput = new ObjectOutputStream(ByteOutput);

    ObjectOutput.writeObject(DasObjekt);


ByteArrayInputStream ByteInput = new ByteArrayInputStream(ByteOutput.toByteArray());

ObjectInputStream ObjectInput = new ObjectInputStream(ByteInput);

    return(ObjectInput.readObject());
  }
}
```

Zuerst werden zwei Ausgabeströme angelegt: Ein *Byte*-Strom und ein Object-Strom. Der *Byte*-Strom wird hinter den *Object*-Strom geschaltet und das zu kopierende Objekt wird auf dem *Object*-Strom ausgegeben und durch die Verknüpfung der beiden Ströme schließlich in den *Byte*-Strom geschrieben. Dieser *Byte*-Strom wird dann mit einem zweiten Strom wieder eingelesen und mit einem weiteren *Object*-Strom wird aus den einzelnen Bytes wieder ein Objekt rekonstruiert. Die Ausgabe des *Object*-Stroms ist dann das kopierte Objekt.

Die Anwendung der Klasse zeigt folgendes Codefragment:

```
class Test
```

```
{
  public static void main(String Param[])
  {
    int EinArray[]={56,42,67,90,12,45};

    int DieKopie[]=(int []) TiefeKopie.Kopie(EinArray);
  }
}
```

Ein Array ist in Java nichts weiteres als ein Objekt und implementiert auch das *Serializeable*-Interface, weshalb das obige Kopierverfahren problemlos greifen kann.

2.34 Wie kann man den Typ *enum* aus C++ in Java umsetzen?

C++ bietet einen sogenannten Enumerationstyp. Eine Variable eines solchen Typs kann nur definierte Werte annehmen, die mit symbolischen Namen bezeichnet werden. Folgende Enumeration könnte zum Beispiel für eine Ampel verwendet werden:

```
enum Ampel {rot, gruen, gelb};
```

2.34.1 Straightforward-Implementierung

Java bietet diesen Typ nicht, er kann aber recht einfach mit einem Interface nachgebildet werden:

```
interface Ampel
{
  public static final int Rot=1;
  public static final int Gruen=2;
  public static final int Gelb=3;
}
```

Ein C++ Compiler führt automatisch die Zuordnung von Variablennamen zu eindeutigen Zahlenwerten durch, in Java muß der Programmierer diesen Prozeß erledigen. Die Verwendung differiert zwischen Java und C++:

```
Ampel EineVariable=rot; // Das ist C++

int EineVariable=Ampel.Rot; // Das ist Java
```

Man erkennt einen großen Nachteil der Java-Version: Da es sich um eine int-Variable handelt, ist es prinzipiell möglich, jeden Wert an diese Variable zuzuweisen, eben nicht nur Werte aus dem Wertebereich von Rot, Gruen und Gelb. Im Gegensatz dazu läßt ein C++-Compiler nur Zuweisungen von Symbolen aus dem Enumerationstyp zu. Insofern ist die Nachbildung in Java weniger sicher als das C++-Pendant und sollte daher mit Vorsicht angewendet werden.

2.34.2 Die typsichere Variante

Wie schon oben angesprochen bietet diese direkte Implementation den Nachteil, daß sie nicht typsicher ist. Doch es ist recht einfach möglich, mit Java eine typsichere Implementierung zu generieren:

```
public final class Ampel
{
  private Ampel() {}

  public static final Ampel Rot = new Ampel();
  public static final Ampel Gruen = new Ampel();
  public static final Ampel Gelb = new Ampel();
}
```

Auf den ersten Blick sieht die Klassendefinition ziemlich kompliziert und unverständlich aus, das Prinzip ist jedoch nicht schwer zu verstehen:

- Zuersteinmal ist die Klasse als *final* deklariert, wodurch verhindert wird, daß von dieser Klasse Ableitungen gebildet werden können. Der Aufzählungstyp kann also - weder absichtlich noch unabsichtlich - durch Vererbung verändert werden
- Der Konstruktor der Klasse ist *private*, dadurch kann nur die Klasse selbst diesen aufrufen. Damit sind die einzigen Instanzen, die von Ampel erzeugt werden können, die, die den *public*-Variablen der Klassen zugewiesen sind.
- Einer Referenz vom Typ Ampel können durch diese Maßnahmen nur die Werte {Rot, Gruen, Gelb} zugewiesen werden, andere Instanzen vom Typ Ampel sind ausgeschlossen

Will man nun diesen Typ von enumeration benutzen, so sieht das in etwa wie folgt aus:

```
Ampel EineVariable=Ampel.Rot; // problemlos möglich
Ampel EineVariable=new Ampel(); // nicht möglich, da Konstruktor private
Ampel EineVariable=Ampel.Pink; // Ist nicht Element der Aufzählung
```

Dieser Typ der Enumeration verwendet also Referenzen und nicht, wie Version 1, int-Werte, die nicht typsicher sind. Durch die Typprüfungsmechanismen von Java wird diese Art von Enumeration vollkommen typsicher und steht dem *enum*{...} Konstrukt von C++ nun in nichts mehr nach.

2.35 Verwirrendes *import*-Statement

In einem Programm kommen folgende beide *import*-Statements vor:

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

Warum reicht nicht das erste Statement aus, um die nötigen Packages zu importieren?

Vom intuitiven her würde man sagen, daß mit Hilfe des "*" alle Packages importiert werden, die unterhalb des Pfades *java.awt* stehen, doch leider ist dies nicht so. Der "*" kann nur den eigentlichen Package-Namen ersetzen, steht aber nicht für die Pfade zum Package.

Mit *import java.awt.*;* werden daher nur alle Packages importiert, die im Verzeichnis *java.awt* stehen, nicht jedoch Unterpackages, die noch tiefer verschachtelt sind. Deshalb ist das zweite *import*-Statement nötig, welches alle Packages in *java.awt.event* importiert.

Das ganze ist zwar nicht sonderlich intuitiv gelöst, da man das Verhalten von "*" normalerweise in anderer Art und Weise gewohnt ist, aber naja...

2.36 Wie kann ich aus Dateien zeilenweise einlesen?

Generell nutzt man für die Datei-Operationen einen Buffered Reader oder eine Klasse, die von dieser abstammt. Diese Klasse kennt dann die Methode *readLine()*, die eine Zeile aus der Datei ausliest und diese Zeile als einen String zurückgibt.

Zu beachten ist, daß bei dem Erreichen des Dateiendes kein leerer String, sondern eine Nullreferenz zurückgegeben wird. Ein Programm, daß aus einer Datei zeilenweise liest, sieht in Java wie folgt aus:

```
import java.io.*;

class ZeilenweiseLesen
```

```
{

public static void main(String[] args) throws IOException
  {
    String Zeile;


File EingabeDatei = new File("Eingabe.txt"); //hier steht die Datei, aus der gelesen wi

FileReader EingabeStrom = new FileReader(EingabeDatei);

BufferedReader Eingabe = new BufferedReader(EingabeStrom);

    while ((Zeile = Eingabe.readLine()) != null)
    {
      System.out.println(Zeile);
    }
  }
}
```

Kapitel 3

Programmoptimierungen

3.1 Allgemeine Bemerkungen

Ziel einer Optimierung ist immer die Erreichung eines der beiden Ziele:

- Reduzierung der Programmgröße
- Steigerung der Geschwindigkeit

Dabei soll im folgenden insbesondere auf den Geschwindigkeitsaspekt eingegangen werden. Heutige Programme weisen eine hohe Lokalität auf, d.h. bei der Ausführung eines Programmes bewegt sich der Prozessor immer in einem relativ kleinen Teil des Gesamtprogramms. Eine Folge daraus ist die 90:10 Regel, die folgendes besagt:

In heutigen Programmen benötigt 10% des Programmcodes 90% der Ausführunsgzeit

Natürlich ist diese Regel nur rein statistisch, trifft in der Praxis in vielen Fällen jedoch zu. Man kann aber auch andere Verhältnisse angeben, ausgedrückt soll nur werden, daß sich die Optimierung auf bestimmte Programmabschnitte beschränken kann. Ziel einer Optimierung ist es daher, diese 10% des Programmcodes zu finden und diesen zu optimieren, denn hier wirken sich Optimierungsversuche am stärksten aus.

- Optimierung sollte nicht um jeden Preis erfolgen, denn auf der Negativseite der Optimierung sind folgende Punkte zu verbuchen:
- Optimierung kann zur Quelle neuer, unter Umständen schwerer zu findender Fehler werden
- Optimierung kann den Code schwerer verständlich machen und damit auch die Pflege des Codes erschweren

- Optimierung kann die Erweiterbarkeit des Codes erschweren oder unter Umständen sogar verhindern
- Eine Optimierung kann auf der einen Plattform die Geschwindigkeit wesentlich steigern, auf einer anderen jedoch das genaue Gegenteil erreichen

Aus diesen Gründen ist Optimierung immer mit Vorsicht und Umsicht einzusetzen. Vor allem gilt, daß man mit Optimierung einen schlecht gewählten Algorithmus nur marginal verbessern kann, die Suche nach besseren Algorithmen ist hier die bessere Wahl. Statt zu versuchen, einen BubbleSort-Algorithmus zu optimieren sollte man zum Beispiel den Quicksort-Algorithmus verwenden, der um Größenordnungen besser ist. Optimierung sollte deshalb nur dann zum Tragen kommen, wenn auf anderem Weg, sprich mit Hilfe anderer Algorithmen, keine Verbesserung mehr zu erwarten ist.

3.2 Division und Multiplikation

Wird in einem Programmfragment durch eine Zweierpotenz geteilt oder mit einer Zweierpotenz multipliziert, so ist es besser, diese Operation mit Hilfe von Schiebebefehlen zu realisieren, denn diese sind einfach in Hardware und damit schnell durchführbar. Dabei schiebt man die zu teilende Zahl nach links bzw. nach rechts und zwar soweit, wie es im Exponenten der Zweierpotenz, die den Teiler der Zahl darstellt, vermerkt ist.

Beispiel:

- Aus x/16 wird x>>4 wegen $16=2^4$ (Divison bedeutet rechtsschieben)
- Aus x*4 wird x<<2 wegen $4=2^2$ (Multiplikation bedeutet linkschieben)

Allgemein gesagt:

- $x/2^p=x>>p$
- $x*2^p=x<<p$

3.2.1 Anmerkung - Mögliche Fehlerquelle

Im Prinzip sind die beiden Operatoren “/” und “>>” identisch, bei negativen Zahlen muß man jedoch ein unterschiedliches Rundungsverhalten beachten. Damit ist die Ersetzung der Division durch Rechtsschieben nur dann unproblematisch, wenn klar ist, daß bei der Division keine negativen Zahlen vorkommen.

Es gilt:

- Der Operator "/" rundet in Richtung 0
- Der Operator ">>" rundet in Richtung $-\infty$

Das unterschiedliche Verhalten wird dadurch erst bei negativen Zahlen sichtbar:

- -7/2 ist -3
- -7>>1 ist -4

3.3 Zugriff auf mehrdimensionale Arrays beschleunigen

Der Speicher eines Computers ist immer eindimensional, die einzelnen Speicherzellen sind wie an einer Schnur hintereinander aneinandergereiht und durchnummeriert. Der Zugriff auf die gewünschte Speicherzelle erfolgt durch die Angabe der Nummer, die die Speicherzelle in diesem Modell hat. Mehrdimensionale Arrays müssen auf diese eindimensionale Darstellung umgerechnet werden. Bei der Verwendung von mehrdimensionalen Arrays erledigt diese Abbildung normalerweise der Compiler und der Programmierer muß sich um diesen Aspekt nicht kümmern. Um jeden Fall verarbeiten zu können, muß sich der Compiler eines allgemeingültigen Verfahrens bedienen und ist deshalb nicht immer effizient.

Im Fall, daß die Dimensionen eines Arrays Zweierpotenzen sind, kann man eine eigene Umrechnung auf den eindimensionalen Speicher verwenden, die schneller als die vom Compiler verwendete Variante ist. Zum Verständnis betrachten wir den Fall eines zweidimensionalen Arrays, mit den Dimensionen d_1 und d_2. Dabei sei n_{d_1} der Exponent der Dimension d_1, und Analoges gilt für n_{d_2}. Dann kann man den Zugriff A[x][y] auf das Array A wie folgt anders schreiben:

A[x][y] ist äquivalent zu A'[y<<n_{d_1} | x]

Dabei wird n_{d_1} als die Obergrenze der ersten Dimension (Variable x) interpretiert und A' ist das zu A korrespondierende

Array der Größe $n_{d_1} \cdot n_{d_2}$.

3.3.1 Standardmethode des Compilers

Unter normalen Umständen macht ein Compiler nicht die Annahme, daß die Dimensionen eines Arrays Zweierpotenzen sind. Ein Compiler verwendet daher unter normalen Umständen die Multiplikation. Für ihn sieht die Umsetzung auf den eindimensionalen Speicher wie folgt aus:

A[x][y] ist äquivalent zu A'[$y \cdot d_1$ + x]

Unter normalen Umständen ist die Multiplikation weniger effizient als die Anwendung des Shift-Operators, weshalb die Compiler-Variante in den Spezialfällen einer Zweierpotenz gegenüber der verbesserten Variante verliert.

3.4 Mehrfache Berechnungen vermeiden

Werden Teile von Formeln mehrfach berechnet, so ist es sinvoll, diese mehrfache Berechnung zu vermeiden und stattdessen diese Teile nur einmal zu berechnen und das Ergebnis in einer Variablen zu speichern und mit diesem Ergebnis zu arbeiten.

Beispiel:

```
a=23*(x/5*9)+13
b=67*(x/5*9)+19-x
```

wird zu

```
c=x/5*9
a=23*c+13
b=67*c+19-x
```

3.5 Schleifeninvarianten

Berechnungen, die unabhängig von Schleifenparametern sind, d.h. während des Durchlaufs durch die Schleife immer das gleiche Ergebnis liefern, sollten außerhalb der Schleife berechnet werden, da ansonsten die Berechnung unnötig oft durchgeführt wird.

3.6 Ausnutzen des Kurzschlußverfahrens bei boolschen Ausdrücken

In allen Programmiersprachen und so natürlich auch in Java, gibt es Schleifen und bedingte Anweisungen. Die Ausführung solcher Konstrukte basiert auf boolschen Ausdrücken, die die beiden Zustände *wahr* und *falsch* annehmen können. Insbesondere sind Ausdrücke häufig, die einzelne Abfragen mit AND (in Java: &&) und OR (in Java: ||) zu größeren Abfragen zusammenfassen.

Betrachtet man nun die Tabellen der AND und der OR-Funktion:

AND	true	false
true	true	false
false	false	false

OR	true	false
true	true	true
false	true	false

Dann erkennt man daraus folgendes:

- Ist bei der AND-Verknüpfung einer der beiden Operanden *false*, dann ist das Ergebnis auf jeden Fall *false*
- Ist bei der OR-Verknüpfung einer der beiden Operanden *true*, dann ist das Ergebnis in jedem Fall *true*

Ist also im Programmfragment

```
if (ErsteBedingung && ZweiteBedingung)
```

die erste Bedingung schon *false*, so muß die zweite Bedingung gar nicht mehr ausgewertet werden, denn das Ergebnis ist ja schon definitiv *false*. Analog ist es bei der OR-Verknüpfung:

```
if (ErsteBedingung || ZweiteBedingung}
```

Ist das Ergebnis der ersten Bedingung schon true, dann braucht die zweite Bedingung gar nicht erst ausgewertet zu werden, denn das Endergebnis steht schon fest und ist definitiv *true*. Java, und auch viele andere Programmiersprachen, berücksichtigen diese beiden Eigenschaften und beenden die Auswertung der boolschen Ausdrücke, sobald das Ergebnis feststeht. Dies bezeichnet man als Short-Cut. Die Auswertung boolscher Ausdrücke beginnt in Java von links aus. Damit kann man folgende zwei Prinzipien für die Optimierung solcher boolschen Ausdrücke herleiten:

- Sind verschiedene Bedingungen durch AND verknüpft, so ordnet man die einzelnen Ausdrücke in der Reihenfolge an, daß Ausdrücke, deren Wahrscheinlichkeit den Wert *false* anzunehmen, hoch ist, weit links stehen.
- Sind verschiedene Bedingungen durch OR verknüpft, so ordnet man die einzelnen Ausdrücke in der Reihenfolge an, daß Ausdrücke, deren Wahrscheinlichkeit den Wert *true* anzunehmen, hoch ist, weit links stehen.

Durch diese zwei Regeln wird sichergestellt, daß nur die minimale Anzahl an Bedingungen ausgewertet wird, um zu einem Ergebnis des zusammengesetzten boolschen Ausdrucks zu kommen.

3.6.1 Ausschalten des Short-Cuts - Erzwungene Auswertung aller Ausdrücke

Unter Umständen kann es sein, daß der Compiler mit der Möglichkeit des Short-Cut-Prinzip zu viel des Guten tut. Dies ist etwa dann der Fall, wenn ein

Ausdruck sowohl einen boolschen Wert zurückliefert als auch als Seiteneffekt irgendwelche weiteren Operationen durchführt. Zwar ist diese Art der Programmierung alles andere als ein guter Programmierstil, weil aus dem Quelltext nicht mehr direkt der Effekt des Code-Fragments sichtbar wird, aber um solche Programme zu einem korrekten Ablauf zu bringen, darf der Short-Cut nicht durchgeführt werden.

Um dies zu erreichen, verwendet man für die UND-Verknüpfung nicht mehr den Operator "&&", sondern den Operator "&" und analog für die ODER-Verknüpfung nicht mehr den Operator "||", sondern den Operator "|".

3.7 Berücksichtigung des Caches

Programiersprachen haben den Zweck, daß der Programmierer von der Hardware abstrahieren kann, sich also keine Gedanken über den Aufbau der zu programmierenden Maschine machen muß. Durch die Programnmübersetzung wird dann Code für die jeweilige Plattform erzeugt. Java geht in diesem Fall sogar noch ein wenig weiter: Der erzeugte Bytecode, der auf Grund des Java-Programms erzeugt wird, ist auch noch maschinenunabhängig und erst die virtuelle Maschine sorgt für die Ausführung auf der spezifischen Plattform. Doch auch bei Java wird irgendwann die konkrete Rechnerarchitektur in den Ausführungsprozeß des Programmes involviert und damit hängt die Performance eines Programmes auch von der Hardware ab.

Heutige Prozessoren verwenden einen sogenannten Cache als Zwischenspeicher für häufig benötigte Daten. Der Zugriff auf Daten im Cache ist um Größenordnungen schneller als der Zugriff auf Daten im Hauptspeicher. Optimal ist es also, wenn Daten oder Programmteile, die oft benötigt werden, nach Möglichkeit fast permanent im Cache zur Verfügung stehen um eine möglichst schnelle Programmausführung zu garantieren. Ein Cache hat aber nur endliche Kapazität, ist der Cache voll, so werden andere Daten aus dem Cache geworfen, bevor neue Daten eingelagert werden. Und genau hier kann eine Optimierung greifen: Unter Umständen sind Zugriffe in einem Programm so ungünstige gewählt, daß Daten, auf die eigentlich dauernd zugegriffen wird, permanent aus dem Cache geschmissen werden.

Hierzu ein Beispiel:

```
class Test
```

```
{
    public static void main(String Param[])
    {
      double EinArray[][]=new double[10000][10000]

      int s,t;

      for(s=0;s<10000;s++)
        for(t=0;t<10000;t++)
          System.out.println(EinArray[s][t]);
    }
}
```

Das Programmfragment tut nichts anderes als ein ziemlich großes Array von *double*-Werten anzulegen und die Werte des Arrays Zeile für Zeile auszugeben. Nun kommt es darauf an, wie die verwendete Java-Version ein zweidimensionales Array in den eindimensionalen Computerspeicher abbildet: Wird das Array Zeile für Zeile nacheinander im Speicher abgelegt, dann kann der Caching-Mechanismus bei obigem Programm effektiv greifen. Es werden immer ein paar Daten im voraus aus dem Speicher gelesen und unser Programm greift auch in dieser Reihenfolge auf den Speicher zu - beinahe 100% ige Effektivität.

Anders sieht es dagegen aus, wenn in der Java-Implementierung ein solches Array spaltenweise in den Computerspeicher abgebildet wird. Es erfolgt ein Zugriff auf ein Element im Speicher und der Cache holt vorsorglich weitere Daten, die vor und nach der gelesenen Adresse liegen, in den Cache-Speicher um bei einem Zugriff die Daten schneller zur Verfügung zu haben. Wir verwenden immer noch obiges Programm für den Zugriff, aber wenn wir auf das Element rechts neben dem aktuellen Element zugreifen, dann liegt dieses physikalisch nicht in direkter Nachbarschaft zum aktuellen Element, sondern ziemlich weit weg. Das Resultat ist, daß der Cache mit falschen Daten geladen wurde und die Daten zeitaufwendig aus dem Hauptspeicher geholt werden müssen. Praktisch bei jedem Durchlauf der beiden Schleifen müssen die Daten aus dem langsamen Hauptspeicher geholt werden und das Programm läuft merklich langsamer ab. Es wäre günstiger, den Code wie folgt abzuändern:

```
class Test
{
    public static void main(String Param[])
    {
      double EinArray[][]=new double[10000][10000]

      int s,t;

      for(s=0;s<10000;s++)
        for(t=0;t<10000;t++)
```

```
System.out.println(EinArray[t][s]);  //s und t vertauscht!
        }
    }
```

Nun orientiert sich der Code an der verwendeten Speicherabbildung und läuft merklich schneller ab. Gerade bei einer Ausgabe macht die Reihenfolge natürlich einen Sinn, weshalb in der Praxis obige Umstellung nicht möglich wäre. Matrizenrechnungen, zum Beispiel die Addition von zwei Matrizen, lassen sich aber in dieser Hinsicht optimieren und es können merkliche Geschwindigkeitsverbesserungen erzielt werden.

3.8 Verwenden von "abgekürzten" Operatoren

Java kennt, genauso wie C++, sogenannte abgekürzte Operatoren. *a=a+5* kann kürzer als *a+=5* geschrieben werden. Sicher kann man hier das Argument bringen, daß die Variante mit den abgekürzten Operatoren die Lesbarkeit des Quelltextes herabsetzt, ein Gegenargument ist aber schnell gefunden: Manche VMs erzeugen aus solchem Java-Quelltext Code, der schneller als der Code der Variante eins abläuft. Dies liegt daran, daß praktisch alle Prozessoren spezielle Befehle für das Inkrementieren bzw. Dekrementieren um eins (*i++* bzw. *i–* statt *i=i+1* bzw. *i=i-1*) besitzen und diese Befehle damit schneller ausgeführt werden. Manche Prozessoren besitzen sogar Befehle, die *i+=5* oder Ähnliches direkt auf die Maschinencode-Ebene abbilden können. Insofern ist es ratsam, die abgekürzten Befehle zu verwenden, langsamer können die erzeugten Programme nicht werden, allerhöchstens schneller und das ist in den meisten Fällen wohl ein gewünschter Effekt.

3.9 Einmaliges Berechnen von Ausdrücken innerhalb von Schleifen

Innerhalb größerer Routinen kann der Fall auftreten, daß komplexe Ausdrücke innerhalb einer Schleife vorkommen. Obwohl diese Ausdrücke konstant sind, werden sie in der Schleife mehrfach berechnet, was natürlich Ausführungszeit kostet. Auf der anderen Seite möchte man aber diese komplexen Ausdrücke nicht außerhalb der Schleife in einer temporären Variablen Zwischenspeichern, weil dadurch die Übersichtlichkeit des Quellcodes verloren geht. Man kann mit Hilfe einer Interface-Deklaration erreichen, daß der komplexe Ausdruck nur einmal innerhalb der Schleife berechnet wird.

Ein Beispiel verdeutlicht das:

```
class Test
```

```
{
  public static void main(String Param[])
  {
    int i;

    for(i=0;i<10000;i++)
    {

interface KomplexerAusdruck {double Wert=sin(67)+sqrt(23);}

      System.out.println(KomplexerAusdruck.Wert);
    }
  }
}
```

Die Interface-Deklaration wird nur genau einmal ausgewertet, denn Variablen innerhalb von Interfaces haben implizit den Modifizierer *static final*, werden also nur genau einmal gesetzt. Damit wird der Inhalt der Variablen Wert nur einmal berechnet, obwohl die Berechnung innerhalb einer Schleife steht. Unnötige Berechnungen werden vermieden und gleichzeitig die Lesbarkeit erhalten, da die Berechnung nicht aus dem Kontext gerissen wird.

3.10 Schleifen nach unten zählen lassen

Überall wo es möglich ist, sollten Zählschleifen nicht aufwärts, sondern abwärts laufen.

```
...
for(i=1;i<=10000;i++)
...
```

Wird langsamer ausgeführt als

```
...
for(i=10000;i>0;i--)
...
```

Der Grund hierfür ist relativ einfach: Im ersten Fall muß i immer mit dem Wert 10000 verglichen werden, was Zeit kostet. Im zweiten Fall erfolgt nur ein Vergleich auf Null und aufgrund des Designs von Prozessoren existiert im Maschinencode immer ein Befehl, der auf den Zustand Null überprüft. Dieser Befehl ist wesentlich schneller als der Vergleichsbefehl und deshalb wird Variante zwei wesentlich schneller ausgeführt.

3.10.1 Allgemein

Generell ist ein Vergleich auf Null einem allgemeinen Vergleich vorzuziehen, da hierfür ein eigener Befehl existiert und vor allem nicht mit einer Variablen, sondern mit einer fetsen Konstante (in diesem Fall die Zahl 0), verglichen werden kann.

3.11 Variablentypen

Java unterscheidet intern zwei Klassen von Variablentypen - die 32-bittigen und die anderen. Intern arbeitet die Virtual Machine mit einer Wortgröße von 32 Bit, sowohl Speicher für die Zugriffe auf lokale Variablen sind 32 bittig als auch Stackelemente.

Direkte Operationen stehen für die Datentypen int, long, float, double zu Verfügung, d.h. alle anderen Datentypen werden vor der Berechnung in den naheliegendsten dieser Typen umgewandelt, es sind Casts nötig und dies kostet unnötige Zeit.

Daraus folgt:

- Operationen auf Variablen vom Typ byte sind langsamer als Operationen auf int-Variablen

Daher sollte man, bei der Optimierung auf Geschwindigkeit, int Variablen den Vorzug gegenüber byte-Variablen geben.

3.11.1 Datentypen erster und zweiter Klasse

Wegen der 32-Bittigkeit existieren Datentypen erster und zweiter Klasse

- Erste Klasse: int, long, float, double
- Zweite Klasse: byte, short, char

Für Datentypen erster Klasse ist ein kompletter Satz an Funktionen vorhanden, Typen zweiter Klasse werden intern wie der Typ int behandelt, müssen also vor den eigentlichen Operationen umgewandelt werden. Daher sollte man Datentypen zweiter Klasse wenn möglich vermeiden.

Kapitel 4

Algorithmen

Dieser Abschnitt soll im Laufe der Zeit einige Algorithmen vorstellen, die unter Umständen in Programmen häufiger vorkommen können. Vor allem kann man aber aus den Programmen auch lernen, denn das Erlernen einer Programmiersprache geht am Besten durch die Beschäftigung mit vielen Beispielen.

4.1 Mathematische Ausdrücke auswerten

In manchen Situationen kann man in die Verlegenheit kommen, eine Formel, die in einem String gespeichert ist, ausrechnen zu müssen. Dies kann zum Beispiel dann der Fall sein, wenn man einen Funktionsplotter programmiert hat, oder eine Tabellenkalkulation oder oder....

In JavaScript gibt es einen einfachen Befehl, *eval(...)*, der den Ausdruck eines Strings berechnet. Java bietet eine solche Funktion leider nicht standardmäßig, also ist man gezwungen, sich eine solche selbst zu implementieren. Im Folgenden wird das Grundprinzip eines solchen Formelparsers besprochen und aufbauend darauf sollte es eigentlich nicht schwer sein, diesen beliebig auszubauen.

4.1.1 Die vier Grundrechenarten

Zuersteinmal gehen wir von einfachen Ausdrücken aus, d.h. Ausdrücke, in denen nur die vier Grundrechenarten +,-,*,/ und *keine* Klammern vorkommen. Außerdem gehen wir davon aus, daß die Ausdrücke syntaktisch korrekt sind, denn die Fehlerüberprüfung soll nicht Inhalt dieses Textes sein.

4.1.2 Die Operatoren + und -

Zu berechnen ist der Ausdruck a+b bzw. a-b, wobei a und b beliebig komplizierte Ausdrücke sein können. Dann kann man diesen Term offensichtlich so berechnen, indem man den Wert von a und den Wert von b bestimmt und die beiden erhaltenen Werte addiert oder voneinander subtrahiert.

4.1.3 Die Operatoren * und /

Auch bei der Multiplikation und der Division geht man so vor, d.h. es werden die beiden Teilausdrücke berechnet und dann entweder multipliziert oder dividiert.

4.1.4 Prioritäten

In einem Term haben die verschiedenen Operatoren verschiedene Prioritäten, d.h. die Reihenfolge der Auswertung ist festgelegt. Dabei gilt +,-,*,/, wobei die Priorität rechts am größten ist und nach links hin abnimmt.

Am Besten sieht man das an einem Beispiel, welches die Berechnungsreihenfolge anhand der Prioritäten demonstriert:

```
2+3*4-6/3
2+12-2
12
```

4.1.5 Die eigentliche Berechnung

Folgendes Verfahren kann man nun zur Berechnung des Termwertes anwenden, wobei von oben nach unten vorgegangen wird, d.h. die angegebenen Bedingungen werden in der aufgelisteten Reihenfolge geprüft:

- Suche nach dem Operator "+" - ist dieser vorhanden, so bestimme die Teilterme links und rechts des Operators und berechne den Wert dieser Terme
- Suche nach dem Operator "-" - ist dieser vorhanden, so bestimme die Teilterme links und rechts des Operators und berechne den Wert dieser Terme
- Suche nach dem Operator "*" - ist dieser vorhanden, so bestimme die Teilterme links und rechts des Operators und berechne den Wert dieser Terme

- Suche nach dem Operator "/" - ist dieser vorhanden, so bestimme die Teilterme links und rechts des Operators und berechne den Wert dieser Terme

Die Tatsache, daß man die Werte der Teilterme links und rechts des Operators berechnet legt eine Rekursion als Grundlage des Algorithmus nahe.

Für a+b sähe das Programmfragment so aus, wobei die Funktion Calculate(...) den Wert des übergebenen Strings ermittelt:

- Calculate(a)+Calculate(b)

4.1.6 Ein Beispiel

Wir betrachten folgendes Beispiel:

```
2+3/5*5/5
```

Außerdem beginnen wir mit der Suche von rechts. Dann erhalten wir Folgendes, wobei die Klammern die Reihenfolge der Auswertung angeben: $2+(3/5*5*5) = 2 + ((3/5) * (5/5)) = 2 + (0.6 * 1) = 2 + 0.6 = 2.6$

4.1.7 Klammern

Bis jetzt ist es nur möglich, Ausdrücke ohne Klammern auszuwerten. Die Erweiterung unseres bisherigen Algorithmus ist jedoch nicht schwer. Die Operatoren werden immer noch in der angegebenen Reihenfolge im String gesucht, allerdings werden die öffnenden und schließenden Klammern, die eventuell vorkommen können, mitgezählt. Ein Operator, im Sinne unserer obigen Auflistung, ist nur dann vorhanden, wenn man sich in der 0. Klammerebene befindet, d.h. alle Klammern sind geschlossen.

Diese Art von Suche erledigt folgender Algorithmus:

```
public static int LevelSearch(String Expression, char Symbol)
```

```
{
    int BracketLevel=0;
    int s;

    for(s=Expression.length()-1;s>=0;s--)
    {
      if (Expression.charAt(s)==')')
      {
        BracketLevel--;
      }
      else
      {
        if (Expression.charAt(s)=='(')
        {
          BracketLevel++;
        }
        else
        {
if (BracketLevel==0 && Expression.charAt(s)==Symbol) return(s);
        }
      }
   }
}
```

Die Berechnung der Terme erfolgt dann, wie schon oben angeführt mit Hilfe der Rekursion und man erhält abschließend folgendes Programm:

```
class MathematicalExpression
{
```

```
public static int LevelSearch(String Expression, char Symbol)
  {
    int BracketLevel=0;
    int s;

    for(s=Expression.length()-1;s>=0;s--)
    {
      if (Expression.charAt(s)==')')
      {
        BracketLevel--;
      }
      else
      {
        if (Expression.charAt(s)=='(')
        {
          BracketLevel++;
        }
        else
        {

if (BracketLevel==0 && Expression.charAt(s)==Symbol) return(s);
        }
      }
    }

    return(-1);
  }

  public static double Calculate(String Expression)
  {
    if (LevelSearch(Expression,'+')!=-1)
    {

String Left=Expression.substring(0,LevelSearch(Expression,'+'));

String Right=Expression.substring(LevelSearch(Expression,'+')+1,Expression.length());

      return(Calculate(Left)+Calculate(Right));
    }
    else
    {
      if (LevelSearch(Expression,'-')!=-1)
      {

String Left=Expression.substring(0,LevelSearch(Expression,'-'));
```

```
String Right=Expression.substring(LevelSearch(Expression,'-')+1,Expression.length());

        return(Calculate(Left)-Calculate(Right));
      }
      else
      {
        if (LevelSearch(Expression,'*')!=-1)
        {

String Left=Expression.substring(0,LevelSearch(Expression,'*'));

String Right=Expression.substring(LevelSearch(Expression,'*')+1,Expression.length());

          return(Calculate(Left)*Calculate(Right));
        }
        else
        {
          if (LevelSearch(Expression,'/')!=-1)
          {

String Left=Expression.substring(0,LevelSearch(Expression,'/'));

String Right=Expression.substring(LevelSearch(Expression,'/')+1,Expression.length());

            return(Calculate(Left)/Calculate(Right));
          }
          else
          {

if (Expression.charAt(0)=='(' && Expression.charAt(Expression.length()-1)==')')

return(Calculate(Expression.substring(1,Expression.length()-1)));
            else
              return(Double.parseDouble(Expression));
          }
        }
      }
    }
  }

  public static void main(String[] param)
  {
    String Test="3*(2+3)+56*3-(3/4)*3";

    System.out.println(Calculate(Test));
  }
```

}

4.2 Zahlensystemwandlungen

Neben dem von uns verwendeten dezimalen Zahlensystem gibt es noch viele andere. Die bekanntesten dürften wohl das Binär-, Oktal-, und Hexadezimalsystem sein. Im Folgenden soll eine Klasse entwickelt werden, die es erlaubt, Dezimalzahlen in andere Systeme umzurechnen und Zahlen aus diesen Systemen ins Dezimalsystem zu konvertieren.

Alle diese Zahlensysteme sind sogenannte Stellensysteme, d.h. jede Stelle hat eine bestimmte Wertigkeit. Am einfachsten kann man dies an dem uns gut bekannten Dezimalsystem verdeutlichen. Die erste Stelle, wobei von rechts begonnen wird, hat die Wertigkeit 1, die darauffolgende Stelle hat die Wertigkeit 10, die darauffolgende die Wertigkeit 100 usw. Man erkennt, daß die Wertigkeit eine Potenz der Basis des Zahlensystems ist, in unserem Fall hat das Dezimalsystem die Basis 10. Analog ist es bei den anderen Zahlensystemen, das Binärsystem hat zum Beispiel die Basis 2.

4.2.1 Fremdes Zahlensystem in Dezimalsystem

Die Umrechnung ist nach den oben gemachten Feststellungen recht einfach: Man nimmt den Wert der jeweiligen Stelle und multipliziert ihn mit der Wertigkeit dieser Stelle. Dies macht man für alle Stellen der Zahl und summiert dann die Einzelergebnisse zum Gesamtergebnis auf. Dieses Gesamtergebnis ist dann die Zahl im Dezimalsystem.

Beispiel:

Man betrachtet die Zahl 1001 im Binärsystem. Dann führt man folgende Rechnung durch $1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 9$. Also ist 1001 im Binärsystem die Zahl 9 im Dezimalsystem.

Allgemein:

Sei b die Basis des Zahlensystems und a_i der Wert der i-ten Stelle der Zahl, wobei bei der Stelle i=0 begonnen wird. Dann hat die Zahl den dezimalen Wert: $\sum_{i=0}^{n} a_i \cdot b^i$

4.2.2 Verbesserung der Berechnung

Bei der obigen Vorgehensweise muß man immer b^i berechnen und dieser Wert muß in einer separaten Variable mitgespeichert werden. Wieder an einem Beispiel kann man eine verbesserte Berechnungsweise erkennen:

1111 im Binärsystem soll ins Dezimalsystem umgewandelt werden. $1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = ((1 \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1$ ist eine andere Schreibweise und man erhält folgendes Verfahren:

```
Setze Ergebnis=0
Setze i=n

Solange i>0
  Ergebnis=Ergebnis*Basis
  Ergebnis=Ergebnis+a[i-1]
  i--

Gib Ergebnis aus
```

4.2.3 Vom Dezimalsystem in ein anderes System

Wir hatten schon festgestellt, daß für eine Zahl z gilt:

$Dezimal = z_n \cdot b^n + z_{n-1} \cdot b^{n-1} + ... + z_o \cdot b^0$

Dies kann man umformen:

$Dezimal = (...(z_n \cdot b + z_{n-1}) \cdot b + z_{n-2}) \cdot b + ...)...) \cdot b + z_0$

Man erkennt: z_0 ist der Rest der ganzzahligen Division der Dezimalzahl, bei Divison durch b. Nun betrachtet man das ganzzahlige ergebnis der Division als neue Zahl und dividiert wieder durch b. Man erhält z_1, dann z_2 usw. Mit fortlaufender Division kann man also die Dezimalzahl in eine Zahl zur Basis b umwandeln.

Allgemein erhält man also folgendes Verfahren:

```
Setze a=0
Setze i=0

Zahl=Dezimalzahl die umgewandelt werden soll

Solange (Zahl!=0)
  a[i]=Zahl % Basis
  Zahl=Zahl / Basis
  i++

In a steht die umgewandelte Zahl
```

4.2.4 Eine Java-Klasse

Zum Abschluß noch die Java-Klasse, die unsere beiden Algorithmen implementiert:

```
  class Zahlensystem
  {

public static String DezimalToX(long Dezimal, int Basis)
    {
      String Ergebnis=new String();

      while (Dezimal!=0)
      {
        int Stelle=Dezimal % Basis;


if (Stelle<10) Ergebnis=String.toString('0'+Stelle)+Ergebnis;

else Ergebnis=String.toString('A'+Stelle-10)+Ergebnis;

        Dezimal=Dezimal/Basis;
      }

      return(Ergebnis);
    }

    public static long XToDezimal(String X, int Basis)
    {
      int i;

      long Ergebnis=0;

      for(i=X.length();i>=0;i--)
      {
        Ergebnis=Ergebnis*Basis;
        Ergebnis=Ergebnis+Long.toLong(X.get(i));
      }

      return(Ergebnis);
    }
  }
```

4.3 Divide and Conquer Strategie

Wie die Überschrift schon sagt, handelt es sich bei Divide & Conquer (zu deutsch : teile und herrsche) um eine Vorgehensweise, um ein gegebenes Problem zu lösen. Dazu wird das Problem in mehrere Teilprobleme aufgeteilt, jedes der Teilprobleme wird gelöst und aus den Teillösungen wird dann wieder eine Gesamtlösung für das ursprüngliche Problem zusammengesetzt.

Ein Beispiel für die Anwendung einer Divide & Conquer Strategie ist der Algorithmus, der einen mathematischen Ausdruck berechnet. Das große Problem wird in die Teilprobleme aufgespalten, den Wert des Terms links des Operators und rechts des Operators zu bestimmen. Die beiden Teillösungen werden zusammengesetzt, indem die Teilergebnisse mit dem Operator verknüpft werden. So entsteht aus den Teillösungen wieder die Gesamtlösung.

Divide& Conquer Strategien haben häufig rekursiven Charakter, denn auf die ermittelten Teilprobleme wird ja erneut die Divide & Conquer-Strategie angewandt.

4.4 Backtracking Strategie

Die Backtracking-Strategie unterscheidet sich von der Divide and Conquer Strategie wesentlich. Auch hier wird im Normalfall eine rekursive Implementierung gewählt, das Vorgehen ist jedoch ein anderes. Überlicherweise stellt man einen potentiellen Teilösungsvorschlag zur Verfügung und basierend hierauf wird rekursiv getestet, ob es mit dieser Teillösung möglich ist, die Gesamtlösung herbeizuführen. Gelingt dies nicht, so wird ein Schritt zurückgegangen (backtrack), eine neue Teillösung erzeugt und der Prozess wiederholt sich erneut.

4.4.1 Das Damen-Problem

Problemstellung: Auf einem Schachbrett sind n Damen (die Schachfiguren), so zu verteilen, daß keine Dame eine andere schlagen kann.

- Ein Backtracking-Algorithmus, der dieses Problem löst, könnte in umgangssprachlicher Notation so aussehen:
- Belege ein Feld mit der Dame
- Kann die Dame von einer der bestehenden Damen geschlagen werden, so verwerfe die Lösung und gehe einen Schritt zurück, d.h. suche neue Position für Dame
- Sind alle Damen verteilt, so gib die Lösung aus

Diese Beschreibung skizziert schon grob einen Backtracking-Algorithmus, bei einer konkreten Implementierung muß man nun eine geeignete Strategie finden, um die Positionierung der Damen so effektiv wie möglich auszuführen.

4.5 Singleton-Pattern

In der Programmierpraxis kann es vorkommen, daß man von einer Klasse nur genau eine Instanz bilden darf, wie aber ist dies am besten zu realisieren. Am einfachsten ist hier die Verwendung des sogenannten *Singleton-Pattern*, welches die gewünschte Funktionalität zur Verfügung stellt.

```
class Singleton
{
  private static Singleton Instanz=new Singleton();

  public static Singleton getInstance()
  {
    return(Instanz);
  }

  private Singleton()
  {
    ...
  }
}
```

Damit der Benutzer nicht unkontrolliert Instanzen der Singleton-Klasse erzeugen kann, sind die Konstruktoren private deklariert und somit von außen nicht zugreifbar. Die Aufgabe, das Objekt zu liefern übernimmt eine statische Methode, also eine Klassenmethode. Die einzige Instanz der Klasse ist ebenso in einer statischen Variable gespeichert. Diese statische Variable wird belegt, wenn das erste Mal auf das Objekt zugegriffen wird, in unserem Fall beim Aufruf der Methode *getInstance()*. Damit kann nur genau eine Instanz erzeugt werden, wiederholte Aufrufe von *getInstance()* liefern immer dasselbe Objekt.

Kapitel 5

Ein paar Worte zur OOP

5.1 Einleitung

Bevor man mit der Einführung in die objektorientierte Softwareentwicklung und der damit verbundenen Prinzipien beginnen kann, muß man sich das Grundziel jeder Softwareentwicklung und das allgemeine Verfahren zur Erlangung dieses Ziels vor Augen führen. Jedes Stück Software ist die Formulierung eines Modells aus unserer Umwelt in die Sprache des Computers. Ein Problem aus der realen Welt wird in ein Problem des Computermikrokosmos überführt.

Ein Charakteristikum dieses Transformationsprozesses ist das Prinzip des selektiven Ignorierens. Dabei werden für die Modellierung unwesentliche Bestandteile des Problems der realen Welt so behandelt, als ob sie überhaupt nicht vorhanden wären. Ein Beispiel verdeutlich dieses Prinzip: Für eine Autovermietungsfirma soll ein System entwickelt werden, welches den Vorgang des Verleihens, der Rückgabe und der Rechnungserstellung für den Fuhrpark des Unternehmens zur Verfügung stellt. Selektives Ignorieren bedeutet in diesem Fall, daß Fahrzeugattribute wie zum Beispiel Farbe nicht in das zu erstellende Computermodell einfließen, denn diese Informationen sind für das eigentliche Problem und damit für das zu erstellende Modell nicht relevant. Die Erstellung des Modells bedeutet in einer ersten Phase also die Feststellung der wesentlichen und der unwesentlichen Merkmale der zu modellierenden Objekte und in einer zweiten Phase quasi das Vergessen der unwesentlichen Merkmale. In der dritten Phase, der eigentlichen Phase der Modellierung werden dann die erkannten wesentlichen Merkmale in ein Modell überführt, welches dann eine optimale Grundlage zur Problemlösung bereitstellt.

Aus dem bisher Gesagten kann man folgendes ableiten:

- Ein Modell ist spezifisch für eine konkrete Problemstellung

- Nur ein spezifisches Modell erlaubt die effiziente Problemlösung

Zu beiden Punkten sind nicht viele Worte zu verlieren, sie sind eigentlich eine direkte Folgerung aus dem Bisherigen. Die Eigenschaft, spezifisch zu sein, ergibt sich aus der Anwendung des Prinzips des selektiven Ignorierens. Hierdurch werden Merkmale herausgefiltert, die für das Problem irrelevant sind. Auf der anderen Seiten wird durch das Fehlen eben dieser Merkmale die Verwendung des gebildeten Modells für andere Problemstellungen verhindert. Dieses spezifische Modell ist deshalb so effizient, weil unnötige Daten nicht "mitgeschleppt" werden. Jedes Hinzufügen von Merkmalen, die nicht gebraucht werden, bläht das Modell auf und macht die Bearbeitung weniger effizient.

5.2 Modellbildung mit Hilfe der Methoden der objektorientierten Programmierung

Bisher wurde nur festgestellt, wie in groben Zügen die Entwicklung eines Modells abläuft, Besonderheiten wurden nicht berücksichtigt, vielmehr sollte das hinter allen Modellierungstechniken steckende, allgemeine Prinzip dargestellt werden. Bevor man die objektorientierte Programmierung als Hilfsmittel zur Programmierung anwenden kann, muß man die Grundprinzipien, die hinter dieser Technik stehen, verstehen.

5.2.1 Objekte, überall Objekte!

Die Welt ist voll von Objekten. Jeder Gegenstand, den der Blick streift, ist ein Objekt. Doch was macht eigentlich ein Objekt, insbesondere ein Objekt im Sinne einer Programmiersprache, überhaupt aus? Folgende zwei Punkte charakterisieren ein Objekt:

- Ein Objekt besitzt Attribute
- Ein Objekt besitzt Methoden

In seinen Attributen speichert das Objekt seinen aktuellen Zustand, mit den Methoden liest es seinen Zustand aus, schreibt einen neuen Zustand oder ändert mit Hilfe seiner Methoden den aktuellen Zustand. Attribute sind die Daten des Objekts und die Methoden die Vorschriften, wie mit diesen Daten zu verfahren ist.

5.2.2 Das Prinzip der Kapselung

Ein grundlegendes Prinzip der Objektsichtweise ist das Prinzip der Kapselung. Es ist eigentlich ein sehr einfaches Prinzip und besagt das Folgende:

- Einzig und allein das Objekt kann seine Attribute ändern, andere Objekte haben dazu keine Berechtigung

Das Objekt ist also sein eigener Herr, niemand kann ohne das Einverständnis des Objektes die Daten ändern. Möchte man, daß das Objekt die Daten ändert, so kann man ihm dies durch eine Nachricht mitteilen. Das Versenden einer Nachricht an ein Objekt entspricht dem Aufruf einer Methode des Objekts. Die Methode erledigt dann das Ändern oder Abfragen der Daten in der in ihr festgelegten Art und Weise und gewährleistet dadurch, daß das Objekt nicht in einen inkonsistenten Zustand gelangen kann.

Der Vorteil der Kapselung liegt klar auf der Hand: Solange sich nicht die Methoden ändern, bleibt es für Außenstehende unbedeutend, ob das Objekt intern seine Daten plötzlich auf eine ganz andere Art und Weise organisiert. Die Schnittstelle, das sogenannte Interface des Objektes, bleibt nach außen hin konstant und unverändert und macht somit Benutzer des Objekts unabhängig von der internen Repräsentation des Objekts.

Damit realisiert das Prinzip der Kapselung ein sogenanntes Black-Box-Verhalten. Der interne Aufbau ist unbekannt, nur die Interaktionsschnittstelle und die Reaktion des Objektes ist im Wissensbereich des Nutzers.

5.2.3 Klassen - zusammenfassen von gleichartigen Objekten

Was haben ein Ford, ein Audi und ein VW gemeinsam? Nun, alle drei lassen sich als PKW bezeichnen. Ein PKW schließlich besitzt (normalerweise) vier Räder, ein Lenkrad und noch ein paar andere Sachen, die ihn zu einem fahrbaren Untersatz machen, der Personen möglichst schnell und sicher von A nach B befördert.

In der objektorientierten Programmierung ist PKW eine Klasse und ein Ford, ein Audi und ein VW sind spezielle Ausprägungen dieser Klasse. Eine Klasse ist damit soetwas wie eine Schablone, die den prinzipiellen Aufbau der Objekte festlegt, die vom Typ dieser Klasse sind. In der Klasse sind alle Attribute und Methoden vorhanden, die die Objekte dieser Klasse besitzen und mit denen sie mit ihrer Umwelt interagieren können. Der Unterschied zwischen Ford und VW liegt nur in ihren Attributen, also den Zuständen der Objekte. Ein Ford kann zum Beispiel 75 PS Leistung besitzen, ein VW nur 60 PS. Beiden gemeinsam ist jedoch, daß sie überhaupt eine Leistung besitzen und diese in beiden Fällen auf die gleiche Art und Weise abgefragt und benutzt wird.

Eine Klasse modelliert die Gemeinsamkeiten, die alle Objekte, die zu derselben Klasse gehören, besitzen.

5.2.4 Interfaces - Festlegen einer gemeinsamen Schnittstelle

Da wir nun schon festgestellt haben, daß VW und Ford trotz unterschiedlicher Markennamen doch gewisse Gemeinsamkeiten im Bezug auf das Objekt, welches sie beschreiben, besitzen, kann man eine noch viel allgemeinere Frage formulieren: Was haben ein Schiff, ein Flugzeug und ein Auto gemeinsam? Auf den ersten Blick würde die Antwort wahrscheinlich "Gar nichts!" lauten, doch auch wenn es zuerst verwirrend erscheinen mag, so existieren doch Gemeinsamkeiten zwischen diesen verschiedenen Dingen. Zum Beispiel werden alle drei mit Hilfe von Erdölprodukten betrieben, beim Flugzeug das Kerosin, beim Auto das Benzin und beim Schiff das nicht sonderlich umweltfreundliche Schweröl. Obwohl diese drei Gegenstände wohl auf keinen Fall in eine gemeinsame Klasse gehören, denn die Attribute sind zu unterschiedlich, so teilen sie doch eine gemeinsame Schnittstelle, ein Interface. Alle drei können eine Funktion besitzen, die den Verbrauch pro hundert Kilometer zurückgelegter Strecke liefert.

Ein Interface definiert eine Liste von Methoden, die alle Objekte besitzen, die dieses Interface implementieren. Bei Kenntnis des Interfaces ist es möglich mit den Objekten zu kommunizieren, indem an diese Nachrichten gesendet werden, was ja, wie schon festgestellt, gleichbedeutend mit einem Methodenaufruf ist.

Interfaces in Java

Java ist eine der wenigen Sprachen, die Interfaces explizit als Sprachkonstrukt anbieten, bei Sprachen wie C++ muß man Interfaces mit Hilfe abstrakter Klassen und Mehrfachvererbung - diese allerdings nur ansatzweise - simulieren.

In Java sieht eine Interface-Definition wie folgt aus:

```
interface EinInterface
{
  public void Methode1(...);
  public double Methode2(...);
  ...
}
```

Eine Klasse verwendet das Interface, indem es dieses implementiert, in Java sähe dies in etwa wie folgt aus:

```
class TestKlasse implements EinInterface
{
  public void Methode1(...)
  {
    ...
  }
  ...
```

```
}
```

Wird ein Interface nicht vollständig implementiert, so ist die Klasse automatisch abstrakt und von ihr können damit keine Instanzen gebildet werden.

5.2.5 Vererbung - Durch Gemeinsamkeiten Code wiederverwenden

Weiter oben wurde ja schon festgestellt, daß sowohl Ford als auch VW der Klasse PKW zuzuordnen sind. Wie aber kann man nun eine Klasse Van bilden, die Personenkleintransporter darstellt? Offensichtlich besitzt die Klasse Van alle Eigenschaften der Klasse PKW, was ja auch nicht verwunderlich ist, denn ein Van ist eine Art von PKW.

In der objektorientierten Programmierung wird die Tatsache "ist eine Art von..." mit Hilfe des Prinzips der Vererbung in die Programmiersprache abgebildet. Die Superklasse, in unserem Fall die Klasse PKW "vererbt" an die Subklasse, in unserem Fall die Klasse Van, alle Attribute und Eigenschaften, die sie besitzt.

Auf dieses "Erbe" kann sich die Klasse Van jederzeit berufen, d.h. die Klasse Van kann mit den Methoden und Attributen der Klasse PKW uneingeschränkt arbeiten. Aber ein Van ist natürlich mehr als nur ein PKW. Ein Van bietet mehr Platz, hat mehr Sitze und vielleicht sonst noch einige Ausstattungsmerkmale, die ein normaler PKW nicht besitzt. Van erbt von PKW und es ist Van vollkommen freigestellt, was es mit dem Erbe anstellen will. Die Klasse kann alles so belassen wie es ist, was allerdings nicht sinnvoll ist oder aber, was viel eher der Fall sein dürfte, neue Attribute und/oder Methoden hinzufügen, die aus einem PKW eben einen Van machen.

Was sind nun eigentlich die Vorteile der Vererbung? Nun, sowohl die Klasse PKW als auch die Klasse Van können von zwei unterschiedlichen Entwicklern hergestellt worden sein. Nun erfährt der PKW-Klassen Entwickler einen Geistesblitz und schlagartig ist ein PKW viel effektiver programmiert als zuvor. Der Programmierer der Van-Klasse war schlau und hatte schon beim Design festgestellt, daß ein Van eben ein PKW ist und dementsprechend das Prinzip der Vererbung angewendet. Natürlich erbt Van jetzt auch die verbesserten Eigenschaften von PKW, profitiert also direkt von den Verbesserungen die der andere Entwickler gefunden und umgesetzt hat. Dies alles geschieht, ohne daß an der Klasse Van auch nur eine einzige Zeile Code verändert werden muß, Van erbt einfach die guten Eigenschaften von PKW und da die Schnittstelle konstant geblieben ist, muß er sich, da die Datenkapselung ja eingehalten wurde, um nichts kümmern.

Um es also kurz zu machen: Vererbung vermindert den Codeumfang, man kann auf der Arbeit von anderen aufbauen, muß also nicht permanent das Rad aufs Neue erfinden. Außerdem steigt die Programmsicherheit und Stabilität, da man auf bereits erprobte Klassen zurückgreifen kann, die (hoffentlich!) schon eine ausführliche Fehlersuchprozedur hinter sich haben.

Vererbung in Java

Java kennzeichnet die oben beschriebene Vererbung mit Hilfe des Schlüsselwortes extends. Vererbung wird in Java wie folgt formuliert:

```
class TestKlasse extends EineSuperKlasse
{
  ...
}
```

Vererbung in C++

C++ bietet im Gegensatz zu Java weitere Arten von Vererbung (siehe dazu weiter hinten in diesem Abschnitt). Die im Text angesprochene Art von Vererbung wird mit dem Schlüsselwort *public* realisiert und sieht dann wie folgt aus:

```
class TestKlasse : public EineSuperklasse
{
  ...
};
```

Mehrfachvererbung

Wie man im richtigen Leben das Glück haben kann, gleich mehrfach als Erbe eingesetzt worden zu sein, so ist dies auch in der Welt der objektorientierten Programmierung möglich. Eine Klasse kann mehrere Superklassen haben, von denen es Eigenschaften (Attribute) und Verhalten (Methoden) übernimmt und wiederverwertet.

Mehrfachvererbung in C++

Im Gegensatz zu Java ist es möglich, das Prinzip der Mehrfachvererbung anzuwenden. Die verschiedenen Superklassen werden durch Kommata getrennt aufgeführt. Damit erhält man Folgendes:

```
class TestKlasse: public SuperKlasse1, public SuperKlasse2, ...
{
  ...
};
```

Bei der Mehrfachvererbung kann es zu Mehrdeutigkeiten kommen. Diese treten immer dann auf, wenn eine Klasse mehrmals als indirekte Basisklasse der abgeleiteten Klasse auftritt.

Ein Beispiel verdeutlicht dies:

```
class A
{
  ...
};

class B : public A
{
  ...
};

class C: public A, public B
{
  ...
};
```

Klasse C enthält damit Methoden und Daten der Klasse A doppelt. In diesem Fall ist nicht mehr eindeutig feststellbar, welche Methoden und welche Attribute gemeint sind. Diese Mehrdeutigkeit kann man auflösen, in dem man die entsprechenden Methoden und Attribute mit dem Klassennamen eindeutig qualifiziert. Eine andere Alternative, die C++ bietet ist das Prinzip der sogenannten virtuellen Vererbung. Mit dem Schlüsselwort *virtual* wird sichergestellt, daß Objekte nicht mehrfach vorhanden sind. Unter Anwendung der virtuellen Vererbung sähe das obige Beispiel dann wie folgt aus:

```
class A
{
  ...
};

class B : public virtual A
{
  ...
};

class C: public A, public B
{
  ...
};
```

Liskovsches Substitutionsprinzip

Vererbung modelliert den Begriff “ist eine Art von” einer Beziehung in der realen Welt auf der Ebene der objektorientierten Umsetzung. Das Liskovsche Substitutionsprinzip besagt nun das Folgende:

Ein Objekt der Subklasse kann jederzeit an die Stelle eines Superklassen-Objekts treten, ein Superklassenobjekt kann also ersetzt (substituiert) werden.

Wird im Rahmen des objektorientierten Modellierunsprozesses dieses Prinzip für mehrere Klassen festgestellt, so ist das Prinzip der Vererbung anwendbar.

Upcasting

Jede Klasse besitzt mindestens eine Superklasse. Durch das Vererbungsprinzip ist jede Subklasse eine Art von Superklasse, nur in einer spezielleren Ausprägung.

Upcasting bezeichnet die Umwandlung eines Subklassen-Objektes in ein Superklassen-Objekt und ist immer möglich zwischen Objekten, die durch das Prinzip der Vererbung miteinander in Verbindung stehen.

Downcasting

Der Begriff Downcasting beschreibt den genau entgegengesetzten Prozeß: Ein Objekt der Superklasse wird in ein Objekt der Subklasse umgewandelt. Im Vergleich zum Upcasting ist der Prozeß des Downcastings nicht generell möglich. Erlaubt ist er nur, wenn das eigentliche Objekt, auf das eine Referenz vom Typ der Superklasse verweist, in Wirklichkeit ein Objekt der Subklasse ist. Downcasting ist also nur dann möglich, wenn diesem zuvor ein Upcast vorangegangen ist.

Verschiedene Arten von Vererbung

Der Begriff Vererbung wird neben dem oben beschriebenen Sachverhalt der Modellierung einer “ist eine Art von” Beziehung auch noch auf andere Weise verwendet. Bekannt ist insbesondere die sogenannte private-Vererbung. Hierbei handelt es sich um eine sogenannte Implementierungsvererbung, es wird nur die Implementierung, nicht aber das nach außen sichtbare Interface geerbt. Dies bedeutet, daß von innerhalb der Klasse auf alle Methoden der Superklasse zugegriffen werden kann, von außen sind diese Fähigkeiten jedoch nicht sichtbar. Allgemein bedeutet Vererbung, daß die Zugriffsmodifizierer private, protected und public in entsprechender Weise verändert werden, wodurch die Sichtbarkeit von Methoden und Attributen verändert wird. Dabei ist festzustellen, daß die Zugriffsmodifizierer nur noch verschärft werden können, d.h. die Möglichkeiten des Zugriffs werden eingeschränkt. Eine Übersicht gibt die folgende Tabelle:

Ableitung	Zugriffe in der Basisklasse	Zugriff in der abgeleiteten Klasse
public	public	public
	protected	protected
	private	⋆
private	public	private
	protected	private
	private	⋆
protected	public	protected
	protected	protected
	private	⋆

Der Stern zeigt an, daß ein Zugriff auf diese Elemente nicht möglich ist.

5.2.6 Abstrakte Klassen

Neben den "normalen" Klassen gibt es in der Welt der Objektorientierung auch noch sogenannte abstrakte Klassen. Worin unterscheiden sich nun abstrakte Klassen von anderen Klassen? Nun, bei abstrakten Klassen sind einige der Methoden nicht implementiert, ihre Realisierung wurde auf später verschoben. Konkret bedeutet dies, daß man andere Klassen von der abstrakten Klasse erben lassen muß und dabei die nicht definierten, also abstrakten Methoden mit Leben füllt, ihre Funktionsweise also definiert.

Und wofür kann man soetwas verwenden? Am besten bleiben wir beim Beispiel des PKW. Heutzutage hat man als PKW-Fahrer die Auswahl zwischen drei verschiedenen Arten von Kraftstoffen: Diesel, Benzin und Super. Dummerweise sind die verschiedenen Kraftstoffarten auch unterschiedlich teuer. Nehmen wir an, unsere PKW-Klasse soll eine Funktion zur Verfügung stellen, die die bisherigen Kraftstoffkosten berechnet. Dazu benötigt diese Funktion den aktuell zurückgelegten Weg des Fahrzeuges, sowie die Kosten des Kraftstoffes, zum Beispiel pro hundert Kilometer. Eine mögliche Lösung des Modellierungsproblems wäre die Folgende:

Die Klasse PKW wird zu einer abstrakten Klasse mit einer abstrakten Methode, die den Preis des Kraftstoffs zurückliefert. Unsere Methode, die die bisher angefallenen Kosten berechnen soll, geht nun wie folgt vor. Zuerst werden die bisher zurückgelegten Kilometer ermittelt, dann die Kraftstoffkostenfunktion aufgerufen und aus dem zurückgelieferten Wert die bisher angefallenen Kosten ermittelt. Zu beachten ist hierbei Folgendes: Die Kostenberechnungsfunktion kann komplett formuliert werden, obwohl noch gar nicht bekannt ist, wie die Kosten des jeweiligen Kraftstoffs sind (diese Methode ist ja abstrakt!).

Von der Klasse PKW können nun Unterklassen gebildet werden. Zum Beispiel sind dies BenzinPKW, DieselPKW und SuperPKW, die jeweils ihre eigene Kraftstoffkostenfunktion besitzen. Bei allen drei arbeitet die Kostenberechnungsfunktion korrekt, denn sie kann ja aus der Superklasse übernommen werden.

Abstrakte Klassen in Java

Java bietet für die Kennzeichnung abstrakter Klassen das Schlüsselwort *abstract*. Direkt vor die Klasse geschrieben macht es aus der Klasse eine abstrakte Klasse, vor eine Methode geschrieben, wird diese Methode als abstrakt definiert und macht damit automatisch die umgebende Klasse abstrakt.

Konkret:

```
abstract class EineKlasse
{
  ...
}
```

Alternativ:

```
class EineKlasse
{
  abstract void Methode1(...);
  ...
}
```

Abstrakte Klassen in C++

C++ bietet kein Schlüsselwort für die Definition von abstrakten Klassen. Vielmehr gibt es die sogenannten pure virtual functions, die nichts anderes sind, als abstrakte Funktionen. Genauso wie in Java wird eine Klasse automatisch abstrakt, wenn sie eine oder mehrere abstrakte Methoden enthält:

```
class EineKlasse
{
  virtual void Methode1(...)=0;
  ...
};
```

5.2.7 Polymorphie - eine Methode hat viele Gesichter

Polymorphie bedeutet soviel wie Vielgestaltigkeit und bewirkt in der objektorientierten Programmierung ein dynamisches Verhalten, das von den Typen der bearbeiteten Objekte abhängt. Grundlage für die Polymorphie bildet das weiter vorne schon angesprochene Vererbungsprinzip.

Vererbung bedeutet, daß ein Objekt die Eigenschaften und Methoden seines Vergängers übernimmt und anwenden kann. Dies bedeutet, daß eine Methode, die in der Superklasse definiert wurde auch in allen Subklassen dieser Superklasse vorhanden ist. Jede Klasse, einschließlich der Superklasse kennt also diese Methode. Ein Objekt der Subklasse kann ohne Probleme in ein Objekt der Superklasse umgewandelt werden, dies ist das sogenannte Upcasting. Nun kann man ein Objekt der Subklasse auch mit Hilfe einer Superklassenreferenz ansprechen, insbesondere die Methoden aufrufen. Polymorphie bedeutet, daß nun nicht an Hand der Referenz, die auf das Objekt verweist, entschieden wird, welche Methode, konkreter welche Methode aus welcher Klasse, aufgerufen wird, sondern an Hand des eigentlichen Objekts, auf das referenziert wird. Dies bezeichnet man als sogenanntes dynamisches Verhalten, oder anders als Polymorphie.

Mit Hilfe der Polymorphie ist es möglich, Operationen über eine Menge von Objekten, die durch das Prinzip der Vererbung miteinander in Verbindung stehen allgemein zu formulieren. Um zum Beispiel die bisher verbrauchte Kraftstoffmenge einer Kraftfahrzeugflotte zu ermitteln genügt es bei Anwendung des Prinzips der Polymorphie, die Iteration über die Objektmenge mit Hilfe der Superklasse aller Objekte, in diesem Fall der Klasse PKW zu formulieren. Obwohl vom Namen her immer die gleiche Methode aufgerufen wird, so gelingt es dank der Polymorphie, die für das jeweils bearbeitete Objekt spezifische Methode aufzurufen.

5.2.8 Aggregation , Assoziation und Benutzung- Vererbung ist nicht alles

Mit Hilfe der Vererbung kann man schon viele Eigenschaften aus der zu modellierenden Menge auf die Computerumwelt abbilden. Doch Vererbung ist nicht alles, mit Vererbung allein kann nicht alles modelliert werden. Und auch nicht überall, wo man Vererbung theoretisch anwenden könnte, ist sie vom logischen Standpunkt aus auch korrekt angewendet.

Assoziation - Beziehungen zwischen Objekten

Sind zwei oder mehr Objekte miteinander assoziiert, so stehen sie in irgendeiner Weise zueinander in Beziehung. Zwischen einem Objekt "Mann" und einem Objekt "Kind" kann z.B. die Beziehung "ist Vater von" und "ist Kind von" bestehen. Beziehungen können einseitig oder aber auch, wie im Beispiel zweiseitig sein.

Aggregation - "ist ein Teil von"

Ein Objekt kann aus mehreren anderen bestehen. Ein PKW besteht zum Beispiel aus den Teilobjekten "Getriebe", "Rad", usw. Das resultierende Objekt ist eine Zusammenfassung der einzelnen Objekte zu einem neuen, das Objekt bildet quasi die zusammenfassende Klammer.

Wo aber liegt nun der Unterschied zwischen einer Aggregation und einer Assoziation? Ein PKW könnte doch genauso mit den Objekten "Getriebe usw. in der Beziehung "ist ein Teil von" stehen. Im Prinzip schon, doch verwendet man das Prinzip der Aggregation normalerweise in folgendem Fall:

Objekte werden zu einem neuen Objekt aggregiert, wenn die Teilobjekte mit dem Erlöschen der Existenz des Hauptobjekts auch aufhören zu existieren. Kurz gesagt: Assoziation immer dann, wenn die Teilobjekte auch für sich alleine existieren können (wenn der Vater nicht da ist, dann hört damit nicht die Existenz des Kindes auf) Aggregation immer dann, wenn die Lebensdauer des Hauptobjekts gleichzeitig auch die Lebensdauer der Teilobjekte ist (ein Getriebe kann in sinnvoller Weise für sich allein nicht existieren).

Eine Alternative zur Verwendung der Aggregation ist die sogenannte, weiter oben schon angesprochene Implementierungsvererbung. Die Verwendung der Implementierungsvererbung erspart das Schreiben von sogenannten Adapterfunktionen, also die Verwendung von Funktionen, deren einzige Aufgabe es ist, die gleich lautende Funktion des aggregierten Objekts aufzurufen.

Benutzung - ein Objekt delegiert Aufgaben

Mit den beiden vorgestellten Prinzipien läßt sich schon ein Großteil der Modellierungsprobleme in die objektorientierte Welt abbilden. Was noch verbleibt ist das Prinzip der Benutzung und damit ist die Liste der zusätzlich zur Vererbung vorhandenen Möglichkeiten praktisch komplett.

Ein Objekt kann andere Objekte benutzen, um seine Aufgaben zu erfüllen. Dabei steht das Objekt weder in irgendeiner Beziehung zu dem Objekt, noch ist das benutzte Objekt ein Teilobjekt dieses Objekts. Das Objekt Geldautomat bedient sich der Leistungen des Objekts Konto, doch ist ein Konto weder ein Bestandteil eines Geldautomaten, noch steht der Geldautomat zu dem Konto in irgendeiner Beziehung. Die Benutzung kann ein einmaliger oder zeitlich beschränkter Vorgang sein, genauso kann die Benutzung sich aber über die gesamte Lebensdauer des Objektes hinziehen.

Die ideale Vorgehensweise, um eine Benutzung zu realisieren, besteht in der Verwendung von Interfaces, die schon weiter vorne im Text angesprochen wurden.

Abschließende Bemerkung

Abschließend bleibt festzustellen, daß sich die genannten Prinzipien bezüglich der Realisierung innerhalb einer objektorientierten Programmiersprache nicht voneinander unterscheiden müssen. Zum Beispiel werden Assoziation und Benutzung programmiertechnisch mit Hilfe von Referenzen ausgedrückt. Auf der

Ebene des Quelltextes, dem letzten Schritt der objektorientierten Modellierung, ist es also nicht unbedingt möglich, die verwendeten Prinzipien zu identifizieren. Dies bedeutet, daß die objektorientierte Designphase die verwendeten Prinzipien klar dokumentieren muß, um die später anstehende Wartung und Verbesserung ohne Probleme zu ermöglichen.

5.2.9 Vom Problem zu Programm - der Einsatz der objektorientierten Methoden

An diesem Punkt sind die grundlegenden Prinzipien der objektorientierten Programmierung abgehandelt worden und es ist nun an der Zeit, ein zumindest prinzipielles Vorgehen festzulegen, um ein gegebenes Problem, also ein Modell, in die Welt der objektorientierten Programmierung zu übertragen.

Modellierung und damit auch die objektorientierte Modellierung ist in gewisser Weise eine Kunst. So wie es für ein und dasselbe Problem viele verschiedene mögliche Lösungen gibt - man denke hier nur an die fast unüberschaubare Anzahl von Sortieralgorithmen - so ist auch der Prozeß der Modellierung keinesfalls ein eindeutiger und nach Schema durchführbarer Prozeß. Zu modellieren heißt kreativ sein und unter Umständen die Dinge aus anderen Perspektiven zu sehen, um eine möglichst optimale Darstellung im Computer zu erhalten. Daher kann im Folgenden nur eine grobe Orientierung gegeben werden, die zumindest ein bißchen als Hilfestellung benutzt werden kann.

Schritt 1 - Identifizierung der Klassen

Das Problem wird auf die vorhandenen Klassen untersucht, wobei nur die wirklich nötigen Klassen betrachtet werden, das Prinzip der selektiven Ignoranz wird also angewendet. Ebenso wird das Prinzip auf jede vorhandene Klasse angewendet, wodurch für das Problem unnötige Attribute und Methoden gar nicht erst ihren Weg in die nachfolgenden Teile der Modellierung finden.

Schritt 2 - Identifizierung von Vererbungshierarchien

Die gebildeten Klasse werden daraufhin untersucht, ob das Vererbunsgprinzip anwendbar ist. Dazu wird zuerst mit Hilfe des Liskovschen Substitutionsprinzips versucht, eine Superklasse zu finden. Schlägt dies fehl, liegt jedoch die Vermutung nahe, daß eine solche Superklasse existiert, so wird diese zusätzlich modelliert, in diesem Fall als abstrakte Klasse. Mit dem zweiten Schritt ist die Modellmenge mehr oder minder geclustert, d.h. ähnliche Objekte werden im gleichen Klassencluster zusammengefaßt.

Schritt 3 - Aggregation, Assoziation und Benutzung testen

Im nächsten Schritt werden die genannten Eigenschaften getestet, wobei die jeweils genannten Bedingungen für die Gültigkeit einer solchen Objektbeziehungsform getestet und gegebenenfalls angewendet werden.

Zusammenfassende Bemerkung

Insbesondere Schritt 1 und 2 haben einen Einfluß auf die Güte des am Ende stehenden Objekts. Gerade diese beiden Schritte lassen sich nicht weiter formalisieren, hier liegt es im Ermessen und der Kreativität des Modellierers, eine möglichst optimale Lösung zu finden.

Die Verwendung von Vererbung und die Bildung abstrakter Klassen sollte nicht nur im Hinblick auf die momentane Optimalität betrieben werden, sondern auch der langfristigen Optimalität Rechung tragen. Die langfristige Optimalität bezeichnet die Fähigkeit des Modellsystem, Erweiterungen im bestehenden Modell einfach, problemlos und auch wieder optimal durchzuführen. Eine Lösung, die das momentane Modell optimal darstellt, jedoch kaum Möglichkeiten einer sinnvollen Erweiterung zuläßt, ist zwar kurzfristig optimal, jedoch keinesfalls langfristig. Gerade unter dem Aspekt, daß die Lebensdauer guter Softwaresysteme in Größenordnungen von zehn oder mehr Jahren liegen kann, ist die langfrsitige Optimalität ein wichtiges Gütekriterium.

Kapitel 6

Rekursion

Eng verbunden mit dem Divide & Conquer Prinzip ist die Verwendung von Rekursion. Rekursion ist dadurch gekennzeichnet, daß innerhalb einer Methode die Methode selbst wieder aufgerufen wird. Eine solche Methode bezeichnet man als rekursiv. In vielen Fällen ist Rekursion gegenüber einer herkömmlichen Lösung kürzer, eleganter und leichter zu durchschauen. Rekursion hat aber auch einen großen Nachteil: ein Programm, welches Rekursion verwendet ist langsamer als ein vergleichbares, nicht-rekursives Programm. Oberstes Ziel bei der Programmentwicklung ist aber die Wartbarkeit der Programme, wozu auch ein leichtes Verständnis des Programmtextes gehört. Rekursion gewinnt deshalb zwar nicht bezüglich Geschwindigkeit, wohl aber bezüglich Übersichtlichkeit und sollte deshalb an geeigneten Stellen wohlüberlegt eingesetzt werden.

6.1 Charakteristisches Aussehen einer rekursiven Funktion

Nachdem schon festgestellt wurde, wie eine rekursive prinzipiell aussieht, hier ein Java-Pseudocode:

```
class Rekursion
{
  int Methode(int n)
  {
    //Abbruchbedingung


//Alternativen, innerhalb denen bei Bedarf wieder Methode(...) aufgerufen wird
  }
}
```

Allgemein sieht eine rekursive Funktion also wie folgt aus:

- Eine rekursive Funktion enthält eine oder mehrere Abbruchbedingungen
- Eine rekursive Funktion enthält einen oder mehrfache Aufrufe ihrer selbst

6.2 Beispiele

Gerade das Prinzip der Rekusrion kann man am besten verstehen, wenn man konkrete Beispiele durchgeht und aus diesen die Informationen zieht. Die nachfolgenden Beispiele sind einfach gewählt worden, um das Grundprinzip deutlich zu machen. Nahezu jeder Algorithmus kann problemlos, sofern er mit Hilfe einer Divide & Conquer Strategie zu beschreiben ist, in Form eines rekursiven Algorithmus beschrieben werden.

6.2.1 Addition zweier natürlicher Zahlen

Das Problem ist einfach und schnell formuliert: Zwei Zahlen x und y sollen addiert werden, wobei x und y nur natürliche Zahlen, also ganze Zahlen >=0, sein können. Um eine rekursive Beschreibung dieses Problems zu finden, müssen wir uns zuerst eine geeignete Divide & Conquer Strategie überlegen. Ziel einer solchen Strategie ist es, ein komplexes Problem so aufzuspalten, daß die dabei entstehenden Teilprobleme wesentlich einfacher zu lösen sind.

Fall 1: x beliebig und y=0

In diesem Fall können wir das Ergebnis sehr einfach angeben, denn x+y=x+0 ist ganz einfach x. Tritt also der Fall y=0 ein, so kann unsere noch zu formulierende rekursive Funktion einfach den Wert x zurückliefern und sich dann beenden.

Fall 2: x und y beliebig

Offensichtlich ist es "schwer" x+y zu berechnen, wesentlich einfacher ist es, x+1+add(x,y-1) zu berechnen, denn hier muß man nur wissen, wie man 1 zu einer Variablen addiert bzw. 1 von einer Zahl subtrahiert. Damit ist das komplexe Problem auf ein einfacheres reduziert und zusätzlich wurde erreicht, daß die Addition von x+y auf die Addition von x und (y-1) zurückgeführt wurde. Durch diese Rückführung ist es möglich, die Rekursionsbedingung zu formulieren.

6.2.2 Die rekursive Addition

Kombiniert man beide Fälle, so hat man alles, was man für die rekursive Addition zweier natürlicher Zahlen benötigt. In einem Java- Quellcode sieht dies wie folgt aus:

```
class Addition
{
  int add(int x, int y)
  {
    if (y==0) return(x); //Abbruchbedingung

    return(x+1+add(x,y-1));
  }
}
```

6.2.3 Multiplikation zweier natürlicher Zahlen

Im vorherigen Abschnitt wurde mehr oder minder ausführlich die Addition als rekursive Funktion hergeleitet, für die Multiplikation erhält man mit analogen Überlegungen einen ähnlichen Algorithmus. Hier zeigt sich, daß eine rekursive Funktion durchaus mehr als eine Abbruchbedingung besitzen kann.

```
class Multiplikation
{
  int mult(int x, int y)
  {
    if (y==0) return(0); //1. Abbruchbedingung
    if (y==1) return(1); //2. Abbruchbedingung

    return(x+mult(x,y-1));
  }
}
```

6.2.4 Fakultätsberechnung

Rekursive Funktionen müssen auch nicht unbedingt immer zwei Parameter besitzen, wie das folgende Beispiel einer rekursiven Fakultätsfunktion belegt.

```
class Fakultät
```

```
{
  int fak(int n)
  {
    if (n==0) return(1);

    return(n*fak(n-1));
  }
}
```

Kapitel 7

Sichtbarkeit bei private, protected, public und friendly

innerhalb...	private	protected	public	friendly
der Klasse	⋆	⋆	⋆	⋆
anderer Klassen im selben Package		⋆	⋆	⋆
abgeleiteter Klassen im selben Package		⋆	⋆	⋆
anderen Klassen in anderen Packages			⋆	
abgeleiteter Klassen in anderen Packages		⋆	⋆	

Anmerkung: Der Modifizierer *friendly* wird nicht explizit geschrieben. Eine Variable oder eine Methode ist *friendly*, wenn keiner der Modifizierer *private*, *protected* oder *public* vor die Variable oder Methode gesetzt wird.

Kapitel 8

Prioritätenliste der Operatoren

Priorität	Operatoren	Assoziativität
0	=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ˆ=, |=	rechts nach links
1	?:	rechts nach links
2	||	links nach rechts
3	&&	links nach rechts
4	|	links nach rechts
5	ˆ	links nach rechts
6	&	links nach rechts
7	==, !=	links nach rechts
8	<>, <=, >=, instanceof	links nach rechts
9	<<, >>, >>>	links nach rechts
10	+, -(binär)	links nach rechts
11	/, *, %	links nach rechts
12	new	-
13	++, –(Präfix), +(unär), -(unär)	-
14	˜, !, () (Konvertierung)	rechts nach links
15	++, – (Postfix)	-
16	(), ., []	links nach rechts

Kapitel 9

Ganzzahldivisionen

In Java gibt es zwei Operatoren für die ganzzahlige Division:

- % liefert den Rest der Division
- / liefert das ganzzahlige Ergebnis der Division

Das Vorzeichen des Restes der Division richtet sich nach dem Vorzeichen des Dividenden, nicht des Divisors. Eine komplette Auskunft gibt die folgende Tabelle:

Vorzeichen	Division	Rest
+,+	+/+=+	+%+=+
+,-	+/-=-	+%-=+
-,+	-/+=-	-%+=-
-,-	-/-=+	-%-=-

Kapitel 10

Elementare Datentypen in Java

Typ	Speicherplatz (Byte)	Kleinster Wert	Größter Wert
boolean	$\frac{1}{8}$Byte=1 Bit	false	true
char	2	u\0000	u\ffff
byte	1	-128	127
short	2	-32.768	32.767
int	4	-2.147.483.648	2.147.483.647
long	8	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
float	4	±1.40129846432481707e-45	±3.40282346638528860e+38
double	8	±4.94065645841246544e-324	±1.79769313486231570e+308

Kapitel 11

Reservierte Schlüsselworte in Java 2

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	null
package	private	protected	public	return	short
static	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while

Anmerkung: Der Ausdruck *const* ist zwar ein reserviertes Schlüsselwort, wird aber momentan noch nicht von der Referenzimplementierung von SUN verstanden.

Index