Programmierung in Java

Einführung in die strukturierte und objektorientierte Programmierung

Kapitel 1.3, 2.1 - 2.11

Prof. Dr. Markus Eikelberg

Rel. 7.30

Für diese Unterlagen gilt:

- © 1999, 2020 Prof. Dr. Markus Eikelberg
- LaTeX Version, Dokument `javakurs-kap-1.3_2.1-2.11.pdf`
- Druckdatum 8. März 2020
- Alle Urheber- und Leistungsschutzrechte vorbehalten.
- Keine unerlaubte Weitergabe dieser Unterlagen, keine unerlaubte Vervielfältigung, Vermietung, Sendung, Aufführung, Veröffentlichung oder Verleih ohne das schriftliche Einverständnis des Autors.
- Hinweis zum Urheberrecht: Soweit nicht anders gekennzeichnet ist der Urheber stets identisch mit dem Verfasser einer zitierten Quelle
- Das Urheberrecht für alle nicht als Zitat gekennzeichneten Abbildungen liegt beim Autor.

Rechtliche Hinweise - Haftungsausschluss

Die Benutzung dieses Skripts und die Umsetzung der darin enthaltenen Informationen erfolgt ausdrücklich auf eigenes Risiko. Der Autor kann für Schäden jeder Art, die sich durch die Nutzung der Unterlagen ergeben, aus keinem Rechtsgrund eine Haftung übernehmen. Haftungsansprüche gegen den Autor wegen Schäden materieller oder ideeller Art, die durch die Nutzung oder Nichtnutzung der Informationen bzw. durch die Nutzung fehlerhafter und/oder unvollständiger Informationen verursacht wurden, sind grundsätzlich ausgeschlossen. Rechts- und Schadenersatzansprüche sind daher ausgeschlossen. Dieses Skript wurde mit größter Sorgfalt erarbeitet. Der Autor übernimmt jedoch keine Gewähr für die Aktualität, Korrektheit, Vollständigkeit und Qualität der bereitgestellten Informationen. Druckfehler und Falschinformationen können nicht vollständig ausgeschlossen werden. Der Autor übernimmt keine Haftung für die Aktualität, Richtigkeit und Vollständigkeit des Inhalts dieses Skripts, ebenso nicht für Druckfehler. Es kann keine juristische Verantwortung sowie Haftung in irgendeiner Form für fehlerhafte Angaben und daraus entstandenen Folgen vom Autor übernommen werden.

Inhaltsverzeichnis

1 Einführung in Java **3**
1.3 Software zur Programmierung in Java . . . 3
1.3.1 Java Development Kit . . . 3
1.3.2 Eclipse . . . 10

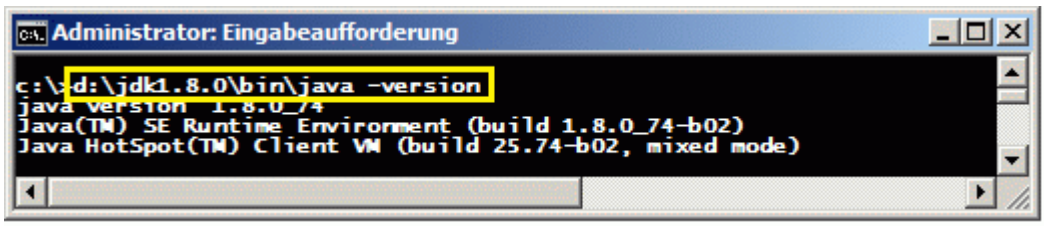
2 Objektorientierte Java Konsolprogramme **25**
2.1 Erweiterung der Ordnerstruktur für die Verarbeitung von Datei-I/O . . . 25
2.2 Grundelemente eines Java-Programms . . . 32
2.2.1 Die Java-Klasse . . . 32
2.2.2 Java-Anwendungen . . . 39
2.2.3 Unicode und Codierung unter Windows . . . 43
2.2.4 Lexikalische Elemente . . . 58
2.3 Elementare Datentypen und ihre Verarbeitung . . . 63
2.3.1 Elementare Datentypen . . . 63
2.3.2 Arithmetische Operatoren . . . 70
2.3.3 Objektbasierte Programmierung . . . 81
2.3.4 Das Paket de.eikelberg.iotools . . . 89
2.3.5 Ein- und Ausgabe elementarer Datentypen mit den Methoden des Paketes de.eikelberg.iotools . . . 91
2.4 Operatoren . . . 102
2.4.1 Typen von Operatoren . . . 102
2.4.2 Relationale Operatoren . . . 103
2.4.3 Logische Operatoren . . . 106
2.4.4 Zuweisungsoperatoren . . . 108
2.4.5 Weitere Operatoren . . . 110
2.4.6 Operator-Vorrangregeln . . . 114
2.5 Steuerungselemente . . . 116
2.5.1 Überblick . . . 116
2.5.2 if-Anweisungen . . . 117
2.5.3 switch-Anweisungen . . . 126
2.5.4 while-Schleifen . . . 136
2.5.5 do-Schleifen . . . 140
2.5.6 for-Schleifen . . . 144
2.5.7 Verwendung von break und continue in Schleifen . . . 153
2.6 Referenztypen . . . 155
2.6.1 Eigenschaften von Referenztypen . . . 155
2.6.2 Arrays . . . 158
2.6.3 String-Objekte . . . 170
2.6.4 StringBuffer-Objekte . . . 180
2.6.5 Ein- und Ausgabe von Arrays und String-Objekten . . . 188
2.6.6 Ein- und Ausgabe mit Standard Java Sprachmitteln . . . 196

2.7 Methoden . . . 203
2.7.1 Deklaration von Methoden . . . 203
2.7.2 Formale und aktuelle Parameter . . . 207
2.7.3 Beispiel zur Trennung von Steuerung und Verarbeitung . . . 213
2.8 Klassenbasierte Programmierung . . . 218
2.8.1 Der Begriff der klassenbasierten Programmierung . . . 218
2.8.2 Syntax der Klassendeklaration . . . 219
2.8.3 Klassen- und Instanzvariablen . . . 221
2.8.4 Klassen- und Instanzmethoden . . . 232
2.8.5 Konstruktoren . . . 242
2.8.6 Get- und Set-Methoden . . . 249
2.8.7 Modifizierer zur Sichtbarkeit . . . 253
2.8.8 Datenrückgabe aus Methoden mit Objekten . . . 260
2.8.9 Beispiele zum klassenbasierten Programmentwurf . . . 264
2.9 Objektorientierte Programmierung . . . 272
2.9.1 Begriffe der objektorientierten Programmierung . . . 272
2.9.2 Vererben und Überschreiben . . . 276
2.9.3 Techniken bei Vererbung . . . 293
2.9.4 Steuerung von Vererbungsvorgängen . . . 298
2.9.5 Exceptions . . . 310
2.10 Stream-Konzept . . . 320
2.10.1 Streams, Reader, Writer und Buffer bei Konsol-IO . . . 320
2.10.2 Dateizugriff . . . 325
2.10.3 Lesen von einer URL . . . 327
2.10.4 Lesen und Schreiben über einen Socket . . . 329
2.11 Generics . . . 333
2.11.1 Generische Klassen . . . 333
2.11.2 Kovarianz und Kontravarianz . . . 339
2.11.3 Reflection . . . 348

A Anhang **351**
A.1 Nicht als Bezeichner verwendbare Zeichenketten . . . 351
A.2 Anleitung zur Installation und Verwendung des Programms HIEW . . . 352

1.3 Software zur Programmierung in Java

In diesem Kapitel wird vorausgesetzt, dass das Java Development Kit in einem Verzeichnis `D:\jdk1.8.0` installiert ist. Sie können dies überprüfen, in dem Sie folgenden Befehl eingeben:



Java ist eine verhältnismäßig junge Programmiersprache, deren Geschichte 1991 begann. Ausgangspunkt war ein Projekt bei der Firma **Sun Microsystems** zur Entwicklung von Software für Geräte der Konsumelektronik. Bezüglich dieser ursprünglichen Zielrichtung war das Projekt jedoch zunächst nicht erfolgreich.

In den Jahren 1993/94 verbreitete sich jedoch unabhängig von dieser Entwicklung das **World Wide Web (WWW)** und mit den ersten HTML-Browsern entstand die Möglichkeit zur graphischen Darstellung von Informationen im Internet. Im Rahmen dieser Entwicklung erkannte man die durch Java im Bereich des Internet eröffneten Möglichkeiten. Ende 1995 kam mit dem Browser **Netscape Navigator 2.0** der erste Browser auf den Markt, der Java bei der Anzeige von WWW-Seiten umsetzen konnte. Dementsprechend wurde bei Sun Microsystems weiter an Java gearbeitet und zeitgleich mit dem Netscape Navigator 2.0 erschien Anfang 1996 der erste Java-Compiler, das **Java Development Kit (JDK) 1.0** von Sun Microsystems. Im April 2009 wurde Sun Microsystems vom Oracle Konzern gekauft, so dass Java aktuell ein Oracle Produkt ist.

Das Java Development Kit wurde von Sun Microsystems schnell und umfassend weiterentwickelt und wird kostenlos verteilt.

1.3.1 Java Development Kit

Das Java Development Kit (JDK) stellt neben dem Java-Compiler auch die virtuelle Maschine bereit, die für die Ausführung von Java-Programmen benötigt wird. Daher wird zumindest die Laufzeitumgebung (Java Runtime Environment JRE) aus dem JDK bei der Ausführung von Java-Programmen benötigt. Für die Softwareentwicklung bietet das JDK **keinerlei Bedienerkomfort** - man könnte auch sagen, daß der Bedienerkomfort des JDK der Arbeitsweise vor (!) der Einführung von Windows entspricht. Dementsprechend geschieht die Arbeit mit dem JDK auf einem Windows-Rechner auch in der sogenannten Konsolbox (unter Windows-Systemen zu erreichen über den Menüpunkt "Eingabeaufforderung"). Die verschiedenen angebotenen Entwicklungsumgebungen zu Java enthalten stets eine JDK-Version, auf die beim Compile zurückgegriffen wird. Damit wird auch bei der Verwendung einer Entwicklungsumgebung im Hintergrund stets eine JDK-Version verwendet.

1.3.1.1 Grundform des Aufrufs von Compiler und Virtueller Maschine (JVM)

Wir erklären die Grundform des Aufrufs von Compiler und Virtueller Maschine anhand des Beispielprogramms **bsp01004.Konsoladd**, das in Bezug auf den Java Quellcode abgesehen vom geänderten package-Namen ansonsten mit dem Beispielprogramm bsp01001.Konsoladd übereinstimmt. Das Programm bsp01004.Konsoladd.java können Sie wie auch alle anderen Beispielprogramme dieses Kurses auf zwei verschiedenen Wegen wie folgt finden:

Alle Programme sind nach folgendem Schema benannt: Ein Programm mit dem Namen **bspnnmmm.Progname** ist ein Beispielprogramm zu Kapitel nn. Den Quellcode **Progname.java** dieses Programms sowie alle zugehörigen Dateien finden Sie, indem Sie entweder

1. **auf der DVD** zu diesem Kurs im Verzeichnis `javakurs-beispiele\ebvnn\`**`jprnnmmm`**`\`**`bspnnmmm`** oder
2. **im FTP-Verzeichnis des Intranet-Servers des Institutes für Mathematik und Informatik (PC132W.bsnw.fb4.fh-bochum.de)** im Verzeichnis `eikelberg\javakurs\beispiele\ebvnn\`**`jprnnmmm`**`\`**`bspnnmmm`**.

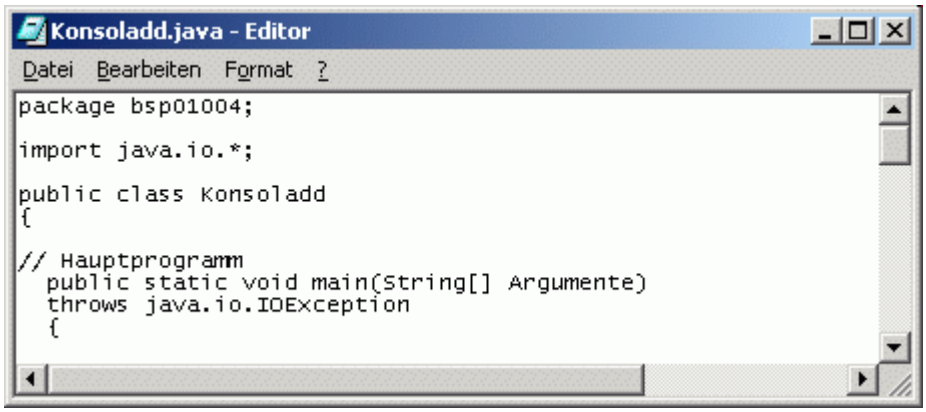
Bei der Java-Programmierung spricht man in der Regel statt von Programmen allgemeiner von sogenannten Klassen. Daher tritt im nachfolgenden Text oft allgemeiner die Bezeichnung **bspnnmmm.Klassenname** für die Quellcode-Datei **Klassenname.java** auf.

Der Server PC132W.bsnw.fb4.fh-bochum.de erlaubt in der Regel einen Zugriff via anonymous FTP (anonyme Anmeldung mit Benutzerkennung **anonymous** nach Eingabe von `ftp PC132W.bsnw.fb4.fh-bochum.de` in der Konsolbox oder aber nach Eingabe von `ftp://PC132W.bsnw.fb4.fh-bochum.de` in einem Browser eines beliebigen Rechners im Netzwerk der Hochschule Bochum bzw. bei Bestehen einer VPN-Verbindung zur Hochschule Bochum).

Der folgende Ablauf legt zugrunde, daß Sie das Verzeichnis `jpr01004` von der DVD oder vom FTP-Server in ein Verzeichnis `$daten\ebv01` auf einem Laufwerk Ihres Rechners mit dem Namen `E:` kopiert haben. Das Programm `Konsoladd.java` finden Sie dann im Verzeichnis

`E:\$daten\ebv01\jpr01004\bsp01004`

Sie können dann den Quellcode in `Konsoladd.java` wie in 1.2.1 beschrieben mit Hilfe eines Editors ansehen. Mit dem Windows-Editor **notepad** (Anwendung "Editor" in der Windows-Programmgruppe "Zubehör") erhalten Sie zum Beispiel folgende Bildschirmanzeige:

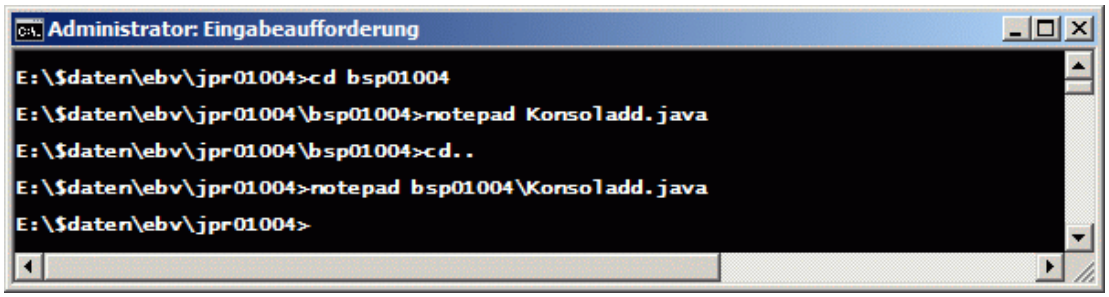


Sie können leicht sehen, daß der Quellcode von bsp01004.Konsoladd bis auf den neuen package-Namen identisch ist mit dem Quellcode von bsp01001.Konsoladd aus Kapitel 1.1 und 1.2.

Da die Arbeit mit dem JDK in der Konsolbox ("Eingabeaufforderung") geschieht, sei hier kurz der Aufruf von notepad aus der Konsolbox erklärt. Die **Syntax** (Beschreibung der sprachlichen bzw. "grammatischen" Regeln zur korrekten Eingabe eines Befehls oder einer Programmieranweisung) des **Aufrufs von notepad** lautet in der vollständigen Form wie folgt:

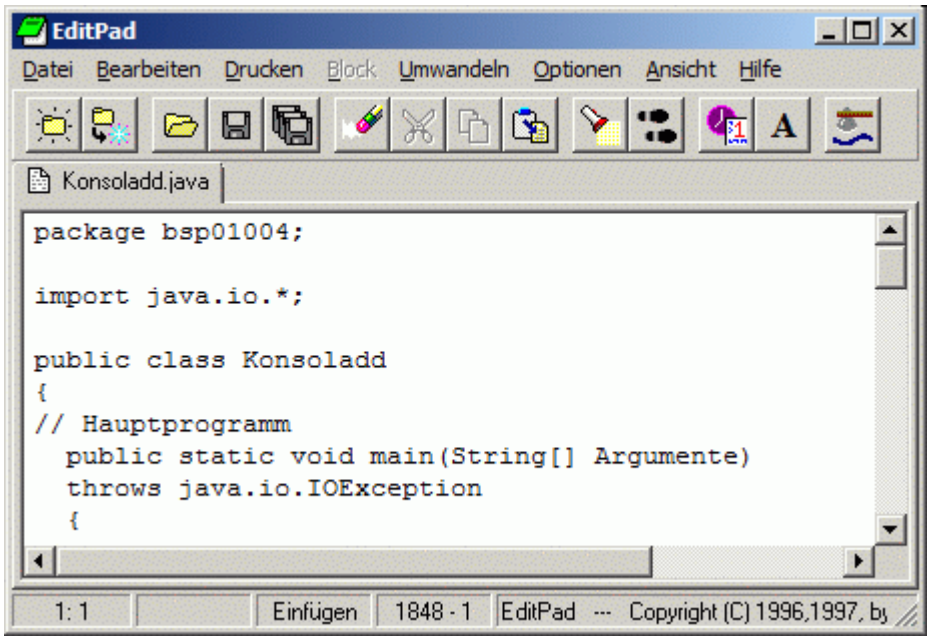
`notepad` *`laufwerk:\..pfadangabe..\dateiname`*`.java`

Dabei wird für *laufwerk* das aktuelle Laufwerk, für *..pfadangabe..* der Pfad, unter der die Datei zu finden ist, und für *dateiname* der Dateiname der Java Quelldatei ohne Zusatz eingesetzt. Es gelten also die üblichen Regeln für die Arbeit in einem Konsolbildschirm bzw. einer Konsolbox. Ist der aktuelle Pfad bereits gleich dem Pfad, in dem sich die Datei befindet, so lautet der Aufruf für die Datei Konsoladd.java zum Beispiel wie im folgenden Screenshot (Momentaufnahme einer Bildschirmanzeige):



Für die Arbeit in der Konsolbox sind also die üblichen Kenntnisse für den Umgang mit dem Betriebssytem ohne grafische Benutzeroberfläche erforderlich. Statt des Microsoft Windows Editors notepad können im übrigen **andere leistungsfähigere Editoren** verwendet werden. Wichtig für die Programmierung ist mindestens die **Angabe der aktuellen Zeilen- und Spaltenanzahl**, in der sich der Cursor befindet. Hilfreich ist ferner eine **Syntaxeinfärbung**, also die farbliche Hervorhebung von Sprachelementen der verwendeten Programmiersprache. Um das zweite Kriterium erfüllen zu können, müssen dem Editor natürlich die aktuellen Sprach-Regeln der verwendeten Programmiersprache bekannt sein.

Das erste Kriterium, nicht jedoch das zweite Kriterium erfüllt zum Beispiel der *Postcard-Ware* Editor **EditPad**, der im im FTP-Verzeichnis `eikelberg\javakurs\download\editpad` des Intranetservers PC132W.bsnw.fb4.fh-bochum.de gefunden werden kann. Eine Bildschirmanzeige von Konsoladd.java mit EditPad sieht wie folgt aus:



Für die nachfolgenden Erklärungen wird vorausgesetzt, dass der Prompt in der Eingabeaufforderung auf das Java-Basisverzeichnis (also das jpr-Verzeichnis) positioniert ist, d.h. dass die Pfadangabe im Prompt mit dem Namen des jpr-Verzeichnisses endet:

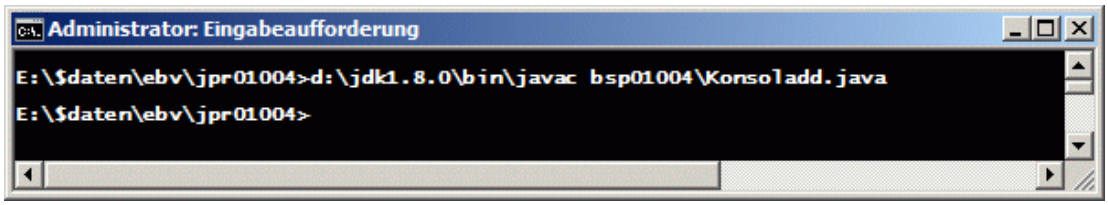
Nachdem die Java Quelldatei mit einem Editor erstellt oder abgeändert wurde, muß sie jetzt wie in Kapitel 1.2 erklärt **mit Hilfe eines Java-Compilers in Bytecode übersetzt** werden. Der Aufruf des JDK-Compilers lautet in vollständiger Form

`D:\jdk1.8.0\bin\javac` *`paketname\dateiname`*`.java`

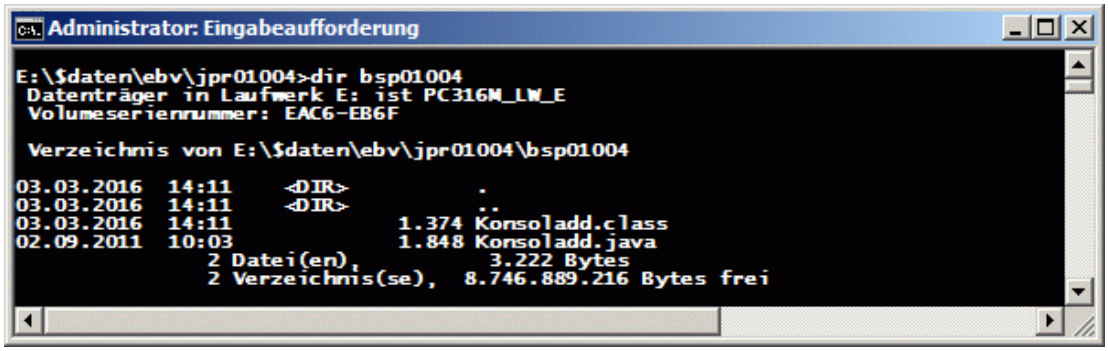
wobei `paketname` der in der **package-Anweisung** genannte Name des Programmpaketes ist und wobei der aktuelle Pfad in der Eingabeaufforderung auf das Basisverzeichnis eingestellt werden muß. Besteht der Paketname aus mehreren Namen `pname1.pname2. ... .pnamen`, so lautet der Aufruf des JDK-Compilers

`D:\jdk1.8.0\bin\javac` *`pname1\pname2\...\pnamen\dateiname`*`.java`

Das Verzeichnis `paketname` bzw. `pname1.pname2. ... .pnamen` wird im folgenden **Paketverzeichnis** genannt. Bezogen auf unser Beispiel erfolgt der Aufruf wie im folgenden Screenshot:



Wie man sieht hat der Java-Compiler die Eigenart, dass er bei einer fehlerfreien Übersetzung des Java-Programms keine Meldung ausgibt. Da beim Compile keine Fehler gefunden wurden, erstellt der Compiler eine Datei mit Bytecode mit dem Namen *`dateiname`*`.class`, die im Paketverzeichnis gespeichert wird. Bezogen auf unser Beispiel finden wir also im Paketverzeichnis bsp01004 zusätzlich zur java-Quelldatei eine Datei mit Namen `Konsoladd.class`:



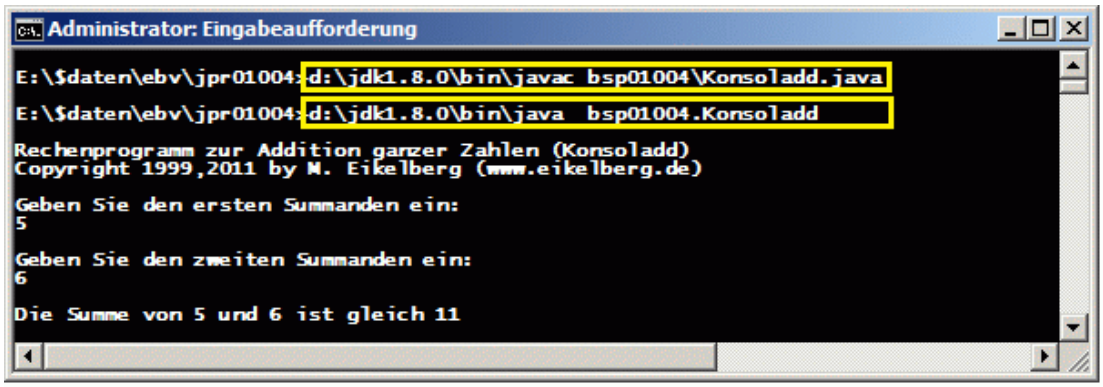
Ist die Java-Klasse erfolgreich compiliert, kann sie **zur Ausführung an die Java Virtual Machine (JVM)** übergeben werden. Dies geschieht durch Aufruf der JVM mit Hilfe des Befehls `java`. Die vollständige Syntax des Aufrufs der JVM lautet wie folgt:

```
D:\jdk1.8.0\bin\java paketname.dateiname
```

bzw. bei einem mehrstufigen Paketnamen

```
D:\jdk1.8.0\bin\java pname1.pname2. ... .pnamen.dateiname
```

Für das Beispiel bsp01004.Konsoladd geschieht der Compile sowie der anschließende Programmstart mit Hilfe der JVM wie folgt:

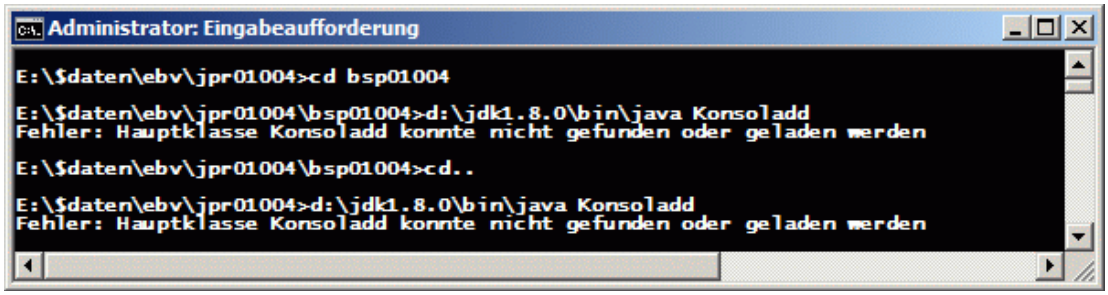


Man beachte folgende Unterschiede beim Aufruf von java und javac:

	Aufruf von javac.exe	**Aufruf von java.exe**
Aufruf	javac	java
Zeichen zwischen Paketname und Klassenname	Backslash	Punkt
Namenserweiterung	.java wird angegeben	.class wird nicht angegeben

<u>Hinweis:</u> Die Klassendatei wird von der JVM ausgehend vom Basisverzeichnis unter Verwendung der sich aus dem Paketnamen ergebenden Pfadangabe in einem entsprechenden Unterverzeichnis des aktuellen Verzeichnisses (nämlich dem Paketverzeichnis) gesucht! Ferner muss der beim Aufruf der JVM angegebene Paketname mit dem in der Java-Quelldatei in der package-Anweisung angegebenen Paketnamen übereinstimmen! Entsprechend muss der Prompt auf das Java Basisverzeichnis eingestellt sein.

Bei Nichtbeachtung dieser Tatsache ergeben sich folgende Fehler:



Man beachte in obigem Screenshot, dass nach Eingabe des letzten Befehls vom der JVM eine Fehlermeldung vom Typ "NoClassDefFoundError" ausgegeben wurde. Wenn man jedoch genau hinsieht, kann man erkennen, dass die JVM sehr wohl die Klasse Konsoladd.java gefunden hat, sich jedoch an der Package-Anweisung stört (wrong name: bsp01004/Konsoladd). Dies zeigt, dass die JVM stets vom Java-Basisverzeichnis aufgerufen werden muss, so dass der in der package-Anweisung genannte Pfad zwischen den Verzeichnis liegt, in dem die JVM aufgerufen wurde, sowie dem Verzeichnis, in dem die Klassendatei liegt.

1.3.1.2 Compile und Ausführung bei Verwendung von Klassen, die nicht zum JDK gehören

Sehr oft werden in Java-Programmen Klassen verwendet, die nicht zum Standardumfang des JDK gehören. In diesem Fall müssen folgende Punkte erfüllt sein:

- Die Klassen sollten in Form einer jar-Datei (eine Art Java-Archivdatei) bereitgestellt werden.
- Diese jar-Datei muss im Java-Basisverzeichnis gespeichert sein.
- Die jar-Datei muss beim Compile mit Hilfe der Compileroption **`-classpath`** bzw. **`-cp`** angegeben werden.
- Die jar-Datei muss bei der Ausführung mit Hilfe der Compileroption **`-classpath`** bzw. **`-cp`** angegeben werden.

Die Copileroptionen **`-classpath`** und **`-cp`** sind äquivalent, **`-cp`** ist lediglich die abgekürzte Schreibweise.

Beispiel für diese Situation ist das Projekt jpr01102. Dieses Projekt enthält die Quelldateien bsp01102\GuiAdd.java und

bsp01102\GuiAddFrame.java. In der Quelldatei bsp01102\GuiAddFrame.java befindet sich u.a. folgender Programmcode:

```
package bsp01102;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import tools01102.CopyrightPanel;
import tools01102.KommentarPanel;

public class GuiAddFrame extends JFrame
{
...
}
```

Dies bedeutet, dass die Bereitstellung der Klassen folgender Pakete im Quellcode von bsp01102\GuiAddFrame.java vorausgesetzt wird:

- java.awt
- java.awt.event
- javax.swing
- tools01102

Die ersten drei Pakete gehören zum Standardumfang des JDK und müssen nicht in jar-Dateien bereitgestellt werden. Dagegen gehört das Paket tools01102 nicht zum Standardumfang des JDK und muss daher in einer jar-Datei bereitgestellt werden. Die jar-Datei mit dem Paket tools01102 hat den Namen `tools01102.jar` und ist im Basisverzeichnis jpr01102 gespeichert.

Der Aufruf des Compilers lautet damit:

```
D:\jdk1.8.0\bin\javac -cp .;tools01102.jar bsp01102\GuiAddFrame.java
D:\jdk1.8.0\bin\javac -cp . bsp01102\GuiAdd.java
```

oder kürzer einfach

```
D:\jdk1.8.0\bin\javac -cp .;tools01102.jar bsp01102\*.java
```

Hinter der Compiler-Option -cp stehen also folgende Einträge, die durch ein Semikolon getrennt werden:

- . (ein Punkt) - steht für das Verzeichnis, von dem aus javac aufgerufen wurde, und deutet somit auf das Java-Basisverzeichnis
- tools01102.jar - Name der jar-Datei, die das Paket `tools01102` enthält. Diese Datei wird im Java-Basisverzeichnis gesucht.

In der Praxis tritt es oft auf, dass mehrere jar-Dateien genannt werden müssen. In diesem Fall werden alle erforderlichen jar-Dateien nach der classpath-Option genannt, jeweils durch ein Semikolon voneinander getrennt.

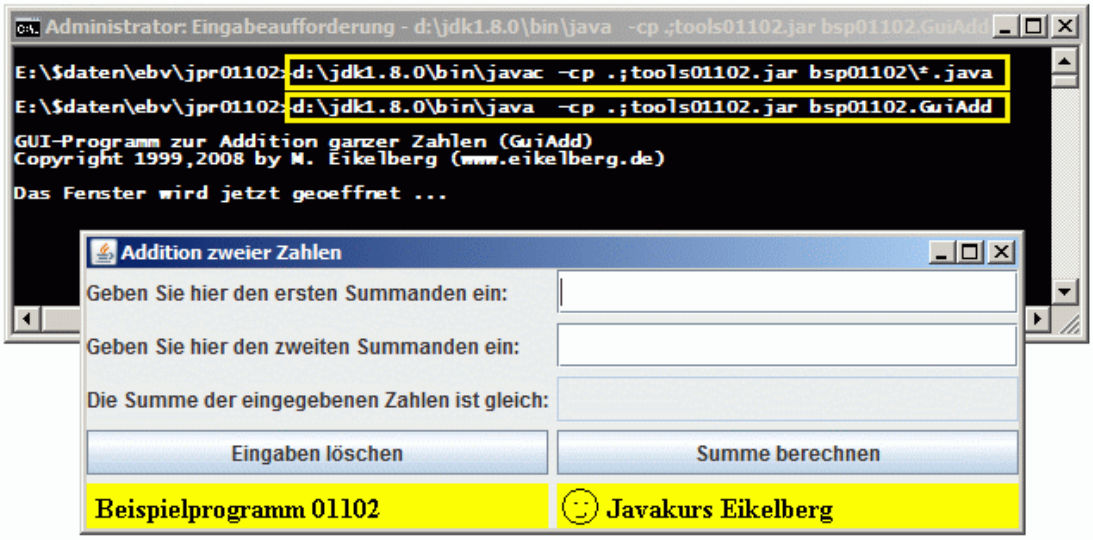
Auch beim Aufruf der JVM müssen die jar-Dateien wieder genannt werden. Beim Aufruf der JVM muss dabei die Klasse angegeben werden, die den mit

```
public static void main(String[] ...)
```

gekennzeichneten Abschnitt des Quellcode enthält. Im Fall des Beispielprojektes jpr01102 ist dies die Quelldatei GuiAdd.java. Entsprechend lautet der Aufruf der JVM wie folgt:

```
D:\jdk1.8.0\bin\java -cp .;tools01102.jar bsp01102.GuiAdd
```

Der nachfolgende Bildschirmausdruck zeigt die Wirkung der Aufrufe von java und javac:

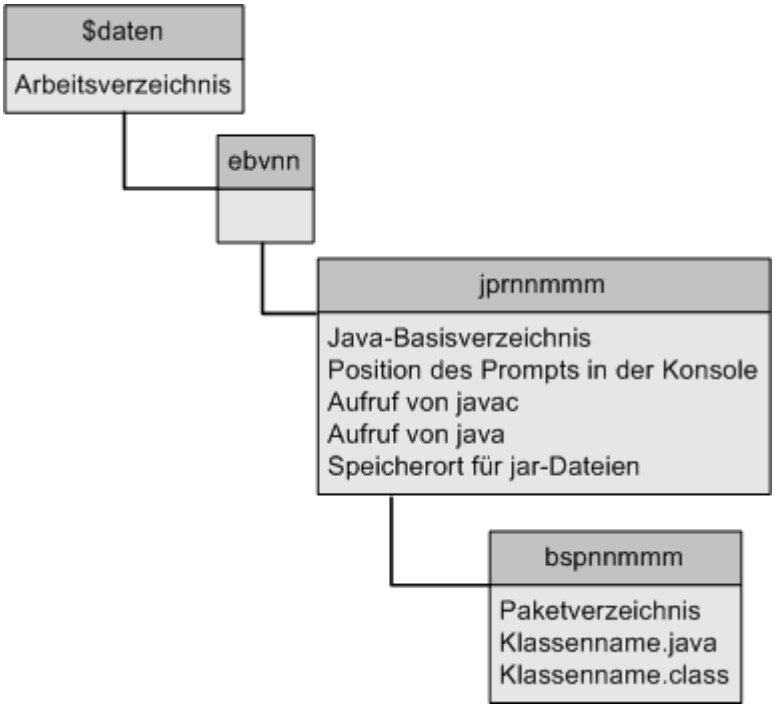


1.3.1.3 Regeln für den Aufruf von Compiler und Virtueller Maschine (JVM)

Bei der Anordnung der Java-Quelldatei sowie bei dem Aufruf von java und javac müssen also folgende Regeln unbedingt beachtet werden:

1. Für die Speicherung von Java-Programmen wählt man in der Regel jeweils ein separates Verzeichnis. Dieses Verzeichnis wird **Java-Basisverzeichnis** bzw. kurz **Basisverzeichnis** genannt.
2. Alle Pfadangaben im Zusammenhang mit der Java-Programmierung werden **relativ zum Basisverzeichnis** - also mit dem Basisverzeichnis beginnend - ermittelt.
3. Der in der **package-Anweisung** angegebene Paketname bestimmt das Unterverzeichnis des Basisverzeichnisses, in dem die jeweilige Java-Quelldatei bzw. die zugehörige Klassendatei gespeichert wird.
4. **Beim Aufruf von Compiler (javac) und JVM (java) muss der in der Konsolbox eingestellte Pfad im Basisverzeichnis positioniert werden.** Ferner müssen beim Aufruf von Compiler und JVM die Paketnamen geeignet angegeben werden.
5. In den Java-Klassen verwendete Klassen, die nicht selber programmiert wurden und die nicht zum Standardumfang des JDK gehören, müssen in Form von jar-Dateien bereitgestellt werden. Diese **jar-Dateien müssen im Java-Basisverzeichnis gespeichert werden**. Beim Compile sowie beim Programmaufruf müssen die jar-Dateien mit Hilfe der **Option -classpath** bzw. **Option -cp** angegeben werden.

Die obigen Regeln werden in folgender Grafik verdeutlicht. Dabei wird das Java-Basisverzeichnis in einem Verzeichnis mit Namen `ebvnn` angeordnet. Dieses Verzeichnis hat hier noch keine Bedeutung, wird jedoch für die in Kapitel 1.3.2 eingeführte Entwicklungsumgebung Eclipse benötigt. Das Verzeichnis `ebvnn` befindet sich seinerseits in einem Verzeichnis `$daten`. Dieses Verzeichnis stellt den Arbeitsbereich auf dem jeweils verwendeten System dar.



Das Beispielprogramm bsp01015.Konsoladd verdeutlicht noch einmal die Situation. Dieses Programm ist dem Beispielprogramm bsp01004.Konsoladd sehr ähnlich, verwendet jedoch eine Klasse des Paketes de.eikelberg.iotools. Dieses Paket wird im Java-Basisverzeichnis jpr01015 in einer jar-Datei mit Namen de_eikelberg_iotools.jar bereitgestellt (vgl. Kapitel 2.1). Daher geschieht der Compile sowie der Programmaufruf wie folgt:

```
Administrator: Eingabeaufforderung

E:\$daten\ebv\jpr01015>dir
 Datenträger in Laufwerk E: ist PC316M_LW_E
 Volumeseriennummer: EAC6-EB6F

 Verzeichnis von E:\$daten\ebv\jpr01015

03.03.2016  14:23    <DIR>          .
03.03.2016  14:23    <DIR>          ..
03.03.2016  14:20    <DIR>          bsp01015
24.06.2015  20:39            58.062 de_eikelberg_iotools.jar
               1 Datei(en),         58.062 Bytes
               3 Verzeichnis(se),  8.746.741.760 Bytes frei

E:\$daten\ebv\jpr01015>dir bsp01015
 Datenträger in Laufwerk E: ist PC316M_LW_E
 Volumeseriennummer: EAC6-EB6F

 Verzeichnis von E:\$daten\ebv\jpr01015\bsp01015

03.03.2016  14:20    <DIR>          .
03.03.2016  14:20    <DIR>          ..
03.03.2016  14:22             1.517 Konsoladd.class
03.09.2006  14:50             1.817 Konsoladd.java
               2 Datei(en),          3.334 Bytes
               2 Verzeichnis(se),  8.746.741.760 Bytes frei

E:\$daten\ebv\jpr01015>d:\jdk1.8.0\bin\javac -cp .;de_eikelberg_iotools.jar bsp01015\Konsoladd.java

E:\$daten\ebv\jpr01015>d:\jdk1.8.0\bin\java  -cp .;de_eikelberg_iotools.jar bsp01015.Konsoladd
--------------------------------------------------------------------------
    Rechenprogramm zur Addition ganzer Zahlen (Konsoladd)
    Copyright 2006 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------

Geben Sie den ersten Summanden ein:
5

Geben Sie den zweiten Summanden ein:
6

Die Summe von 5 und 6 ist gleich 11

Programm beenden mit RETURN !
```

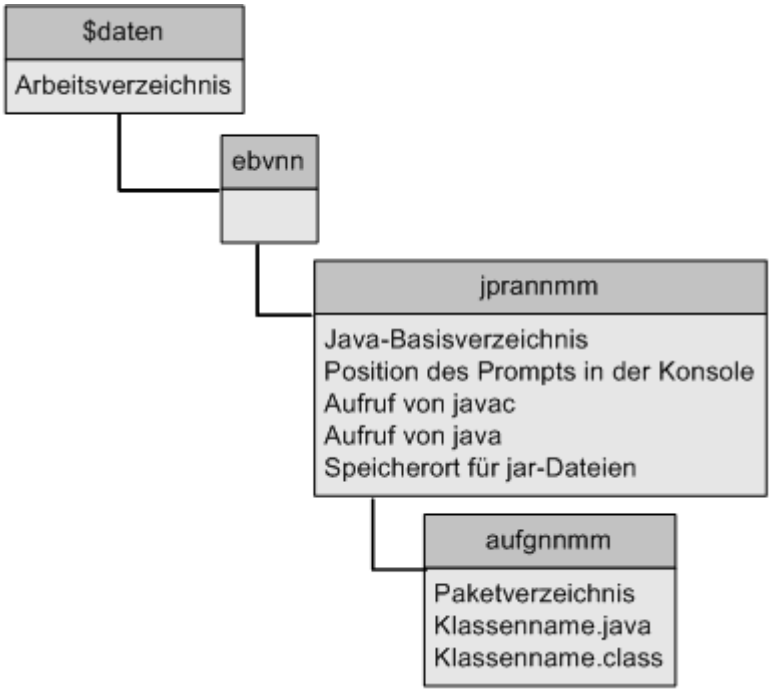
Compile

Programmstart

Hinweis: Für das Praktikum gelten folgende Namenskonventionen:

1. Das Basisverzeichnis zur Lösung der Übungsaufgabe *mm* zu Kapitel *nn* sollte stets `jpr`*nnmm* heißen.
2. Das Paketverzeichnis für die Java-Klassen der Lösung einer Übungsaufgabe mit Nummer *mm* und zu Kapitel *nn* sollte in der Regel mit dem Namen `aufg`*nnmm* benannt werden.

Für das Praktikum ergibt sich somit zunächst folgende Situation - dabei wird wieder in einem Verzeichnis `$daten` gearbeitet und es wird wieder in diesem Arbeitsbereich zunächst ein Unterordner `ebvnn` angelegt, in dem anschließend das Java-Basisverzeichnis angeordnet wird.



1.3.2 Eclipse

Das Programm **Eclipse** (Version 4.5.1) ist eine integrierte Entwicklungsumgebung für die Programmentwicklung mit dem JDK (standardmäßig JDK 1.8.0). Eclipse verwendet eine vom Benutzer einstellbare Version des JDK und verfügt über eine graphische Benutzeroberfläche. In der Grundversion bietet Eclipse neben vielen weiteren Funktionen bereits folgende Funktionalität:

- Graphische Benutzeroberfläche
- Verwaltung aller Dateien zu einem Software-Projekt in einer Projektumgebung.
- Hilfe bei der Erstellung der Dateistruktur zu einem Software-Projekt.
- Automatische Erstellung von Java-Quellcode für die Grundstruktur der verschiedenen Programmtypen.
- In die Oberfläche integrierter Editor.
- Zugriff auf Java Compiler und JVM über die Oberfläche von Eclipse.
- Markierung fehlerhafter Programmzeilen direkt im Editor bei Compile-Fehlern.
- Syntaxeinfärbung

Diese Funktionen erleichtern ganz wesentlich die Entwicklung und Weitergabe von Java Programmen. Sie sind typisch für integrierte Entwicklungsumgebungen und sehr hilfreich bei der Programmierung. Daher wird für die Durchführung dieses Java-Kurses die Entwicklungsumgebung Eclipse eingesetzt.

Eclipse ist ein Open Souce Programm und damit nicht kostenpflichtig. Es wird gemäß der **Eclipse Public License** lizensiert. Eclipse steht in der Tradition bewährter Entwicklungsumgebungen und wird von der Eclipse Organisation hergestellt und vertrieben. An der Eclipse-Organisation beteiligen sich Firmen wie Sun Microsystems, IBM und CodeGear/Borland. Da Eclipse mit verschiedenen Java-Versionen kombiniert werden kann und in vielfältiger Weise erweitert werden kann, stellt der Autor für die Arbeit mit diesem Java-Kurs speziell konfigurierte Eclipse-Versionen bereit. Die Verteilung dieser Versionen erfolgt über DVDs mit dem Titel "Software für den Unterricht Eikelberg". Diese Anleitung erkärt die Verwendung der Version "September 2016" dieser DVDs mit der Eclipse-Version 4.5.1 und der JDK-Version 1.8.0.

Hinweis: Beachten Sie die **zu anderen Entwicklungsumgebungen bereitgestellten Anleitungen im Anhang**!

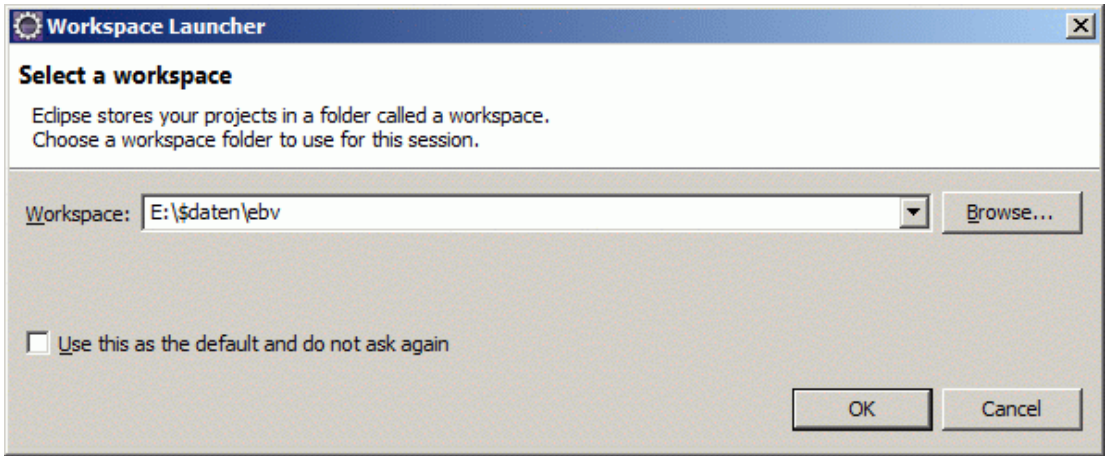
Auf der DVD mit der Software für den Unterricht Eikelberg Stand "September 2016" befinden sich 32 bit Programmversionen. Diese sind lauffähig unter Windows 7 bzw. 10 (32bit sowie 64bit).

1. **Start von Eclipse 4.5.1.**

   Alle Eclipse 4.5.1 Projekte müssen in einem gemeinsamen Verzeichnis angelegt werden:

   - Dieses Verzeichnis wird im Weiteren **Eclipse Basisverzeichnis** genannt.
   - Für dieses Verzeichnis wird in dieser Anleitung der Verzeichnisname "ebv" (**E**clipse **B**asisverzeichnis) gewählt. ***Man beachte, dass für die Beispiele und Musterlösungen dieses Kurses jedoch kapitelweise Eclipse Basisverzeichnisse mit dem Namen ebvnn angelegt wurden!***

   Im Eclipse Basisverzeichnis wird von Eclipse 4.5.1 projektübergreifend ein Verzeichnis mit den Namen ".metadata" angelegt. Ferner wird in diesem Verzeichnis für jedes Programmierprojekt ein Unterverzeichnis angelegt, das die Funktion des **Java-Basisverzeichnisses** übernimmt. Beim Start von Eclipse 4.5.1 wird standardmäßig zunächst der Name des Eclipse Basisverzeichnisses abgefragt. Dieser muss in folgendem Fenster eingetragen werden:



   Es startet Eclipse Version 4.5.1, auch Version "Mars.1" genannt.

   Schließen sie ggf. nach Abschluss des Programmstarts von Eclipse 4.5.1 die Registerkarte "Welcome".

   Hinweis: Falls beim Start von Eclipse das obige Fenster nicht erscheint, können Sie wie folgt in Eclipse konfigurieren, dass dieses Fenster beim Start wieder erscheint: Wählen Sie im Menü von Eclipse Window / Preferences, erweitern Sie im Fenster Preferences

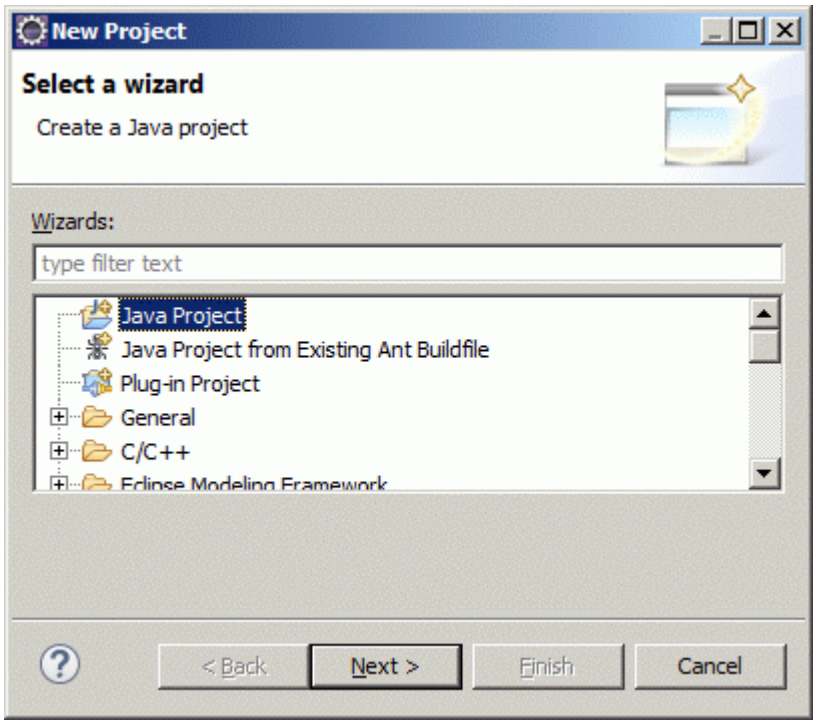
den Zweig General sowie den Zweig "Startup and Shudown" und markieren Sie das Blatt "Workspaces". Aktivieren Sie anschließend im rechten Bereich des Fensters die Option "Prompt for workspace on startup".

Mit dem beschriebenen Ablauf können Sie einen neuen Workspace anlegen oder mit einem bereits existierenden Workspace weiter arbeiten. Wenn Sie einen neuen Workspace angelegt haben, sollten Sie folgende Konfigurationseinstellungen vornehmen:

- Öffnen Sie im Eclipse-Menü Window / Preferences und erweitern Sie den Zweig Java / Installed JREs. Es sollte eine installierte Version des JRE angezeigt werden mit der Pfadangabe ("Location")
  `C:\eclipse-eikelberg-5` bzw. `D:\eclipse-eikelberg-5`
  je nachdem, in welches Laufwerk Ihre Eclipse-Version installiert wurde. Korrigieren Sie ggf. den vorhandenen Eintrag, in dem Sie den vorhandenen Eintrag markieren, auf den Button "Edit" klicken und die Eigenschaften des Eintrags geeignet ändern.
- Öffnen Sie im Eclipse-Menü Window / Preferences und erweitern Sie den Zweig General / Editors / Text Editors / Spelling und deaktivieren Sie die Option "enable spell checking". Hiermit deaktivieren Sie die (englische) Rechtschreibprüfung, die für Programme mit deutschen Kommentaren und deutschen Programmausgaben natürlich keine sinnvollen Ergebnisse ergibt.
- Öffnen Sie im Eclipse-Menü Window / Preferences und erweitern Sie den Zweig Java / Code Style / Formatter. Klicken Sie im rechten Bereich des Fensters auf den Button "Edit" und öffnen Sie so das Fenster "Profile ...". Erstellen Sie wie folgt ein Profil mit individuell eingestellten Formatierungsregeln für den Quellcode:
  - Stellen Sie im Fenster "Profile ..." in der Registerkarte "Indentation" die Tab policy um auf "Spaces Only" und tragen Sie in den Feldern "Indentation size" und "Tab size" jeweils den Wert 2 ein. Tragen Sie im Feld "Profile name" den Namen "eclipse-javakurs" ein und speichern Sie die gewählten Einstellungen mit Klick auf den Button "Apply".
  - Wechseln Sie nun in die Registerkarte "Braces". Ändern Sie die "Brace positions" für alle angebotenen Möglichkeiten um von "Same line" auf "Next line" und bestätigen Sie diese Änderungen mit Klick auf den Button "Apply".
  - Wechseln Sie in die Registerkarte "Comments" und wählen Sie folgende Optionen aus: "Perserve white space between code and line comments" / "Never indent line comments on first column" / "Never indent block comments on first column" / "Never join lines"
  - Wechseln Sie in die Registerkarte "Line Wrapping" und wählen Sie die Option "Never join already wrapped lines" aus. Markieren Sie im Zweig "Function Calls" das Blatt "Arguments" und stellen Sie die "Line wrapping policy" "Wrap all elements, every element on a new line" ein. Konfigurieren Sie die "Indentation policy" "Indent by one".
  - Wechseln Sie in die Registerkarte "Control Statements". Deaktivieren Sie die Option "Keep else if in one line". Aktivieren Sie im Feld "General" alle Optionen "Insert new line...".
  - Übernehmen Sie die Einstellungen mit Klick auf den Button "Apply". Schließen Sie danach das Fenster "Profile 'eclipse-javakurs'" mit Klick auf den Button OK.
- Schließen Sie anschließend das Fenster "Properties" ebenfalls durch Klick auf den Button OK.

2. **Anlegen eines Programmierprojektes**

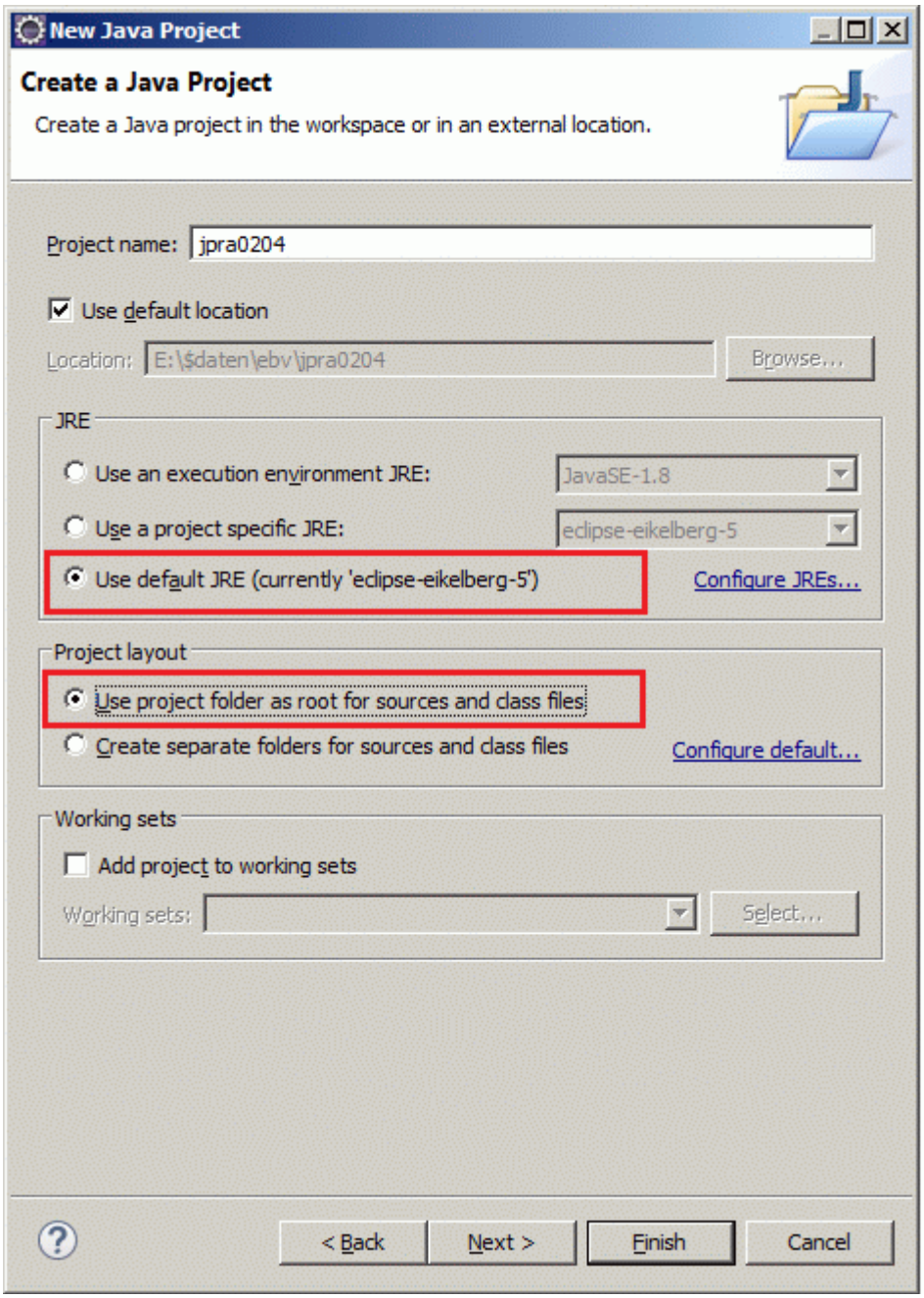
Wählen Sie im "File"-Menü die Auswahl "New" / "Project". Wählen Sie anschließend im Fenster "New Project" den Assistenten zum Anlegen eines Java-Projektes aus:



Wechseln Sie mit Klick auf den Button "Next" in das Fenster "New Java-Project". Tragen Sie dort zunächst den Namen des Basisverzeichnisses ein - dieser Name wird für die Aufgabe 4 in Kapitel 2 wie folgt gebildet:

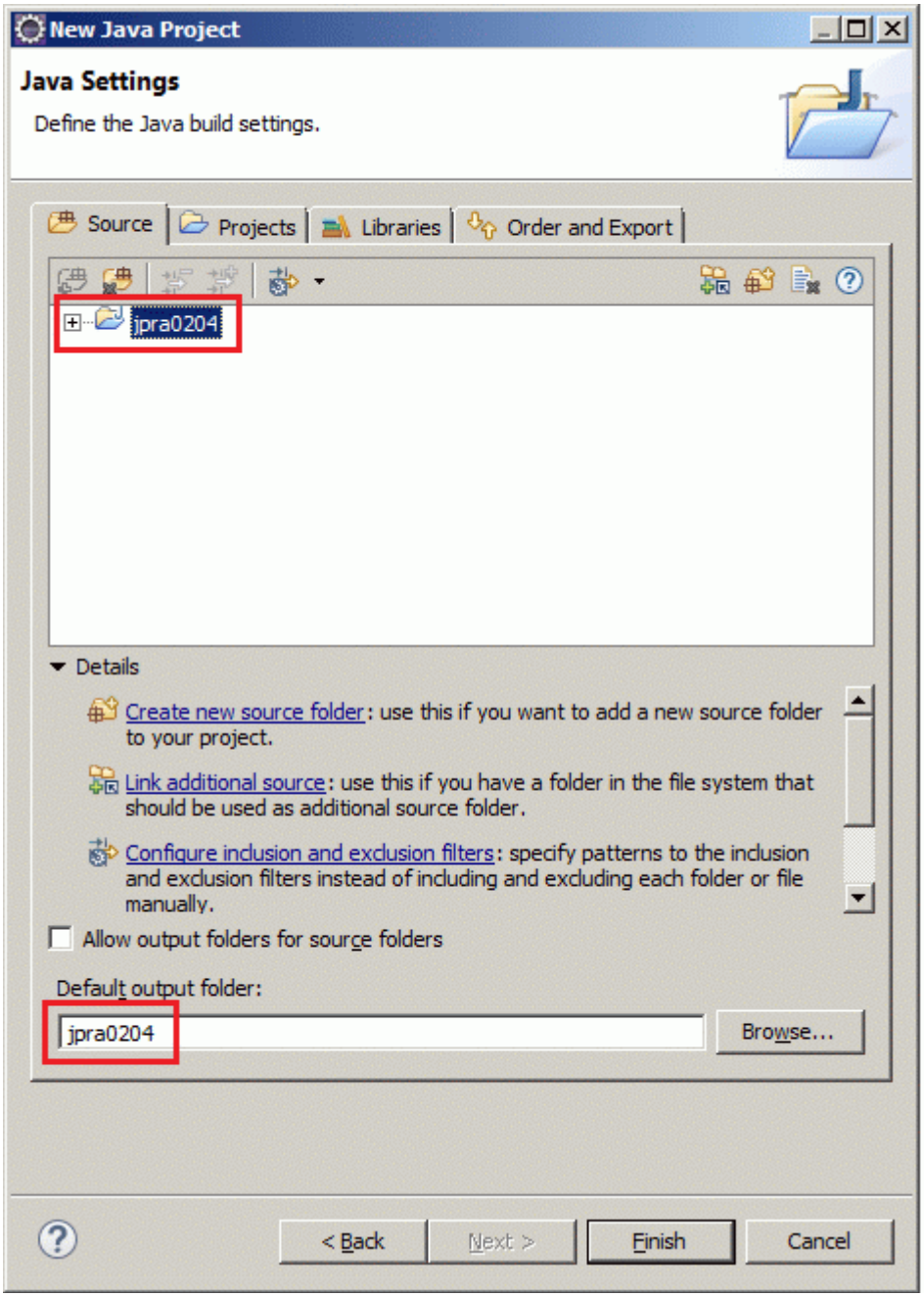
Feld	Eintrag für eine Lösung der Programmieraufgabe *mm* in Kapitel *nn*	Beispiel
Projektname	jpra*nnmm*	jpra0204
Verzeichnisname	volle Pfadangabe für das Java-Basisverzeichnis	e:\$daten\ebv\jpra0204

Im Fenster "New Java-Project" werden dann die folgenden Projekteinstellungen angezeigt:



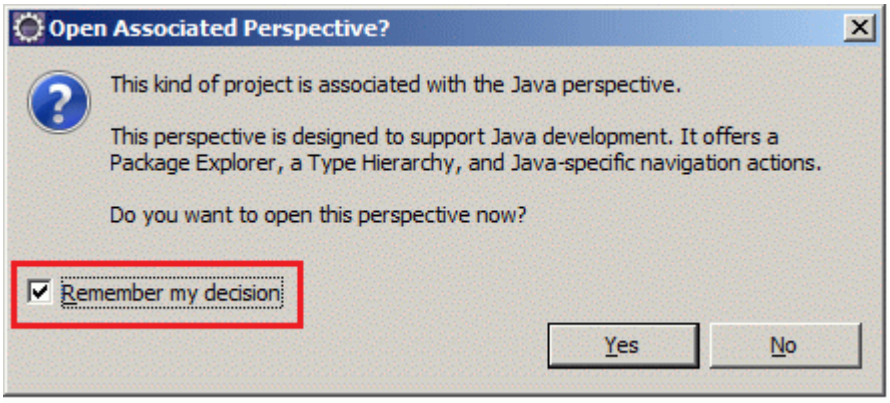
Achten Sie darauf, dass die Option "Use project folder as root for sources and class files" eingestellt ist. Ferner sollte Option "Use default JRE ..." ausgewählt werden. Wählen Sie ggf. diese Optionen aus!

Bestätigen Sie diese Konfiguration mit "Next" und wechseln Sie so in das Fenster "Java Settings". In diesem Fenster sollten folgende Einstellungen angezeigt werden:

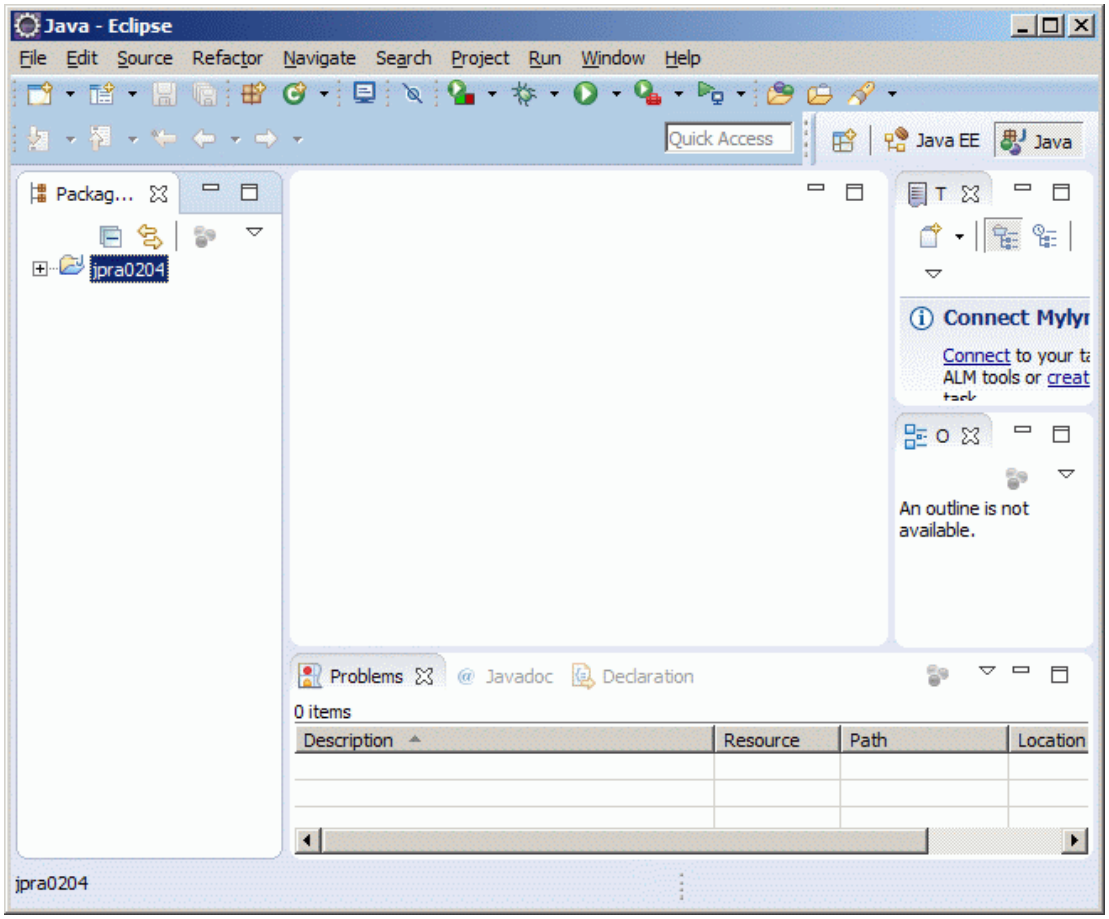


Achten Sie in diesem Fenster darauf, dass in diesem Fenster als souce folder sowie als default output folder jeweils der Ordner für Ihr Projekt angezeigt wird. Ist dies nicht der Fall, müssen Sie mit dem Button "Back" in das vorhergehende Fenster zurück wechseln und die Einstellung der Option "Use project folder as root for sources and class files" vornehmen.

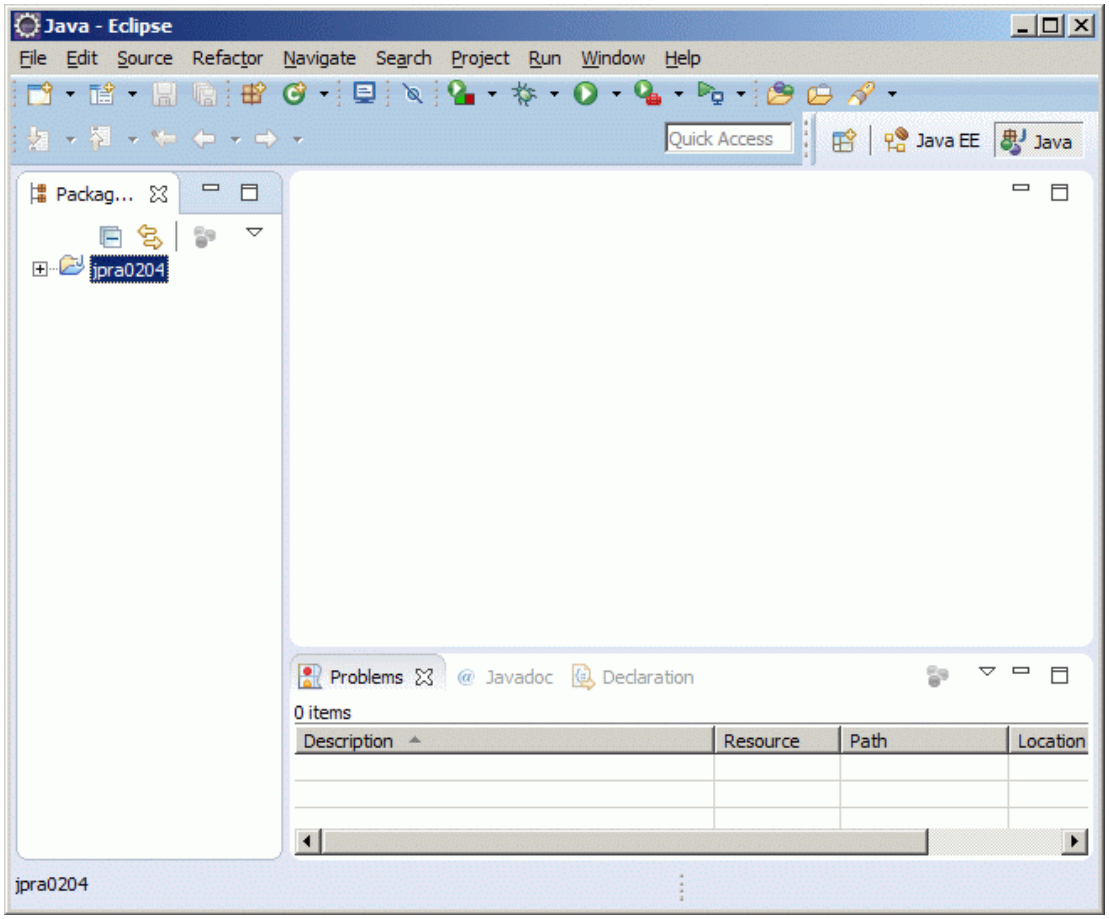
Mit den jetzt vorhandenen Einstellungen wird erreicht, dass Ihre Projektkonfiguration in Bezug auf die Ordnerstruktur den Standardeinstellungen des JDK entspricht. Beenden Sie den Assistenten mit Klick auf den Button "Finish". Gegebenenfalls erscheint die folgende Meldung, die Sie mit "Yes" bestätigen sollten und für die Sie die Option "remember my decision" einstellen können:



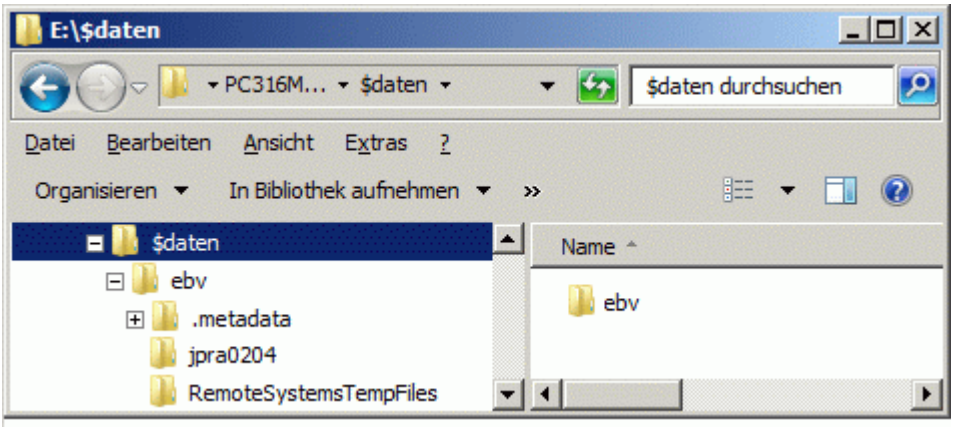
Ausgehend von den oben eingegebenen Informationen legt Eclipse 4.5.1 jetzt das Projekt an und zeigt dieses in einer Java-Perspektive an:



Eine Eclipse Perspektive ist eine Anordnung von Fensterelementen in Eclipse, die für einen bestimmten Zweck konfiguriert wurde. Eine Java-Perspektive ist also eine Fensterkonfiguration, die bei der Java-Programmierung verwendet wird. Eclipse kennt viele weitere Perspektiven, z. Bsp. die C/C++ Perspektive und die Debug Perspektive. In der vorliegenden Perspektive können die Registerkarten "Task List" und "Outline" geschlossen werden, so dass sich der Anzeigebereich für den Quellcode vergrößert:

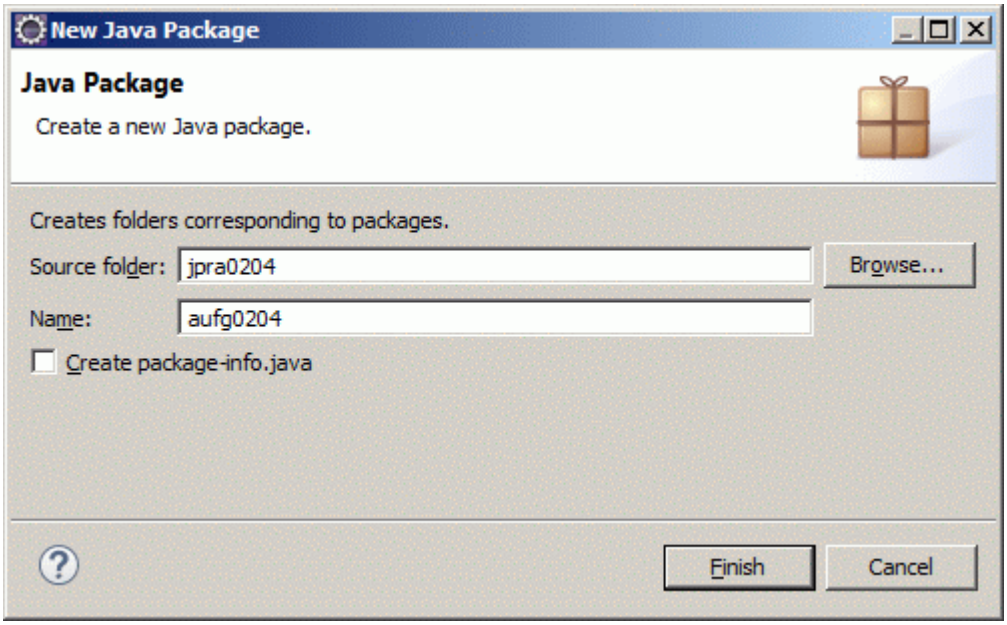


Nach Anlegen des Projektes ist im Eclipse Basisverzeichnis folgende Struktur entstanden:

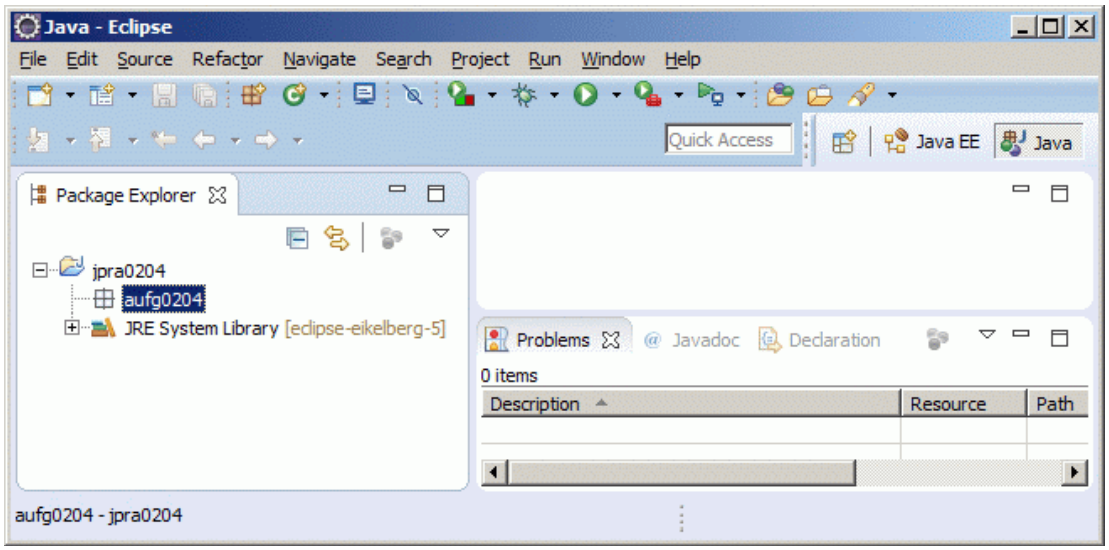


3. **Anlegen einer Quelldatei für eine Java-Klasse mit main-Methode (Hauptprogramm)**

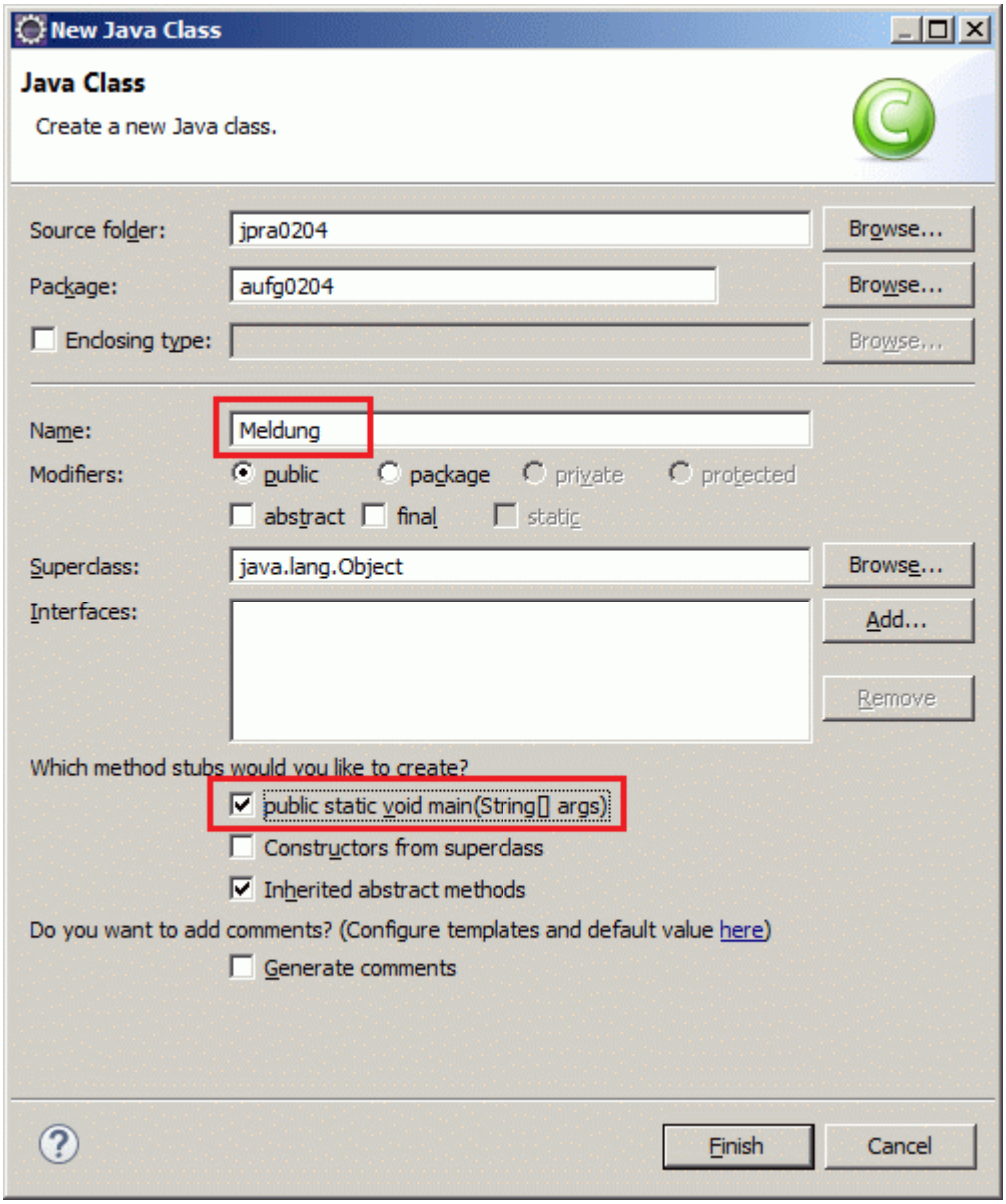
Markieren Sie im linken Bereich des Fensters im Package Explorer das Projekt jpra0204 und wählen Sie anschließend im "File"-Menü die Auswahl "New" / "Package". Es öffnet sich das Fenster "New Java-Package". Geben Sie in diesem Fenster im Feld "Name" den Paket-Namen Ihrer Java-Anwendung ein und klicken Sie auf den Button "Finish":



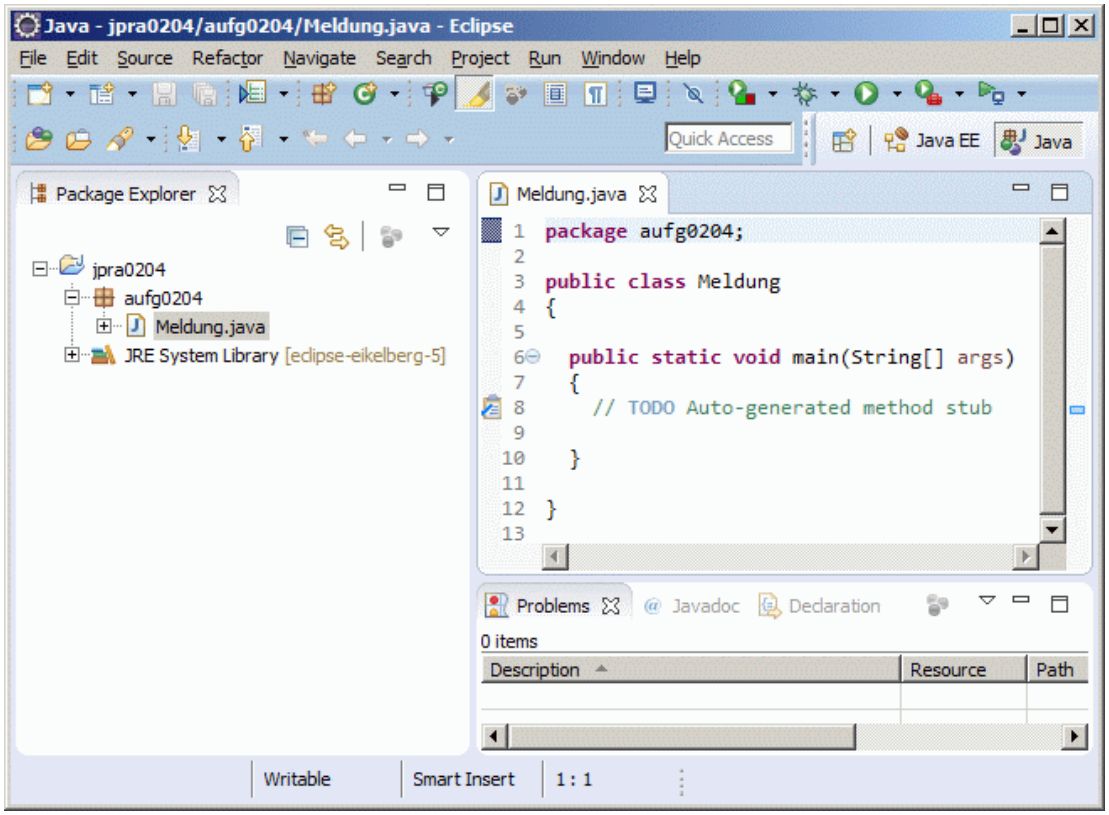
Im Eclipse wird das neu angelegte Paket wie folgt angezeigt:



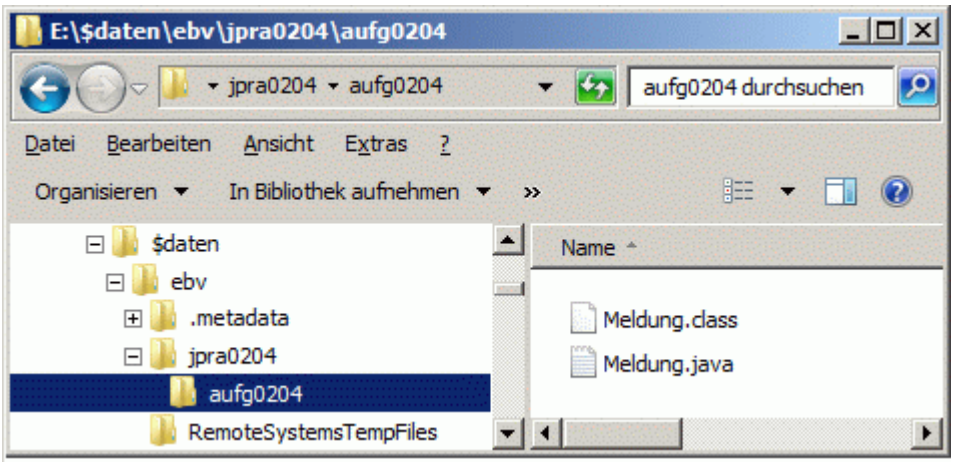
Markieren Sie im linken Bereich des Fensters im Package Explorer das Paket aufg0204 und wählen Sie anschließend im "File"-Menü die Auswahl "New" / "Class". Es öffnet sich das Fenster "New Java Class". Kontrollieren Sie, ob der Name des Java-Basisverzeichnisses (jpra0204) sowie der Paketname (aufg0204) korrekt eingetragen sind. Geben Sie den Namen der Klasse (Meldung) im Feld "Name" ein und wählen Sie aus, dass eine main-Methode erzeugt werden soll:



Bestätigen Sie Ihre Eingaben durch Klick auf den Button "Finish". Daraufhin wird die angeforderte Klasse erzeugt und in der Java-Perspektive angezeigt:



Im Eclipse Basisverzeichnis ist jetzt folgende Dateistruktur vorhanden:

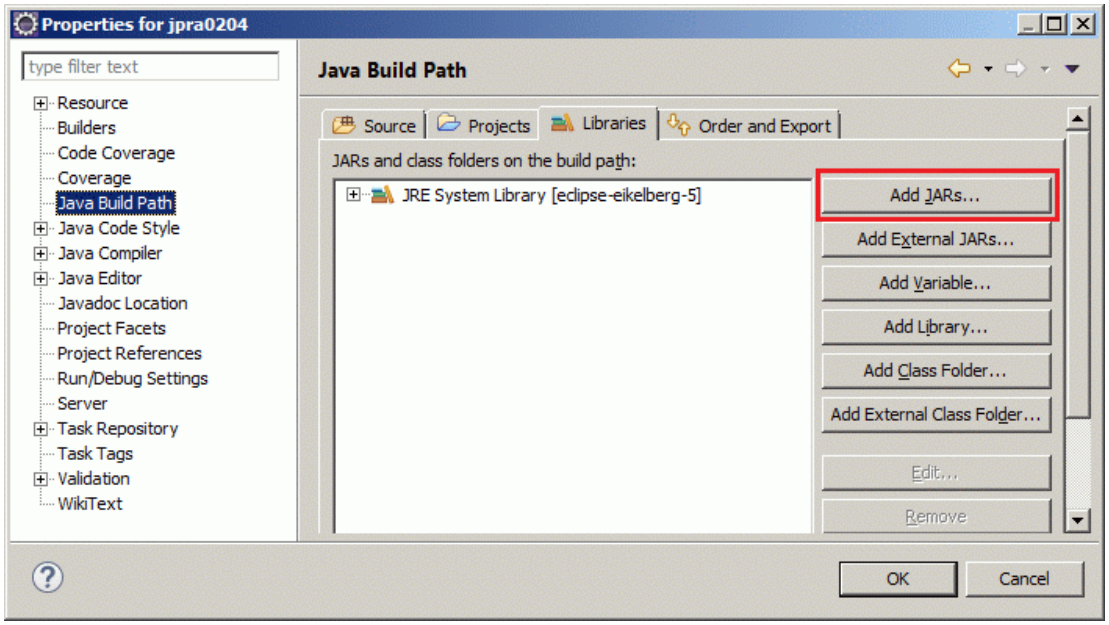


4. **Einfügen des Paketes de.eikelberg.iotools in das Projekt**

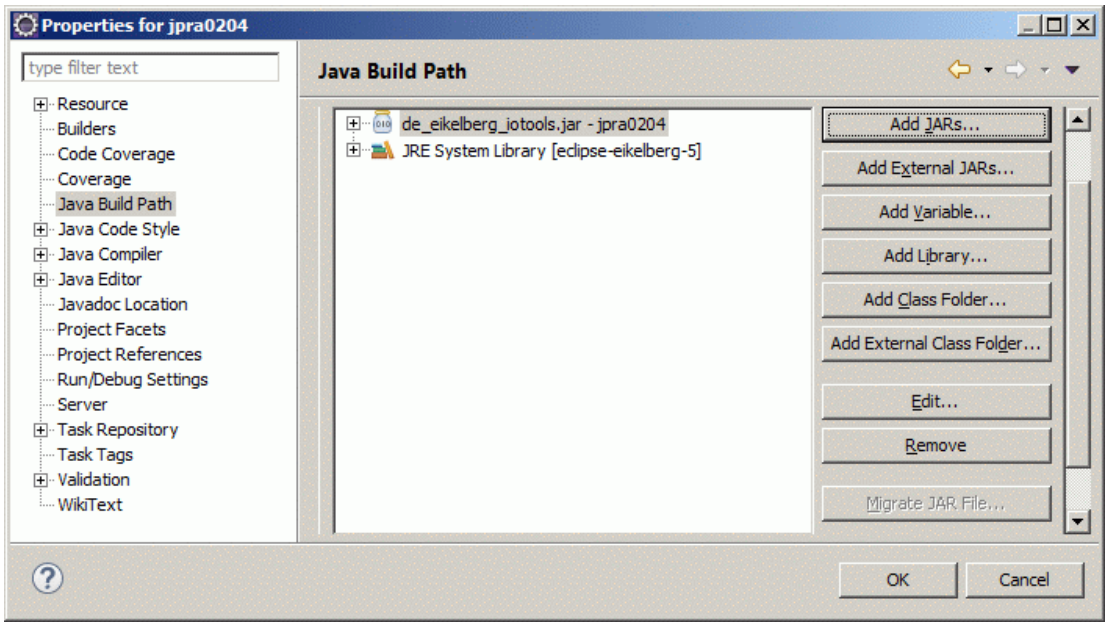
Für die Arbeit mit Eclipse 4.5.1 muss das Paket de.eikelberg.iotools als sogenannte JAR-Datei in das Java Basisverzeichnis eingefügt werden. Das Paket de.eikelberg.iotools liegt als JAR-Datei mit dem Namen de_eikelberg_iotools.jar vor. Laden Sie das **ZIP-Archiv mit der Datei de_eikelberg_iotools.jar** herunter und speichern Sie das Archiv im Java-Basisverzeichnis (also im Beispiel im Verzeichnis E:\$daten\ebv\jpra0204). *Die Datei de_eikelberg_iotools.jar darf nicht weiter bearbeitet bzw. muss auch nicht entpackt werden.*

> Da die jar-Datei de_eikelberg_iotools.jar nicht von Eclipse in das Java-Basisverzeichnis eingefügt wurde, ist diese Einfügung Eclipse nicht bekannt. Sie müssen daher an dieser Stelle das Java-Projekt im Package Explorer von Eclipse markieren, in das Sie die jar-Datei eingefügt haben. Öffnen Sie anschließend mit der rechten Maustaste das Kontextmenü für das Projekt und wählen Sie im Kontextmenü den Eintrag Refresh (alternativ Taste F5). Daraufhin sollte die jar-Datei im Package Explorer von Eclipse im entsprechenden Projekt angezeigt werden.

Markieren Sie in der Registerkarte "Package-Explorer" der Java-Perspektive das Projekt jpra0204 und wählen Sie im "Project"-Menü den Menüpunkt "Properties". Es öffnet sich das Fenster "Properties for jpra0204". Markieren Sie im linken Bereich des Fensters den Eintrag "Java Build Path" und wechseln Sie im rechten Bereich des Fensters in die Registerkarte "Libraries":



Klicken Sie auf den Button "Add JARs" und fügen Sie die im Java-Basisverzeichnis gespeicherte jar-Datei de_eikelberg_iotools.jar hinzu. Verwenden Sie *nicht* den Button "Add External JARs"! Die jar-Datei wird daraufhin in der Registerkarte "Libraries" angezeigt:



Bestätigen Sie die Eingaben mit OK und wechseln Sie so wieder in die Java-Perspektive zurück.

Tipp: Das Kennzeichnen einer jar-Datei als Bibliothek kann auch über den Eclipse Package Explorer, Kontextmenü zur im Package Explorer angezeigten Jar-Datei, Menüauswahl Build Path/ Add to Build Path erfolgen.

5. **Codierung des Java-Programms**

Codieren Sie jetzt im mittleren Bereich der Java-Perspektive Ihr Java-Programm und vergessen Sie nicht, Ihre Quelldatei regelmäßig zu sichern. Im vorliegenden Beispiel - Lösung von Aufgabe 4 aus Kapitel 2 des Java-Kurses - entsteht nach Einfügen des in der Aufgabenstellung genannten Java-Quellcodes sowie nach geeigneter Änderung der Anordnung der Klammern folgendes Java-Programm:

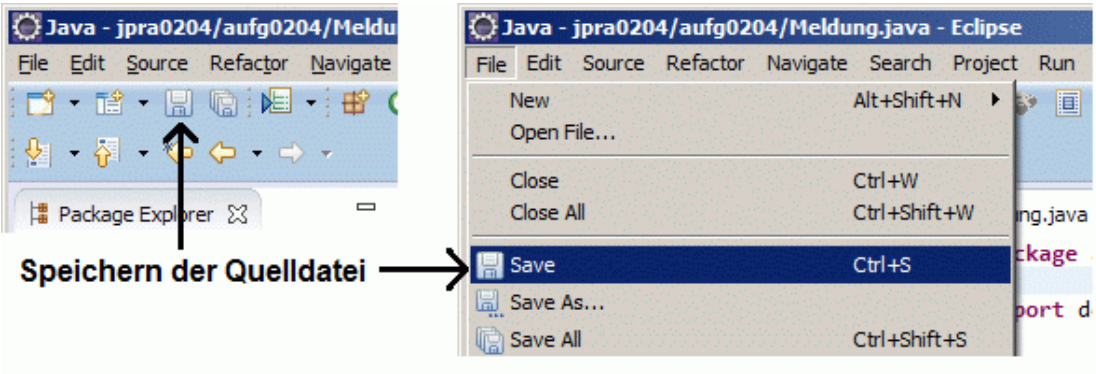
```
Meldung.java
1  package aufg0204;
2
3  import de.eikelberg.iotools.*;
4
5  public class Meldung
6  {
7
8    public static void main(String[] args)
9    {
10     ScreenOut.kPrintln();
11     ScreenOut.kPrintln("Hier spricht das Programm aufg0204.Meldung!");
12   }
13 }
```

Hinweis: Wenn Sie später Ihr Programm in der Eingabeaufforderung von Windows mit der Java Virtuellen Maschine (JVM) ausführen wollen, müssen Sie beim Aufruf der JVM wie folgt als Suchpfad das Java-Basisverzeichnis sowie den Namen der JAR-Datei zusätzlich mit angegeben:

```
Administrator: Eingabeaufforderung
E:\$daten\ebv\jpra0204>d:\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar aufg0204.Meldung
Hier spricht das Programm aufg0204.Meldung!
E:\$daten\ebv\jpra0204>
```

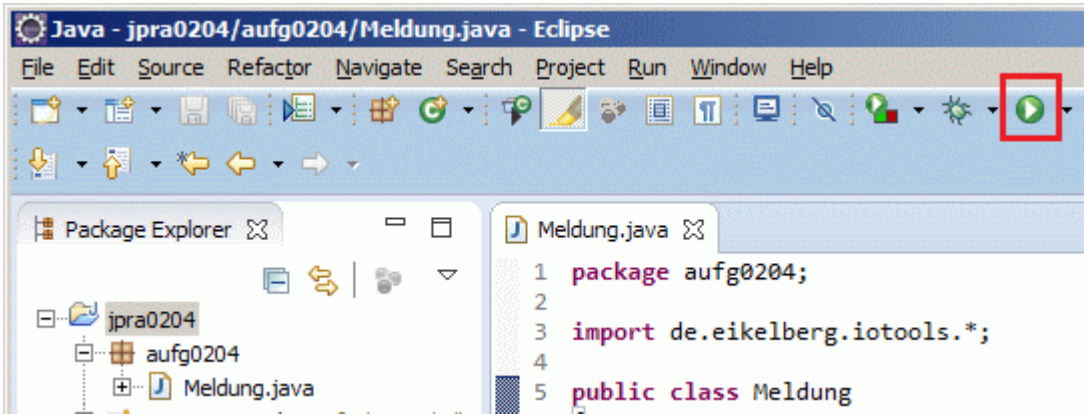
6. **Compile**

Eine Besonderheit von Eclipse ist, dass der Compiler nicht direkt gestartet werden kann. Vielmehr wird der Compiler stets gestartet, wenn die Quelldatei gespeichert wird (Strg+S bzw. Button "Save" bzw. Menüauswahl "File / Save":



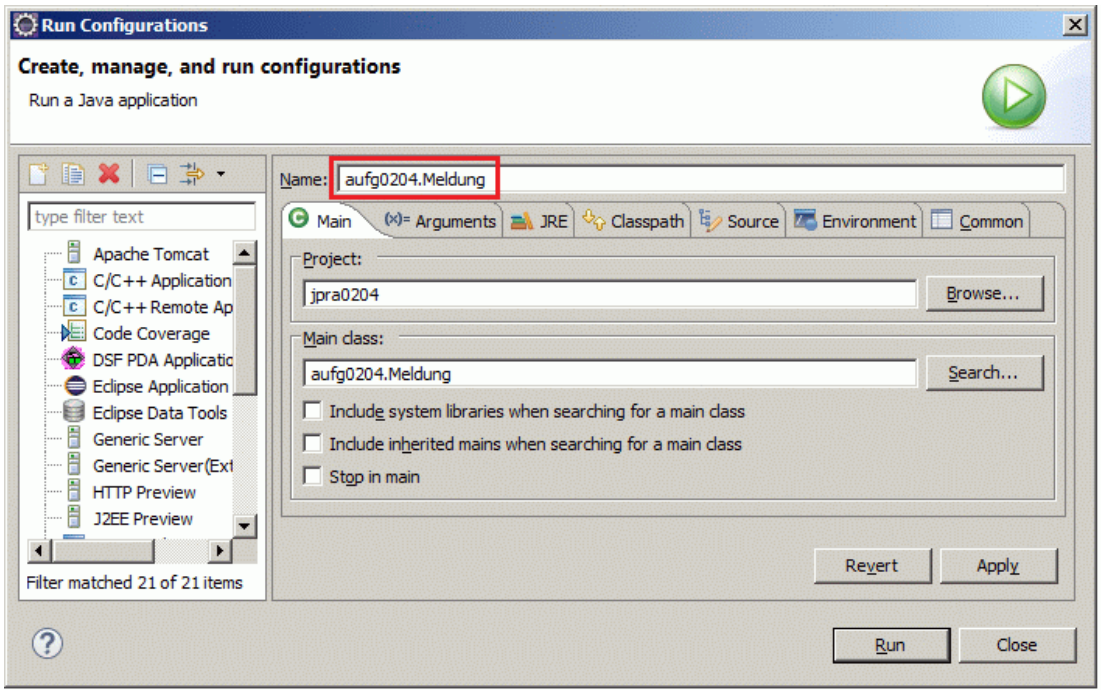
7. **Ausführen des Programms**

Die Ausführung des Programms erfolgt nach Klick auf den Button "Run".

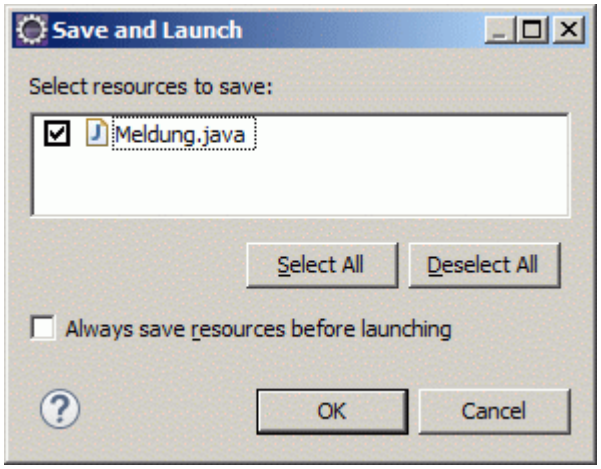


Daraufhin wird eine Lauf-Konfiguration ("Run Configuration") im Hintergrund erzeugt und das Programm mit Hilfe dieser Lauf-

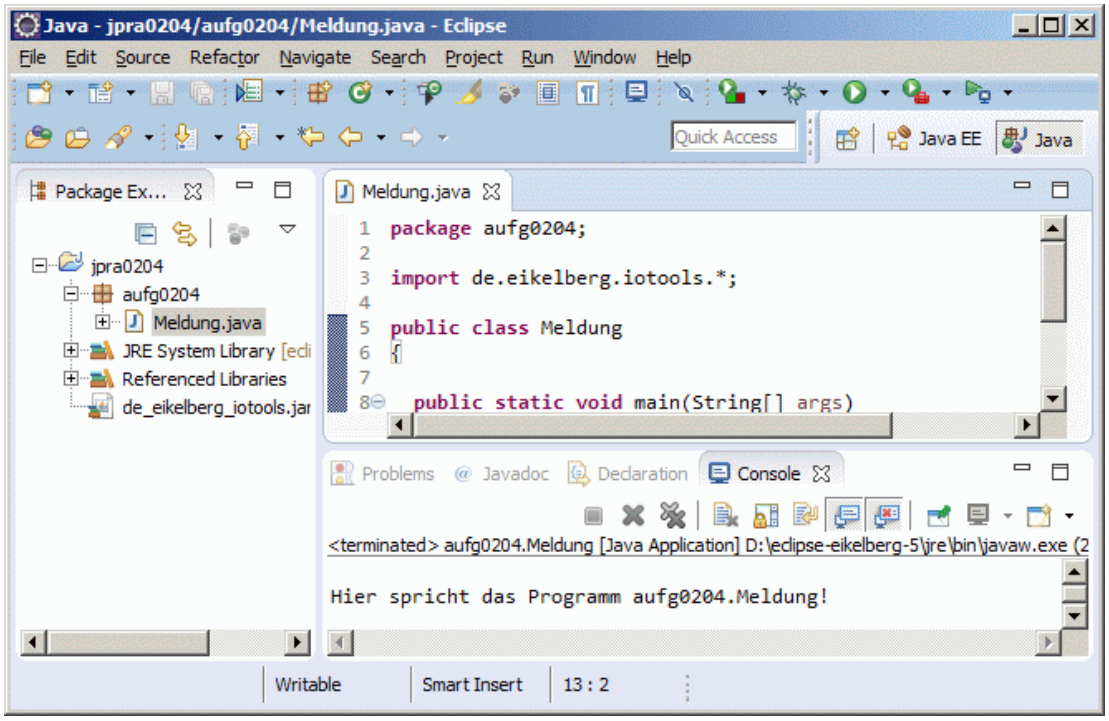
Konfiguration gestartet. Sie können diese Lauf-Konfiguration anpassen nach Klick auf den schwarzen Pfeil direkt neben dem grünen Run-Pfeil und Auswahl des Menüpunktes "Run Configurations". Im Fenster "Run" können Sie dann zum Beispiel wie folgt den Namen der Lauf-Konfiguration um den Paketnamen ergänzen:



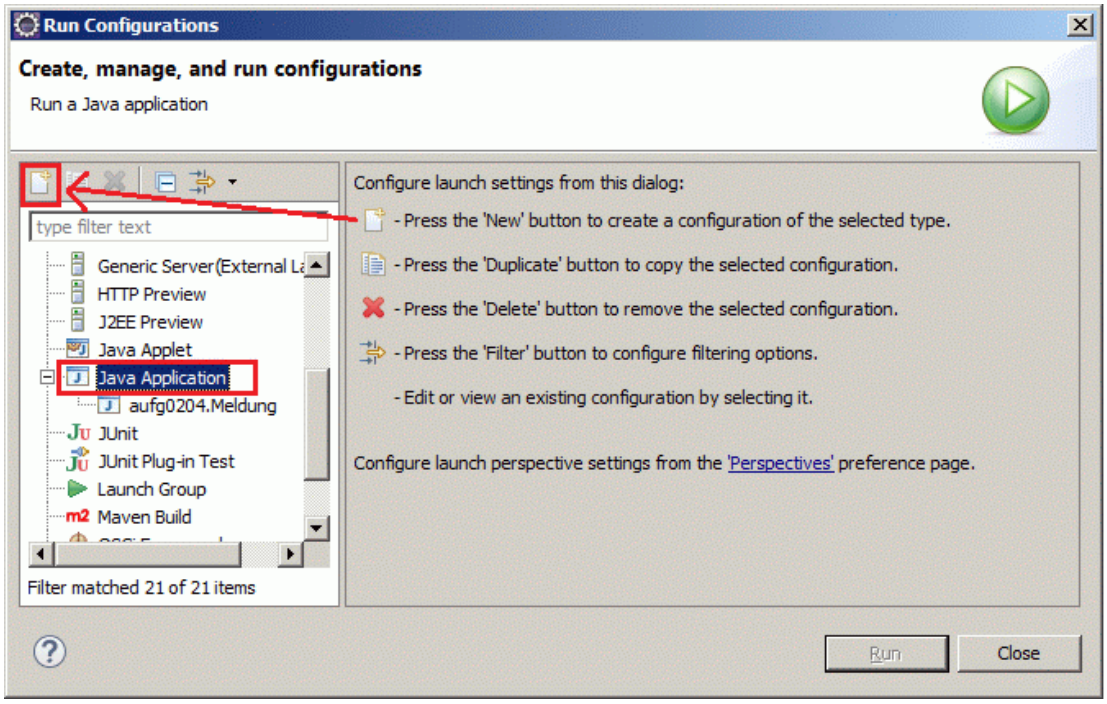
Aktivieren Sie Ihre Einträge durch Klick auf den Button Apply. Mit Klick auf den Button "Run" wird aus dem Fenster "Run" die Ausführung des Java-Programms gestartet. Falls sich der Quelltext des Programms nach dem letzten Speichern geändert hat, wird vor der Ausführung des Programms noch einmal die Speicherung der Quelldatei (und damit auch ein erneuter Compile) angestoßen:



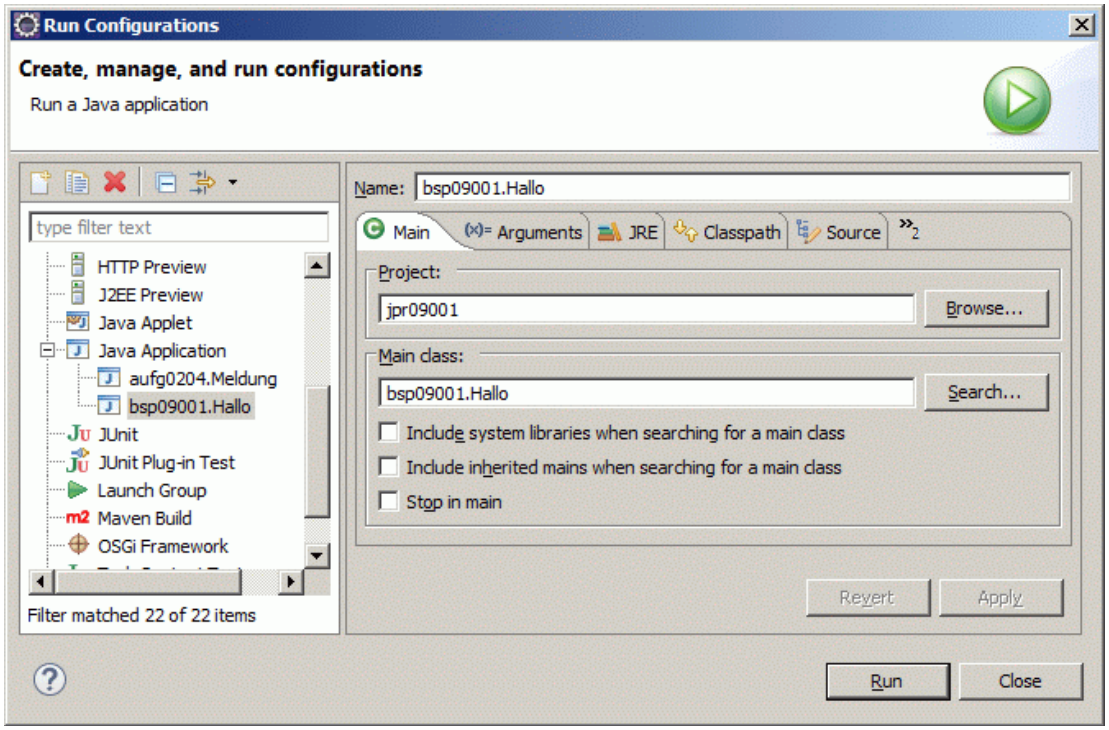
Im unteren Bereich der Java-Perspektive erscheint eine Registerkarte "Console", in der die Konsolaus- und eingabe des Programms erfolgt:



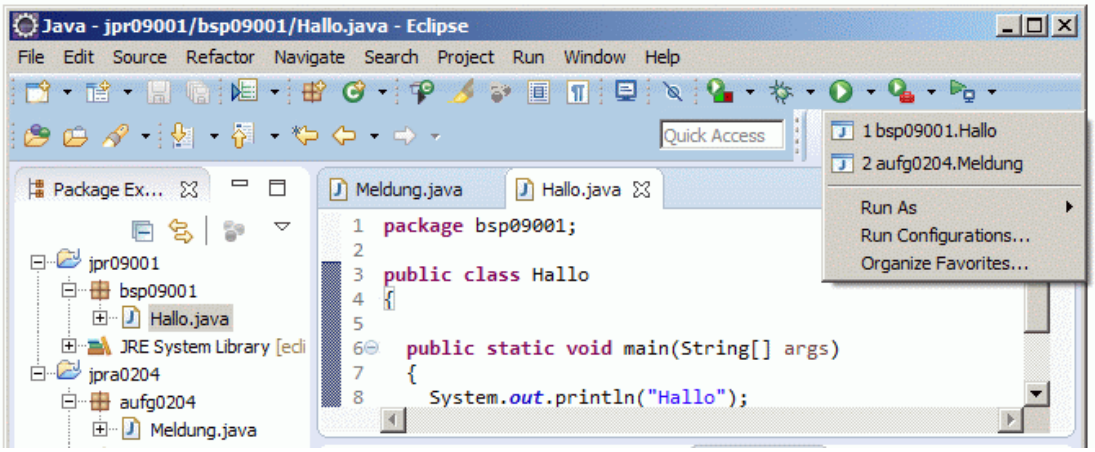
Hinweis: Werden mehrere Projekte im Eclipse 4.5.1 angezeigt, muss jedes auszuführende Programm in Eclipse 4.5.1 eingetragen werden. Dies geschieht nach Auswahl des Menüpunktes "Run..." im Menü "Run". Sie gelangen dann in das Fenster "Run" (s.o.).



Nach Markieren des Eintrags "Java-Application" im linken Bereich des Fensters und nach Klick auf die Schaltfläche "New" (s.o.) kann dann eine weitere main-Methode bzw. eine weitere Java-Anwendung eingetragen werden:



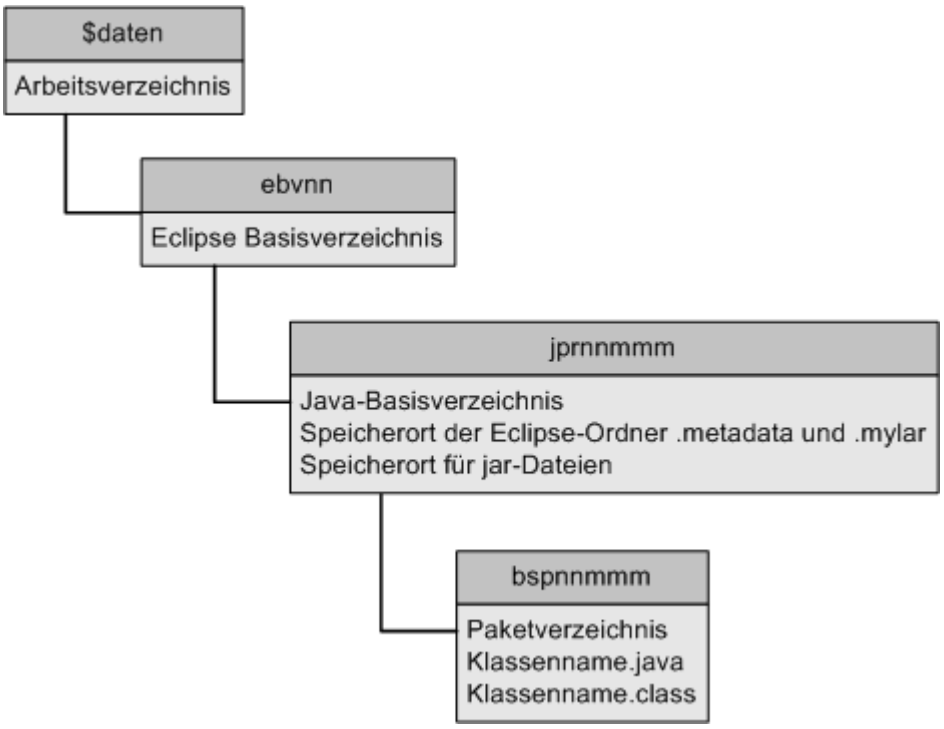
Öffnet man das Menü zum Button "Run", werden dann alle eingetragenen Java-Anwendungen zur Auswahl angeboten:



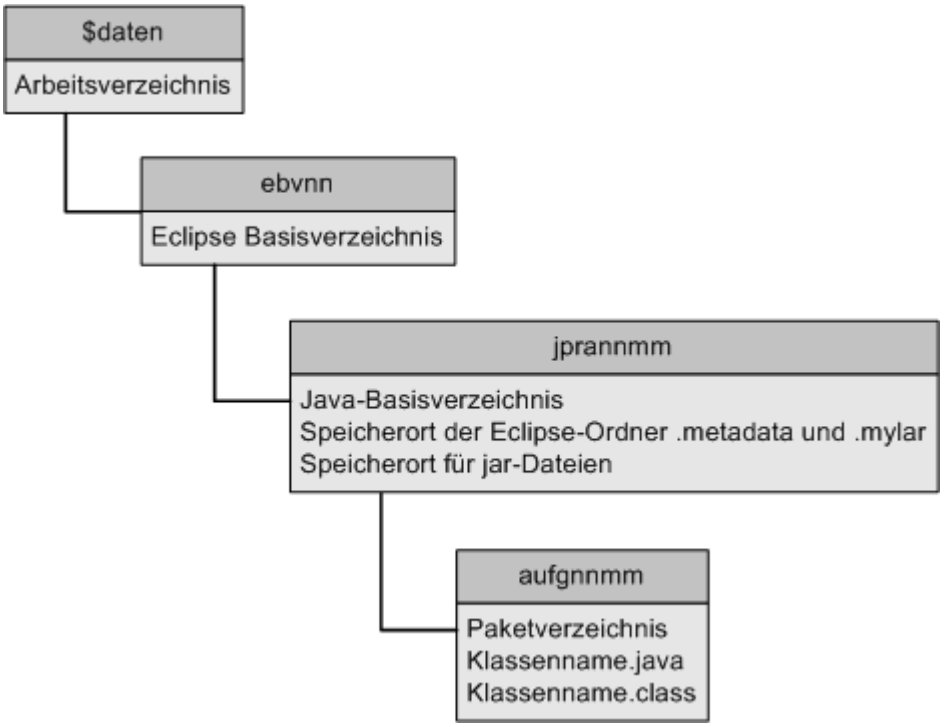
8. **Schließen und erneutes Öffnen eines Projektes**

   Das Schließen und Öffnen eines Projektes im Eclipse geschieht mit dem Kontextmenü des jeweiligen Projektes (Menüpunkte "Close Project" bzw. "Open Project "). Wird das zu öffnende Projekt nicht im Eclipse 4.5.1 angezeigt wird, kann es mit dem Menüpunkt "Import..." des "File"-Menüs importiert werden, was dann auch zur Anzeige des Projektes führt.

Die folgende Grafik fasst noch einmal die wesentlichen bei der Arbeit mit Eclipse zu beachtenden Punkte bezüglich Dateistruktur und Namenskonvention zusammen. Dabei ist zunächst die **Namensstruktur für die Beispiele** berücksichtigt. Man beachte, dass für die Beispiele zu diesem Kurs für jedes Kapitel ein eigenes Eclipse-Basisverzeichnis gewählt wurde. Statt des Namens **ebv** wurde für die Beispiele zu Kapitel ***nn*** der Verzeichnisname **ebv*nn*** gewählt. Damit ergibt sich für die Beispiele zu einem Kapitel folgende Struktur:



Für das Praktikum ist es ebenfalls empfehlenswert, für jedes Kapitel ein eigenes Eclipse-Basisverzeichnis anzulegen und dieses ebenfalls mit dem Namen **ebv*nn*** zu benennen. Insgesamt muss für das Praktikum entsprechend folgende Situation erreicht werden:



Zusammenfassung und Rezept: Der Ablauf zum Anlegen eines Eclipse-Projektes und zum Anlegen einer Java-Quelldatei geschieht (für die Lösung der Aufgaben zu Kapitel 1 und 2) in folgenden Schritten:

1. **Start der Entwicklungsumgebung** Eclipse
2. **Anlegen eines Eclipse Projektes**
3. **Anlegen einer Quelldatei** für eine Java-Klasse mit main-Methode (Hauptprogramm)
4. **Einfügen von Paketen** in jar-Dateien (z. Bsp. de.eikelberg.iotools) in das Projekt
5. **Codierung des Java-Programms**
6. **Speicherung der Quelldateien und Compile**
7. **Ausführen** des Programms
8. **Speichern der Dateien** des Eclipse-Projektes und **Schließen des Projektes**

2 Objektorientierte Java Konsolprogramme

2.1 Erweiterung der Ordnerstruktur für die Verarbeitung von Datei-I/O

In diesem Kapitel befassen wir uns mit dem Aufbau eines strukturierten Java Konsolprogramms. Einfachere Konsolprogramme lesen Daten über die Tastatureingabe in der Konsolbox sowie aus Dateien und schreiben Daten auf den Konsolbildschirm sowie in Dateien. Daher werden im Rahmen dieses Kapitels diese Formen der Ein- und Ausgabe behandelt, wobei zur Vereinfachung auf vorgefertigte Programmelemente zurückgegriffen wird.

Für das Einlesen aus Dateien und das Schreiben in Dateien - **Datei-I/O** gennannt, wobei I für Eingabe/Input und O für Ausgabe/Output steht - ist hierzu eine Erweiterung der Ordnerstruktur für das Java-Projekt erforderlich. Im Folgenden werden die entsprechenden - teilweise bereits bekannten und teilweise neuen Regeln - zusammengestellt:

- Für die Codierung von Java-Programmen mit der Entwicklungsumgebung Eclipse wird ein Verzeichnis benötigt, das den Arbeitsbereich (Workspace) bildet, in dem alle Programmierprojekte gespeichert werden. In diesen Lehrunterlagen wird dieses Verzeichnis **Eclipse Basisverzeichnis** genannt. Als Name für das Eclipse Basisverzeichnis wird jeweils der Name `ebv` bzw. `ebvnn` gewählt, wobei `nn` die Nummer des zugehörigen Kapitels ist.

Das Eclipse Basisverzeichnis `ebvnn` enthält die Java-Basisverzeichnisse `jprnnmmm` bzw `jprannmm`

- Alle Dateien und Verzeichnisse zu einem Programmierprojekt Nummer `mmm` zu Kapitel `nn` - in Zukunft kurz **Projekt nnmmm** genannt - sollen im Java-Basisverzeichnis bzw. in Unterverzeichnissen des Java-Basisverzeichnisses gespeichert werden. Das Java-Basisverzeichnis heißt `jprnnmmm` für Beispiele und `jprannmm` für Übungsaufgaben.

Das Java-Basisverzeichnis `jprnnmmm` bzw `jprannmm` enthält die Verzeichnisse zu den Java-Paketen

- Die zu dem Programmierprojekt gehörenden Java-Quelldateien sind standardmässig in **Paketen** mit Name `bspnnmmm` (Beispiele) bzw. `aufgnnmm` (Übungsaufgaben) angeordnet. Die Quelldateien enthalten daher stets die Anweisung
  ```
  package bspnnmmm;
  ```
  bzw.
  ```
  package aufgnnmm;
  ```
  . Für die Ordnerstruktur im Betriebssystem bedeutet dies, dass die Java-Quelldateien entsprechend in Unterverzeichnissen `bspnnmmm` bzw. `aufgnnmm` des Java-Basisverzeichnisses gespeichert werden müssen. Ferner gilt, dass die Folge der Unterverzeichnisse, die der Folge der Namen des Paketes entspricht, im Java-Basisverzeichnis beginnen muss.

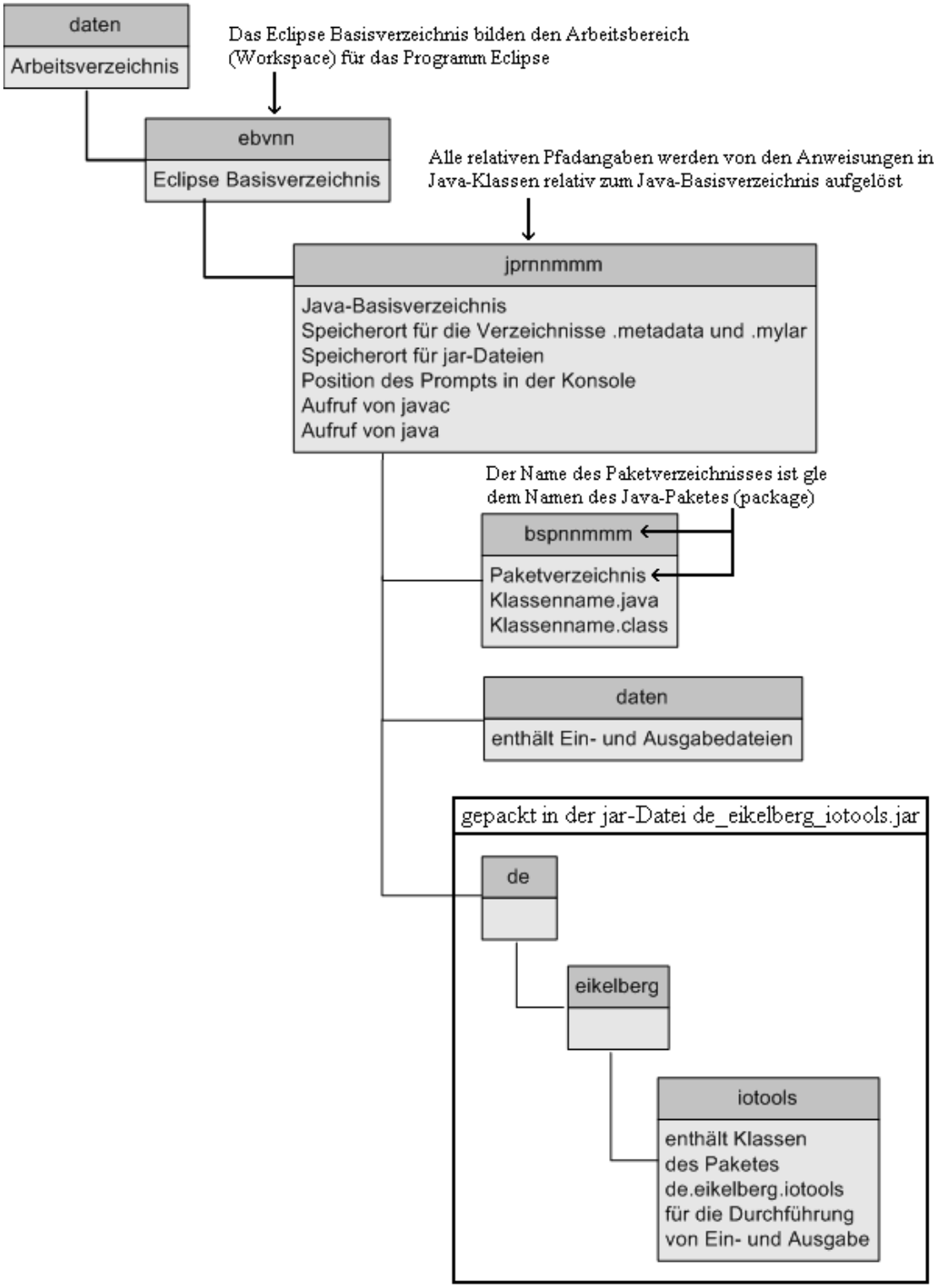
Die Java-Pakete bzw. die der Paketstruktur entsprechenden Ordner enthalten die Java-Quelldateien.

- Die von Eclipse angelegten Ordner `.metadata` und `.mylar` müssen im Eclipse Basisverzeichnis angeordnet sein.
- **Datendateien**, in die geschrieben werden soll oder aus denen gelesen werden soll, werden bei Programmierung relativer Pfadangaben von den Java-Programmen ausgehend von den Java-Basisverzeichnissen gesucht. Sie werden für die Beispielprogramme in der Regel **in einem Unterverzeichnis `daten`** des Java-Basisverzeichnisses angeordnet. Allgemeiner sucht die virutelle Maschine im Allgemeinen die Dateien, die die Java-Programme verwenden, stets in Verzeichnissen, deren Ort ausgehend vom Java-Basisverzeichnis - also relativ zum Java-Basisverzeichnis - bestimmt wird.
- Für das Programmierprojekt verwendete vorgefertigte Pakete müssen in das Java-Basisverzeichnis hineinkopiert werden. Diese Pakete sollten in Form von jar-Dateien bereitgestellt werden.
- Für den Download bereitgestellte ZIP-Archive zur Weitergabe von Beispiel-Klassen eines Programmierprojektes `jprnnmmmm` wurden bei der Erarbeitung dieses Kurses in der Regel `jprnnmmmm weitergabe.zip` genannt und enthalten in der Regel das Java-Basisverzeichnis mit darin angeordneten Dateien. In diesem Fall entsteht beim Entpacken des Archivs automatisch eine korrekte Verzeichnisstruktur mit Java-Basisverzeichnis.

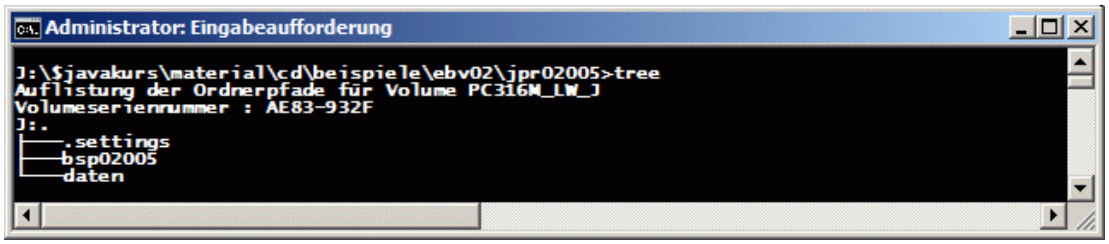
Für die Arbeit mit dem JDK bzw. mit Eclipse bedeutet dies im Einzelnen:

- Der aktuelle Pfad - bei Standardeinstellung in der Konsolbox von Windows von der Eingabeaufforderung angezeigt - muss auf das Java-Basisverzeichnis `jprnnmmm` bzw. `jprannmm` zeigen, wenn der Java Compiler oder die Virtuelle Maschine des JDK aufgerufen werden.
- Beim Start von Eclipse muss im Fenster "Workspace Launcher" stets das Eclipse Basisverzeichnis angegeben werden, in dem sich das Java-Basisverzeichnis des jeweilige Programmierprojektes befindet.

In der nachfolgenden Grafik werden die oben genannten Punkte noch einmal zusammengefasst:



Als Beispiel wird das Projekt 02005 betrachtet, dessen Java-Basisverzeichnis `jpr02005` auf dem Rechner des Autors in einem Verzeichnis `J:\$javakurs\material\dvd\beispiele\ebv02\jpr02005` angeordnet ist. Das Java-Basisverzeichnis `jpr02005` befindet sich somit in einem Eclipse-Basisverzeichnis `ebv02`. Die Unterverzeichnisstruktur des Java-Basisverzeichnisses `jpr02005` sieht wie folgt aus:



Das in der Verzeichnisstruktur enthaltene Verzeichnis `.settings` ist ein Verzeichnis, das nicht zum Java-Programm gehört. Vielmehr handelt es sich hier um ein Verzeichnis, in dem Eclipse Informationen zu dem entsprechenden Java-Programm speichert.

Das Basisverzeichnis enthält zwei vom Autor erstellte Batch-Dateien `$compile.bat` (Compile) und `Â§start.bat` (Programmstart) sowie die die jar-Datei de_eikelberg_iotools.jar (Paket de.eikelberg.iotools). Dateinamen und Verzeichnisse, deren Name mit einem Punkt beginnt, sind Dateien, die von Eclipse verwendet werden und nicht direkt zum Java-Programm gehören. Das Paketverzeichnis `bsp02005` enthält die Java-Klassen zum Programm und das Verzeichnis `daten` die Dateien, aus denen das Programm bsp02005.Encoding lesen bzw. in die dieses Programm schreiben soll.

```
Administrator: Eingabeaufforderung
J:\$javakurs\material\cd\beispiele\ebv02\jpr02005>dir /O:gn
 Datenträger in Laufwerk J: ist PC316N_LW_J
 Volumeseriennummer: AE83-932F

 Verzeichnis von J:\$javakurs\material\cd\beispiele\ebv02\jpr02005

31.07.2011  12:10    <DIR>          .
31.07.2011  12:10    <DIR>          ..
31.07.2011  12:10    <DIR>          .settings
03.09.2011  09:28    <DIR>          bsp02005
31.07.2011  12:10    <DIR>          daten
03.09.2011  09:28               269 $compile.bat
03.09.2011  09:28               169 $start.bat
20.02.2008  12:51               289 .classpath
15.02.2008  22:41               384 .project
16.09.2006  19:36            23.970 de_eikelberg_iotools.jar
               5 Datei(en),         25.081 Bytes
               5 Verzeichnis(se), 84.265.836.544 Bytes frei
```

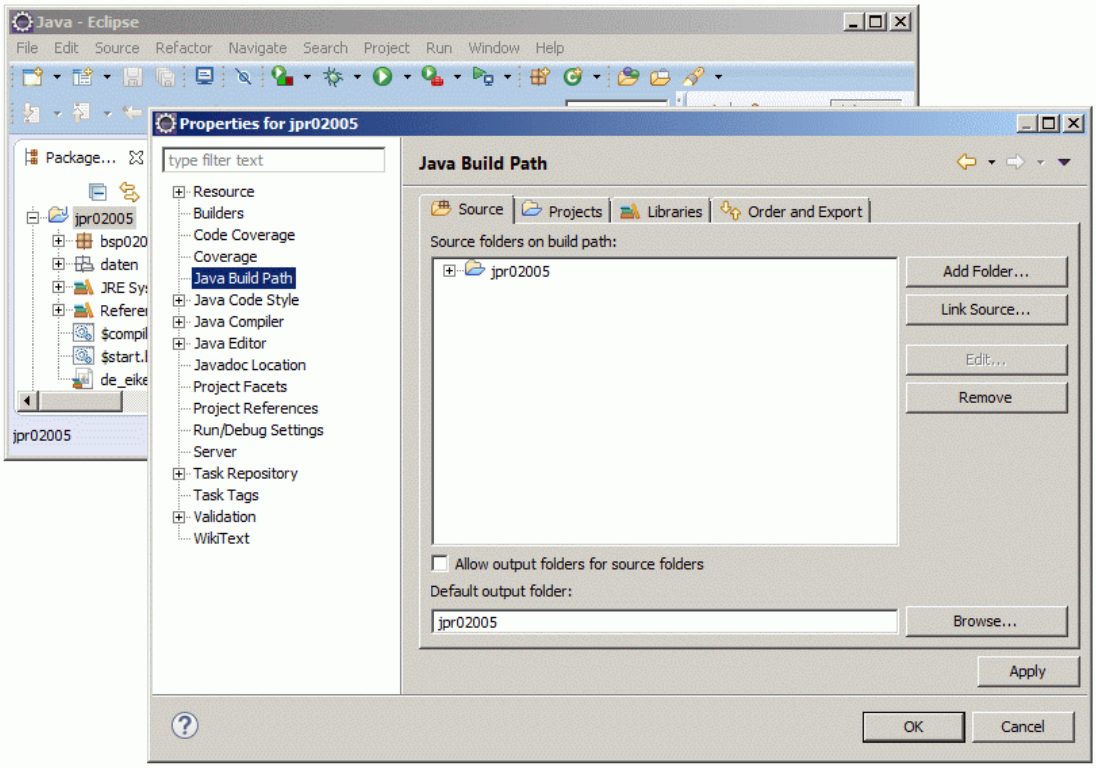
Der Programmcode zum Paket bsp02005 besteht aus einer einzigen Java-Datei mit Name `Encoding.java`. Im Programmcode wird mit der Anweisung

```
package bsp02005;
```

festgelegt, dass die Java-Klasse zum Paket bsp02005 gehören soll. Die Java Quelldatei `Encoding.java` sowie die zugehörige Klassendatei befinden sich dementsprechend im Verzeichnis `bsp02005`:

```
Administrator: Eingabeaufforderung
J:\$javakurs\material\cd\beispiele\ebv02\jpr02005>dir bsp02005 /O:gn
 Datenträger in Laufwerk J: ist PC316N_LW_J
 Volumeseriennummer: AE83-932F

 Verzeichnis von J:\$javakurs\material\cd\beispiele\ebv02\jpr02005\bsp02005

03.09.2011  09:28    <DIR>          .
03.09.2011  09:28    <DIR>          ..
03.09.2011  09:28             8.601 Encoding.class
16.02.2008  11:26            12.448 Encoding.java
               2 Datei(en),         21.049 Bytes
               2 Verzeichnis(se), 84.265.824.256 Bytes frei
```

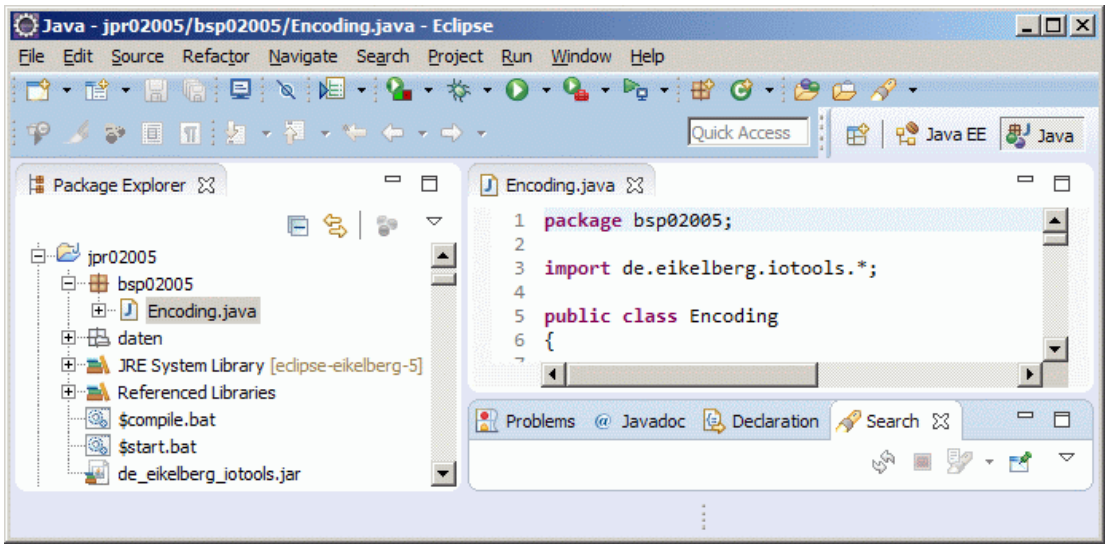
Entsprechend dieser Anordnung der Dateien ist das Java-Basisverzeichnis in den Projekteigenschaften von Eclipse an den entsprechenden Stellen (vgl. Kapitel 1.3.2) eingetragen:



Im Verzeichnis `daten` befinden sich die Dateien `eingabe.txt` und `ausgabe.txt`, aus denen das Programm bsp02005.Encoding liest bzw. in das das Programm schreibt:

```
Administrator: Eingabeaufforderung

J:\$javakurs\material\cd\beispiele\ebv02\jpr02005>dir daten /O:gn
 Datenträger in Laufwerk J: ist PC316M_LW_J
 Volumeseriennummer: AE83-932F

 Verzeichnis von J:\$javakurs\material\cd\beispiele\ebv02\jpr02005\daten

31.07.2011  12:10    <DIR>          .
31.07.2011  12:10    <DIR>          ..
29.02.2008  14:19             1.070 ausgabe.txt
05.09.2006  17:10                43 eingabe.txt
15.02.2000  13:49                 1 eingabe-0x91.txt
               3 Datei(en),        1.114 Bytes
               2 Verzeichnis(se), 84.262.096.896 Bytes frei
```

Die Jar-Datei `de_eikelberg_iotools.jar` enthält die Klassen des Paketes de.eikelberg.iotools. Der Inhalt dieser jar-Datei kann mit Eclipse betrachtet werden, indem in der Registerkarte "Package Explorer" von Eclipse im Zweig "Referenced Libraries" der Zweig zu dieser Jar-Datei erweitert wird. Das Paket de.eikelberg.iotools wird in Kapitel 2.3 erklärt und besteht aus Klassen zum vereinfachten Einlesen bzw. Ausgeben von Daten:



Der Java Compiler und die Java Virtuelle Maschine des Java Development Kits können jetzt für das Programm bsp02005.Encoding aufgerufen werden, wenn der aktuelle Pfad in der Konsolbox auf das Java-Basisverzeichnis zeigt.

```
Administrator: Eingabeaufforderung

...\jpr02005>..\jdk1.8.0\bin\javac -classpath .;de_eikelberg_iotools.jar bsp02005\*.java

...\jpr02005>..\jdk1.8.0\bin\java -classpath .;de_eikelberg_iotools.jar bsp02005.Encoding

--------------------------------------------------------------------------------
    Programm zur Codierung und Decodierung von Zeichen
    Copyright 1999,2006 by M. Eikelberg (www.eikelberg.de)

--------------------------------------------------------------------------------

Dieses Programm ist nur zur Verwendung auf Windows Systemen
oder Systemen mit Unicode Verschluesselung konzipiert!

Wird Umwandlung von (Unicode-)Integercodierung
   nach Zeichen gemaess Codierung Ihres Systems [i]
oder Umwandlung Zeichen gemaess Codierung Ihres Systems
   nach (Unicode-)Integercodierung [u]
gewuenscht ? : i

Das Programm wird beendet bei Eingabe einer negativen Zahl !
Die Ausgabe finden Sie in der Datei ausgabe.txt!

Geben sie den Integer-Wert des Unicode-Zeichens ein : 223
Eingegeben wurde: 223


Geben sie den Integer-Wert des Unicode-Zeichens ein : -1
```

Beim Programmlauf erfolgt eine Ausgabe in die Datei `daten\ausgabe.txt`:

```
Administrator: Eingabeaufforderung
...\jpr02005>notepad daten/ausgabe.txt
...\jpr02005>
```

```
ausgabe.txt - Editor
Datei  Bearbeiten  Format  Ansicht  ?
*********************************************************************
* Haben Sie diese Datei unter Windows erstellt, so muessen Sie      *
* sie mit einem Windows-Editor betrachten. Haben Sie die Datei      *
* unter einem anderen Betriebssystem erstellt, so erhalten Sie      *
* nur dann eine korrekte Anzeige, wenn Betriebsystem und Editor     *
* gemaess Unicode ver- bzw. entschluesseln.                         *
* Ansonsten muessen Sie wegen unterschiedlicher Codierungen mit     *
* fehlerhaften Anzeigen fuer Zeichen zwischen hexadezimal           *
* 0x007F (dezimal 128) und 0x00FF (dezimal 255) rechnen!            *
*********************************************************************

Der Unicode-Dezimalwert 223 ergibt das Zeichen: 'ß'
```

Exkurs: Packen und Entpacken von jar-Dateien

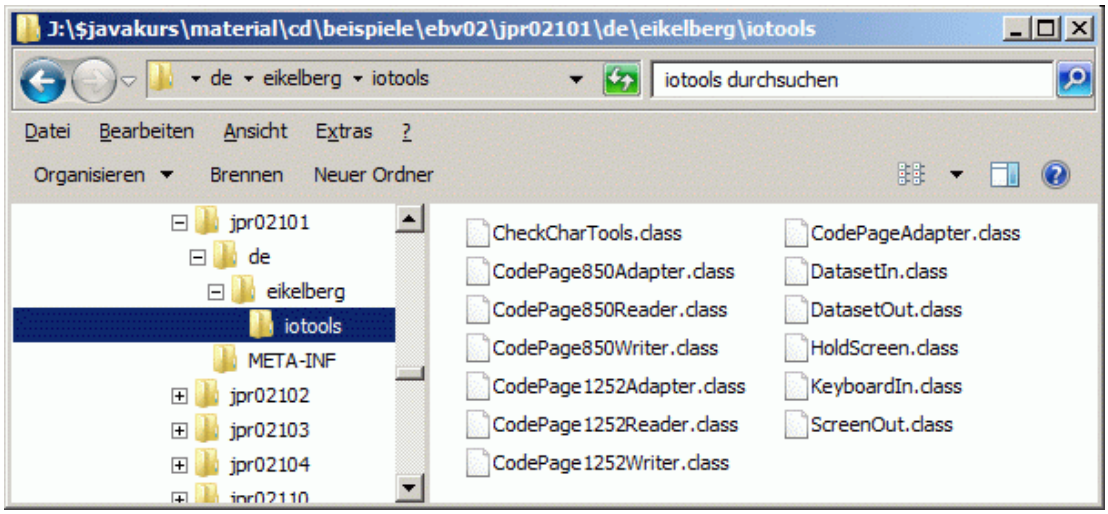
Das Packen und Entpacken von jar-Dateien erfolgt in analoger Weise zum Packen und Entpacken von ZIP-Archiven. Für jar-Dateien steht hierzu das Programm `jar.exe` des JDK zur Verfügung. Das Programm jar.exe wird in der Konsolbox aufgerufen und über verschiedene Parameter gesteuert:

```
Administrator: Eingabeaufforderung
...\jpr02101>..\jdk1.8.0\bin\jar
Verwendung: jar {ctxui}[vfmn0PMe] [jar-file] [manifest-file] [entry-point] [-C dir] Dateien...
Optionen:
    -c  Neues Archiv erstellen
    -t  Inhaltsverzeichnis für Archiv anzeigen
    -x  Benannte (oder alle) Dateien aus dem Archiv extrahieren
    -u  Vorhandenes Archiv aktualisieren
    -v  Ausgabe im Verbose-Modus aus Standard-Ausgabe generieren
    -f  Dateinamen für Archiv angeben
    -m  Manifestinformationen aus angegebener Manifestdatei einschließen
    -n  Pack200-Normalisierung nach Erstellung eines neuen Archivs ausführen
    -e  Anwendungseinstiegspunkt für Standalone-Anwendung angeben
        in einer ausführbaren JAR-Datei gebündelt
    -0  Nur speichern; keine ZIP-Komprimierung verwenden
    -P  Komponenten mit vorangestelltem "/" (absoluter Pfad) und ".." (übergeordnetes Verzeichnis) aus Dateinamen beibehalten
    -M  Keine Manifest-Datei für die Einträge erstellen
    -i  Indexinformationen für die angegebenen JAR-Dateien erstellen
    -C  Zum angegebenen Verzeichnis wechseln und folgende Datei einschließen
Falls eine Datei ein Verzeichnis ist, wird dieses rekursiv verarbeitet.
Der Name der Manifestdatei, der Name der Archivdatei und der Name des Einstiegspunkts werden
in derselben Reihenfolge wie die Kennzeichen "m", "f" und "e" angegeben.

Beispiel 1: Archivieren Sie zwei Klassendateien in ein Archiv mit Namen "classes.jar":
       jar cvf classes.jar Foo.class Bar.class
Beispiel 2: Verwenden Sie die vorhandenen Manifestdatei "mymanifest", und archivieren Sie alle
           Dateien im Verzeichnis foo/ directory in "classes.jar":
       jar cvfm classes.jar mymanifest -C foo/ .
```

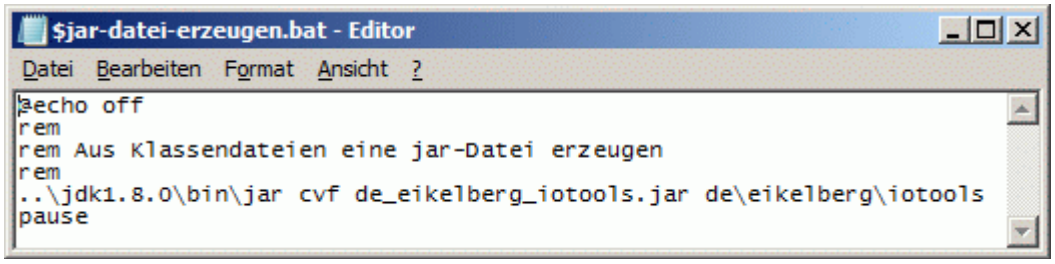
Am einfachsten erfolgt das Packen und Entpacken mit Hilfe von Batch-Dateien entsprechend den Beispielen jpr02101 und jpr02102. Im Beispiel jpr02101 wird die Datei de_eikelberg_iotools.jar wie folgt entpackt:

```
$jar-datei-entpacken.bat - Editor
Datei  Bearbeiten  Format  Ansicht  ?
@echo off
rem
rem jar-Datei entpacken
rem
..\jdk1.8.0\bin\jar -xvf de_eikelberg_iotools.jar
pause
```

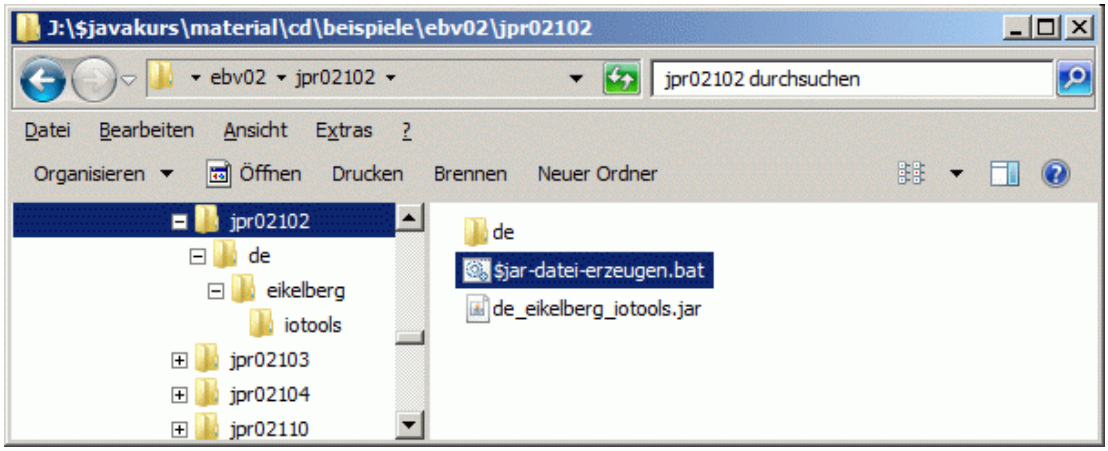
Das Entpacken von de_eikelberg_iotools.jar ergibt einen Verzeichnisbaum aus drei Unterverzeichnissen und im untersten Unterverzeichnis 13 Dateien:



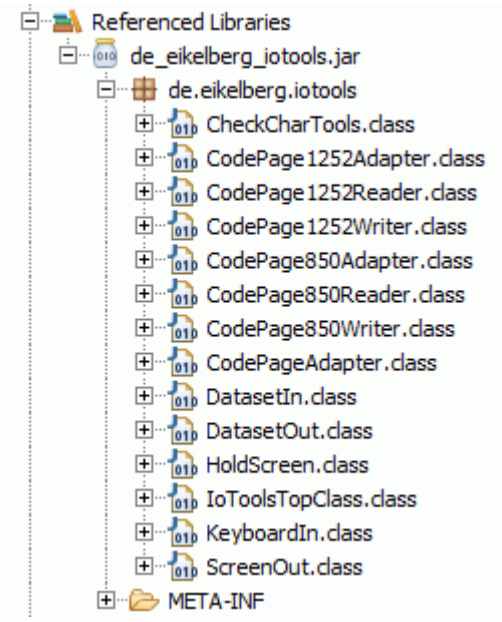
Umgekehrt erfolgt das Packen wie im Beispiel jpr02102. Dort werden die Klassen des Paketes de.eikelberg.iotools (angeordnet im Unterverzeichnis de\eikelberg\iotools des Basisverzeichnisses jpr02102) in die jar-Datei de_eikelberg_iotools.jar gepackt:



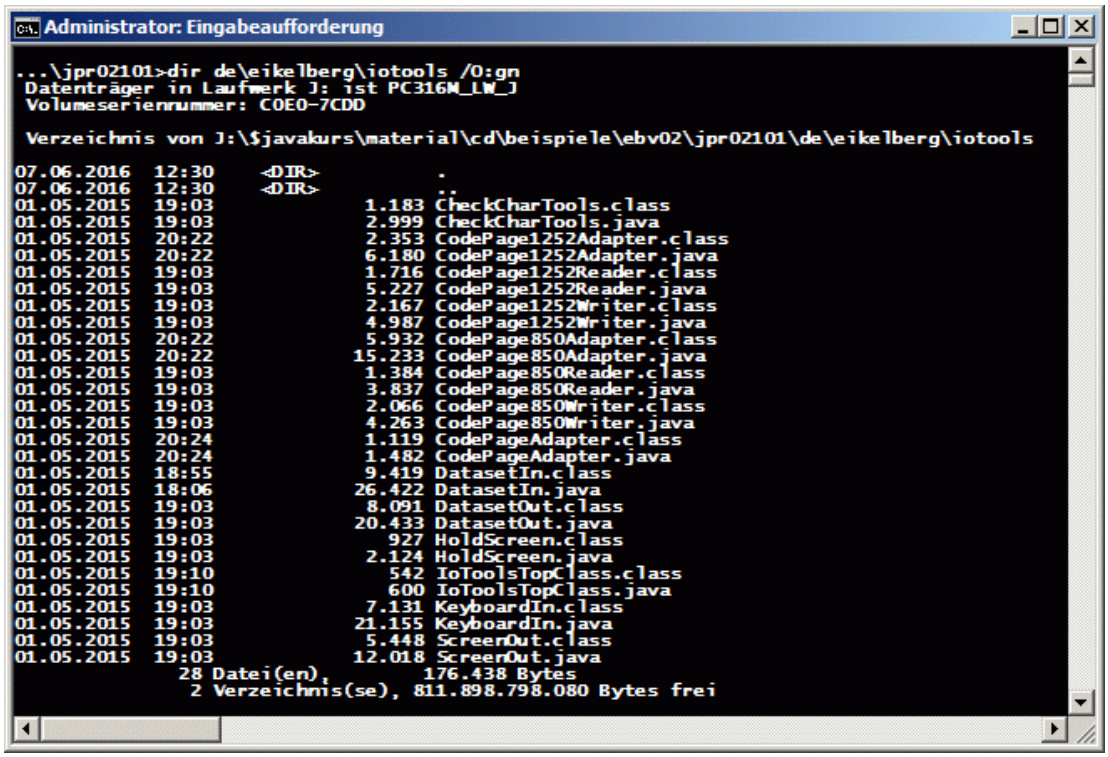
Der folgende Screenshot zeigt die Lage der Dateien.



Nach Aufruf der Batch-Datei $jar-datei-erzeugen.bat werden die 13 Klassen des Paketes de.eikelberg.iotools in der jar-Datei gepackt gespeichert. In Eclipse wird diese jar-Datei wie folgt angezeigt, wenn Sie in ein Projekt eingebunden wurde:



Der Inhalt der jar-Datei entspricht damit dem Inhalt des Verzeichnisses de\eikelberg\iotools:



```
Administrator: Eingabeaufforderung

...\jpr02101>dir de\eikelberg\iotools /O:gn
 Datenträger in Laufwerk J: ist PC316M_LW_J
 Volumeseriennummer: C0E0-7CDD

 Verzeichnis von J:\$javakurs\material\cd\beispiele\ebv02\jpr02101\de\eikelberg\iotools

07.06.2016  12:30    <DIR>          .
07.06.2016  12:30    <DIR>          ..
01.05.2015  19:03             1.183 CheckCharTools.class
01.05.2015  19:03             2.999 CheckCharTools.java
01.05.2015  20:22             2.353 CodePage1252Adapter.class
01.05.2015  20:22             6.180 CodePage1252Adapter.java
01.05.2015  19:03             1.716 CodePage1252Reader.class
01.05.2015  19:03             5.227 CodePage1252Reader.java
01.05.2015  19:03             2.167 CodePage1252Writer.class
01.05.2015  19:03             4.987 CodePage1252Writer.java
01.05.2015  20:22             5.932 CodePage850Adapter.class
01.05.2015  20:22            15.233 CodePage850Adapter.java
01.05.2015  19:03             1.384 CodePage850Reader.class
01.05.2015  19:03             3.837 CodePage850Reader.java
01.05.2015  19:03             2.066 CodePage850Writer.class
01.05.2015  19:03             4.263 CodePage850Writer.java
01.05.2015  20:24             1.119 CodePageAdapter.class
01.05.2015  20:24             1.482 CodePageAdapter.java
01.05.2015  18:55             9.419 DatasetIn.class
01.05.2015  18:06            26.422 DatasetIn.java
01.05.2015  19:03             8.091 DatasetOut.class
01.05.2015  19:03            20.433 DatasetOut.java
01.05.2015  19:03               927 HoldScreen.class
01.05.2015  19:03             2.124 HoldScreen.java
01.05.2015  19:10               542 IoToolsTopClass.class
01.05.2015  19:10               600 IoToolsTopClass.java
01.05.2015  19:03             7.131 KeyboardIn.class
01.05.2015  19:03            21.155 KeyboardIn.java
01.05.2015  19:03             5.448 ScreenOut.class
01.05.2015  19:03            12.018 ScreenOut.java
              28 Datei(en),        176.438 Bytes
               2 Verzeichnis(se), 811.898.798.080 Bytes frei
```

Hinweis: Die Klassen des Paketes de.eikelberg.iotools korrigieren (soweit möglich) ein Problem der Codierung von Schriftzeichen in Windows-Betriebssystemen, das in Kapitel 2.2.3 genauer beschrieben wird. Dieses Problem betrifft die Darstellung der deutschen Umlaute 'ä','ö','ü','Ä','Ö','Ü' sowie des Zeichens 'ß'. Diese werden stellenweise nicht richtig angezeigt, was von den Klassen des Paketes de.eikelberg.iotools - soweit dies möglich ist - korrigiert wird. Bei Anwendung der Klassen dieses Paketes werden die genannten Sonderzeichen korrekt verarbeitet mit Ausnahme des Zeichens 'ü' in der Konsolbox. Dieses kann ab Java-Version 1.2 prinzipiell nicht mehr von der Java virtuellen Maschine in der Konsolbox ausgegeben werden. Daher geben die Klassen des Paketes de.eikelberg.iotools statt des Zeichens 'ü' in der Konsolbox das Zeichen 'u' aus. Dieses Verhalten ist in diesem Skript in vielen Bildschirmausdrucken sichtbar.

2.2 Grundelemente eines Java-Programms

2.2.1 Die Java-Klasse

Wir befassen uns in diesem Kapitel mit dem wichtigsten Teil eines Programms, dem Hauptprogramm. Das Hauptprogramm bildet den Anfangs- und Endpunkt der Verarbeitung und enthält die 'oberste' Ebene der Verarbeitung. Bei der Programmierung in Java liegt das Hauptprogramm als spezieller Bestandteil einer Java Klasse vor:

2.2.1.1 Die Kennzeichnung des Hauptprogramms in einer Java-Klasse

Wie bereits in Kapitel 1.2 erklärt gibt es neben den Dateien mit **Java-Quellprogrammcode**, dem Betriebssystem als solche gekennzeichnet mit der Dateiendung **.java**, die Dateien mit **Byte-Code**, **Klassendateien** genannt und dem Betriebssystem gegenüber kenntlich gemacht mit der Dateiendung **.class**. Diese Klassendateien enthalten - wie in Kapitel 1.4.2 erklärt - jeweils eine Java-Klasse.

Obwohl Klassendateien also ausführbaren Byte-Code enthalten, darf der Begriff Klasse jedoch nicht mit dem Begriff Programm gleichgesetzt werden. Vielmehr ist der Klassenbegriff allgemeiner gehalten: Java-Klassendateien können neben Byte-Code zu einer Java-Anwendung auch nicht einzeln lauffähige Klassen, Programmcode zu einem Fenster bzw. zur Ereignisbehandlung oder aber auch Applets enthalten.

Wie erkennt die JVM, dass die ihr übergebene Klasse eine Klasse zu einem Java Hauptprogramm ist und mit welcher Anweisung beginnt die JVM die Ausführung der Klasse, die ihr übergeben worden ist? Für Java Programme bzw. besser Java-Anwendungen wird diese Frage dadurch geregelt, dass die Klassendefinition eines an die JVM zur Ausführung übergebenen Hauptprogramms stets **eine Methode mit Namen main() enthalten muss**. Für die Methode main() gelten folgende Bemerkungen:

- Die JVM beginnt stets mit der Ausführung der Methode main(), wenn ihr eine Klassendatei übergeben wird.
- Die Methode main() ist wie jede andere Methode einer Klasse zu programmieren, wird jedoch durch ihren Namen als (möglicher) Startpunkt einer Java-Anwendung hervorgehoben.
- Klassendateien können auch dann zur Instantiierung von Objekten verwendet werden, wenn sie eine Methode main() enthalten. In diesem Fall wird die Methode main() von der instantiierenden Anwendung nicht verwendet.

Die Methode main() wird in Kapitel 2.2.2 genauer erklärt. In diesem Kapitel wenden wir uns zunächst der Form einer Klassendatei zu:

2.2.1.2 Die Grundform einer Java Klassendefinition

Jede Datei mit Quellcode zu einer Java Klassendatei muss folgende Bedingungen erfüllen:

1. Der Quellcode zu einer Java-Klasse mit Namen "Klassenname" muß in einer Datei mit Namen `Klassenname.java` gespeichert werden.
2. Der Quellcode muß innerhalb der Datei `Klassenname.java` in einem Bereich codiert werden,
   - der mit **`public class Klassenname {`** beginnt und
   - der mit einer Klammer **`}`** endet
3. Nach dem Compile befindet sich der Bytecode zur Java-Klasse mit Name Klassenname in einer Datei mit dem Namen `Klassenname.class`.

Es besteht somit eine eineindeutige Entsprechung des Java-Quellcodes im mit `public class Klassenname { ... }` gekennzeichneten Bereich und den Dateien `Klassenname.java` sowie `Klassenname.class`. Daher wird der Begriff Klasse oft synonym sowohl für den Bereich mit Quellcode als auch für die beiden Dateien verwendet. Man beachte hierbei folgende Punkte:

- Beim Klassennamen `Klassenname` wird zwischen Großbuchstaben und Kleinbuchstaben unterschieden!
- Auch bei der Datei `Klassenname.class` wird zwischen Groß- und Kleinschreibung unterschieden. Es muß daher unbedingt darauf geachtet werden, daß zum Beispiel beim Kopieren oder bei der Übertragung einer Datei auf ein Dateisystem eines anderen Betriebssytems nicht Kleinbuchstaben gegen Großbuchstaben oder umgekehrt ausgetauscht werden. Diese Probleme können am einfachsten vermieden werden, wenn der Klassenname stets vollständig aus Kleinbuchstaben besteht.
- Das Schlüsselwort `public` regelt die sogenannte "Sichtbarkeit" der Klasse. Es ist erst im Zusammenhang mit der objektorientierten Programmierung von Interesse und muß bei strukturierten Java-Anwendungen stets eingefügt werden.
- Außerhalb des mit `public class Klassenname` gekennzeichneten Bereiches für den Java-Quellcode zur Klasse können nur noch die **package**-Anweisung sowie **import**-Anweisungen stehen (vgl. 2.2.1.3 und 2.2.1.4).

Die einfachste Form einer Java-Klasse beinhaltet also zum Beispiel folgenden Java Quellcode, wenn der Name der Klassendatei ClassAufbau.java ist:

```
public class ClassAufbau
{

}
```

Man beachte, dass diese Klasse noch nicht Bestandteil eines Paketes ist.

2.2.1.3 Die package-Anweisung

Vor dem Bereich für den Quellcode einer Klasse kann zur Gruppierung von Java-Klassen maximal eine **package**-Anweisung eingefügt werden: Die **package**-Anweisung hat die Form

```
package name1.name2.name3;
```

wobei auch mehr oder weniger als drei Namen, jedoch mindestens ein Name `name1` codiert werden können. Diese Anweisung fügt die Klassendatei in ein **Paket** ein, das bei der Ausführung in einer Folge von Unterverzeichnissen `name1\name2\name3\` gesucht wird. Die package-Anweisung muß die erste Anweisung in einer Java-Quelldatei sein und insbesondere vor import-Anweisungen stehen.

Möchte man also die in 2.2.1.2 angeführte Klasse ClassAufbau in ein Paket bsp02001 einfügen, so muss man den Programmcode in der Datei ClassAufbau.java wie folgt erweitern:

```
package bsp02001;

public class ClassAufbau
{

}
```

Pakete sind bereits in Kapitel 1 angesprochen worden und werden in diesem Java-Kurs sowohl zur Gruppierung der Beispielprogramme als auch bei der Bereitstellung fertiger Java-Klassen für die Ein- und Ausgabe (vgl. Kapitel 2.3) verwendet.

2.2.1.4 import-Anweisungen

Eine **import**-Anweisung hat die Form

```
import name1.name2.Klassenname;
```

wobei wieder mehr oder weniger als zwei Namen `name1 bzw. name2`, jedoch mindestens ein Name `name1` codiert werden können. Mit der import-Anweisung wird dem Compiler mitgeteilt, welche weiteren Klassen(dateien) vom jeweiligen Java-Programm verwendet werden. Dabei teilt sich die Angabe `name1.name2.Klassenname` in einen Paketnamen (hier name1.name2) und einen Klassennamen (hier Klassenname) auf.

Alternativ ist es auch möglich, statt eines spezifischen Klassennamens mit Hilfe der generischen Angabe * alle Klassen eines Paketes anzugeben. Dies kann auch dann erfolgen, wenn im Programm nicht alle Klassen des angegebenen Paketes verwendet werden. Die import-Anweisung hat in diesem Fall dann die Form

```
import name1.name2.*;
```

Mit dem Klassennamen zur import-Anweisung wird wie bei der package-Anweisung wieder eine Folge von Verzeichnisssen - also eine Unterverzeichnisstruktur - festgelegt. Diese Unterverzeichnisstruktur wird von der Java virtuellen Maschine (JVM) ausgehend von dem Verzeichnis gesucht, von der aus die JVM aufgerufen wird. Daher sollte die JVM (und auch der Compiler) vom Basisverzeichnis aus aufgerufen werden.

Als Beispiel erweitern wir das Programm bsp02001.ClassAufbau zum Programm bsp02002.ClassAufbau durch Einfügen einer import-Anweisung, die die Methode `HoldScreen()` aus einem Paket mit Namen `de.eikelberg.iotools` verfügbar macht:

```
package bsp02002;

/* Bei der Verwendung von Packages wird der Pfad zu den Klassen
   des Paketes aus der Pfadangabe zum Java-Basisverzeichniss
   sowie dem Paketnamen zusammengesetzt.
*/

import de.eikelberg.iotools.HoldScreen;

@SuppressWarnings("unused")  // Unterdrücken der Warnung wg. unbenutztem Import
public class ClassAufbau
{

}
```

Alternativ könnte auch allgemeiner folgender Programmcode verwendet werden:

```
package bsp02002;

/* Bei der Verwendung von Packages wird der Pfad zu den Klassen
   des Paketes aus der Pfadangabe zum Java-Basisverzeichniss
```

```
   sowie dem Paketnamen zusammengesetzt.
*/

import de.eikelberg.iotools.*;

@SuppressWarnings("unused")  // Unterdrücken der Warnung wg. unbenutztem Import
public class ClassAufbau
{

}
```

Der Kommentar im Programm erläutert noch einmal, wo Compiler und JVM die Klassen aus einem selbst codierten Paket wie zum Beispiel dem Paket de.eikelberg.iotools sucht. Das Paket de.eikelberg.tools besteht im übrigen aus Klassen, die keine Hauptprogramme bereitstellen, sondern in anderen Programmen wiederverwendbare Prozeduren (Vgl. Kapitel 2.3).

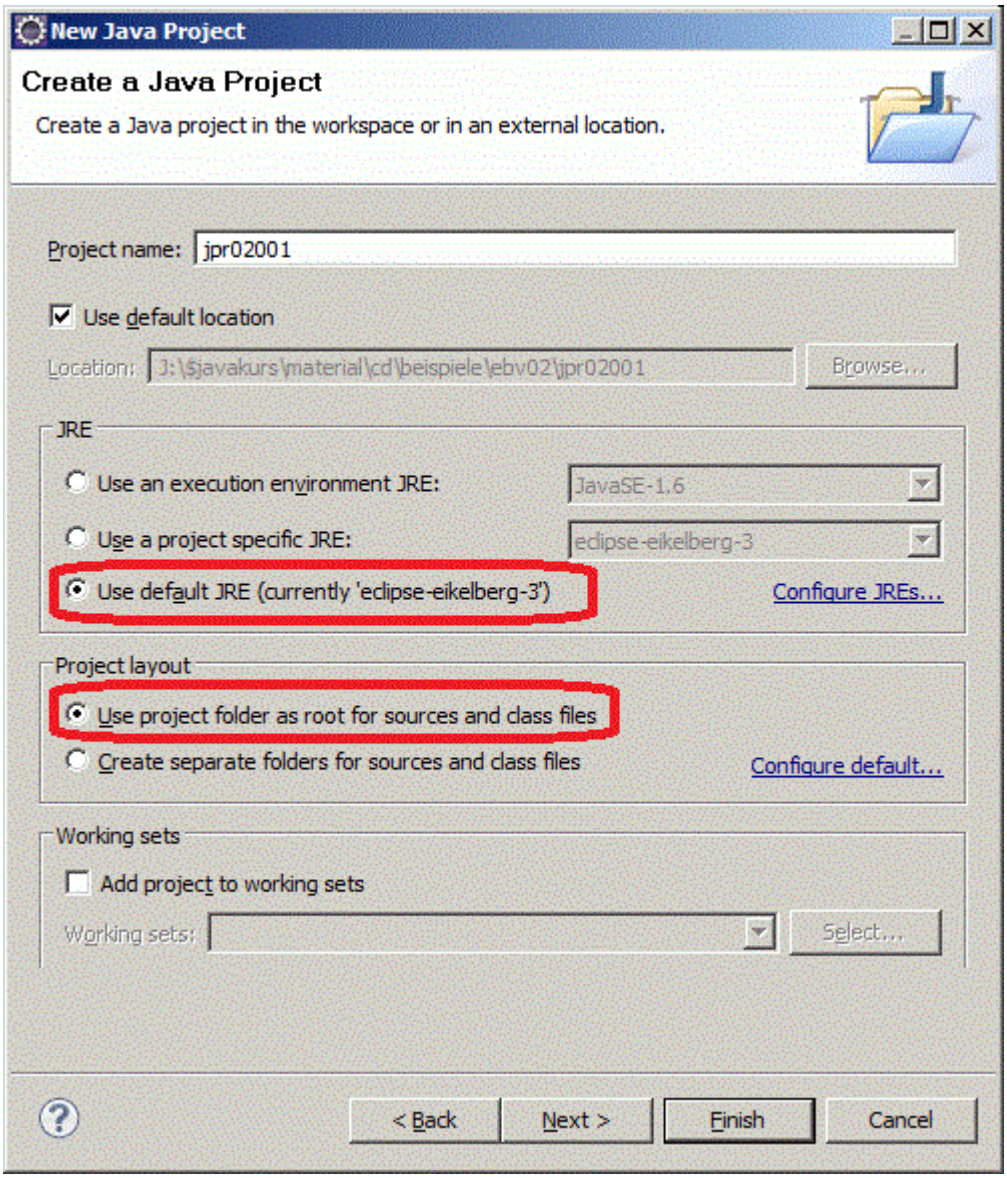
2.2.1.5 Erstellung einer Klassendatei mit Eclipse

Die Erstellung einer Klassendatei mit Eclipse erfolgt mit Hilfe des Assistenten zum Anlegen einer neuen Java-Klasse. Wichtig ist dabei, dass zunächst gemäß Kapitel 1.3.2 bzw. 2.1 beschrieben ein Eclipse-Projekt angelegt wird, in das auch alle benötigten Pakete eingefügt werden. Für eine Anwendung bsp02001.ClassAufbau, anzulegen in einem Projekt jpr02001 bedeutet dies somit *vor Anlegen der Klassendatei* folgende Schritte:

Anlegen des Eclipse Basisverzeichnisses mit Hilfe des Windows Explorers, sofern dies nicht bereits existiert. Das Eclipse Basisverzeichnis hat hier den Namen `ebv02` und befindet sich im Verzeichnis `J:\$javakurs\material\dvd\beispiele`.

1. Starten von Eclipse unter Verwendung des Eclipse Basisverzeichnisses `J:\$javakurs\material\dvd\beispiele\ebv02`.
2. Anlegen eines Projektes mit Hilfe der Auswahl File-Menü / New / Project. Man wähle den Assistenten (wizard) für das Anlegen eines Java-Projektes und klicke auf den Butten "Next". Es öffnet sich das Fenster "New Java Project". In diesem Fenster
   - trage man den Namen des Basisverzeichnisses (jpr02001) als Projektname ein,
   - man wähle die Standard-Java-Laufzeitumgebung (default JRE - Java Runtime Environment) der bereitgestellten Eclipse-Version (das ist Version 1.8.0) aus
   - und man lege fest, dass der Projektordner als Java-Basisverzeichnis zu verwenden ist (use project folder as root for sources and class files).

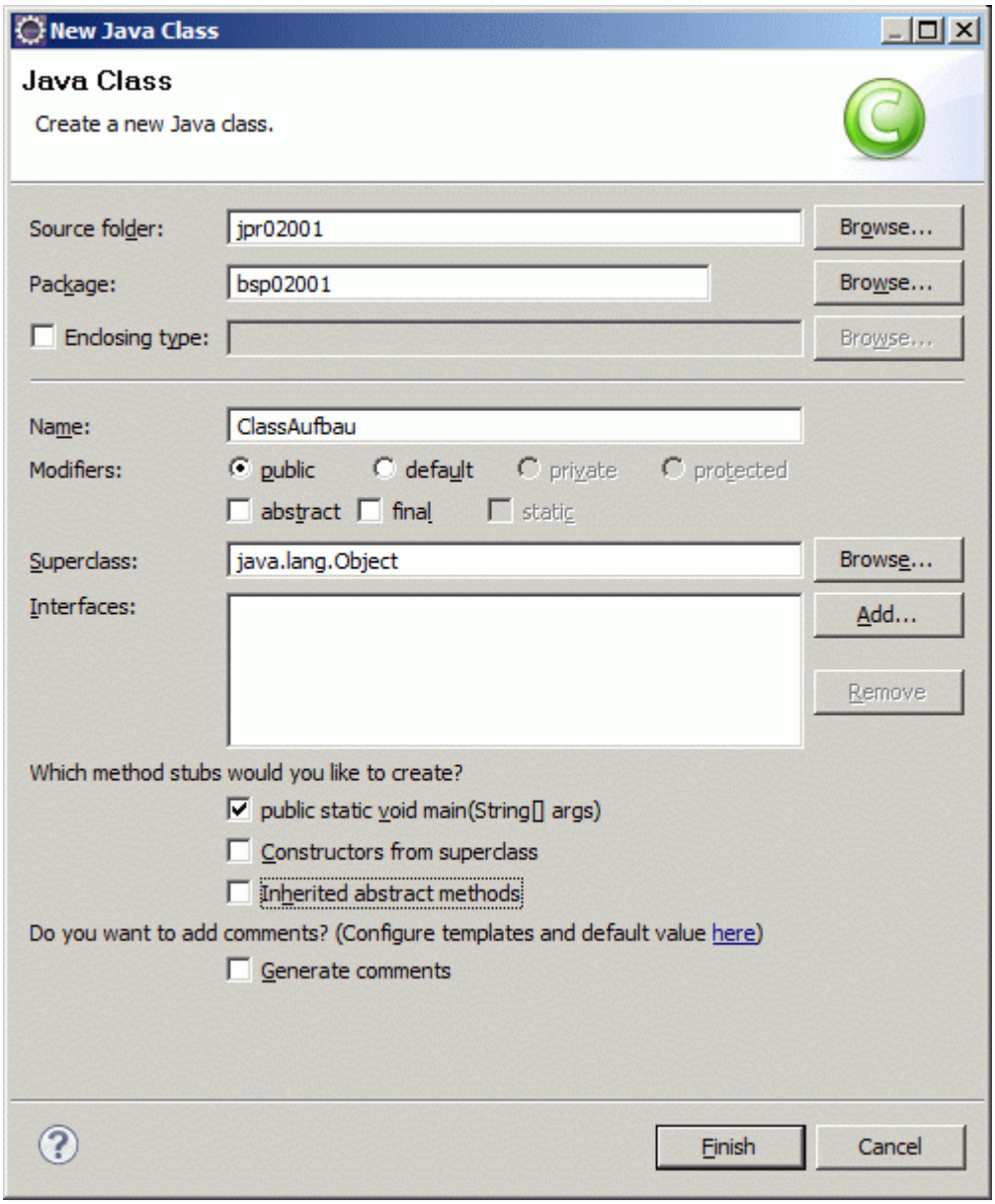
   Mit diesen Festlegungen ist die Konfiguration des Java-Projektes abgeschlossen, sie kann also mit Klick auf den Button "Finish" abgeschlossen werden:



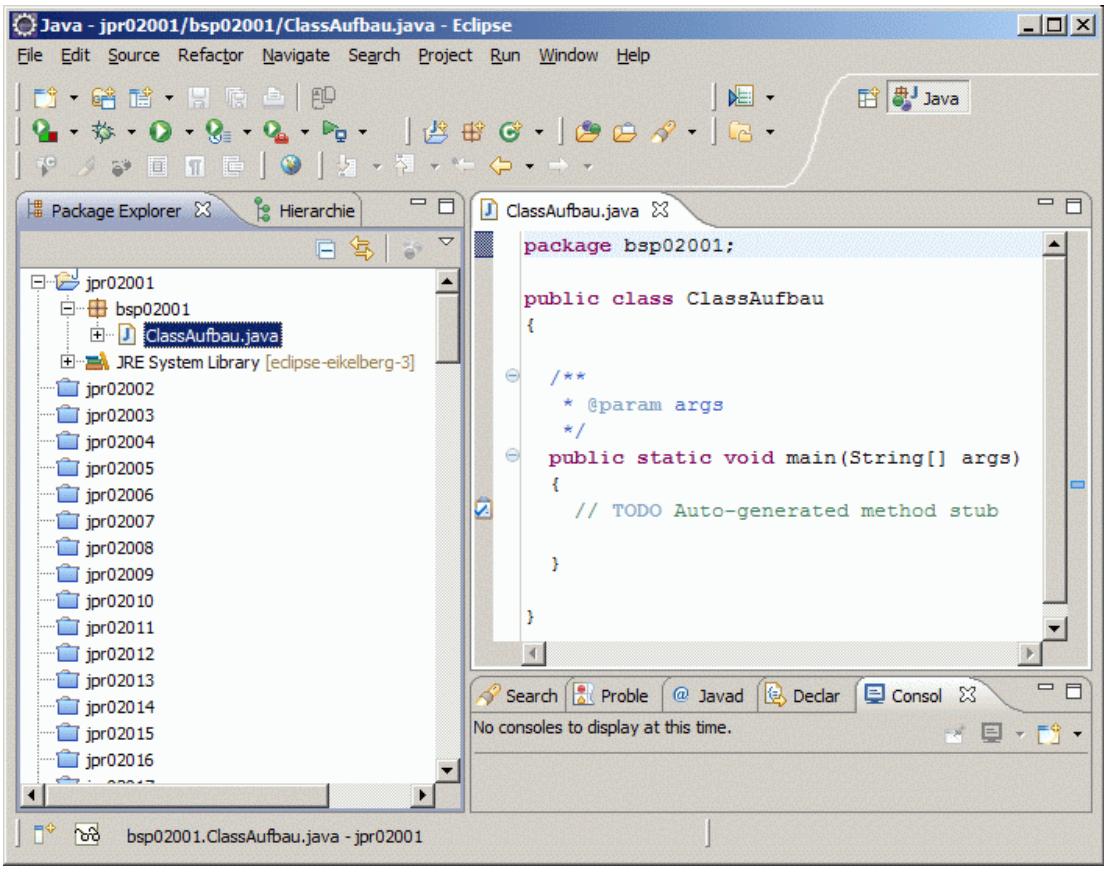
In dieses so angelegte Projekt kann man auf zwei Wegen eine Datei mit Quelltext zu einer Java-Klasse einfügen:

1. Eine neue Datei für eine Java-Klasse erstellt man in Eclipse wie folgt:
   - Man markiere im Package Explorer das Projekt, in das man die neue Quelldatei einfügen will.
   - Mit der Auswahl File-Menü / New / Class öffne man das Fenster New Java Class. Dort trage man folgende Einstellungen ein:
     - Im Feld Source Folder sollte das Java-Basisverzeichnis eingetragen sein (jpr02001).
     - In das Feld Package wird der Paketname der neuen Klasse eingetragen (bsp02001).
     - In das Feld Name wird der Name der neuen Klasse eingetragen (ClassAufbau).
     - Für die Klasse sollte nur der Modifizierer (Modifier) public ausgewält werden.
     - Ferner sollte das Anlegen von fertigem Methodenköpfen (stub) für folgende Methode aktiviert werden: public static void main(String[] args)

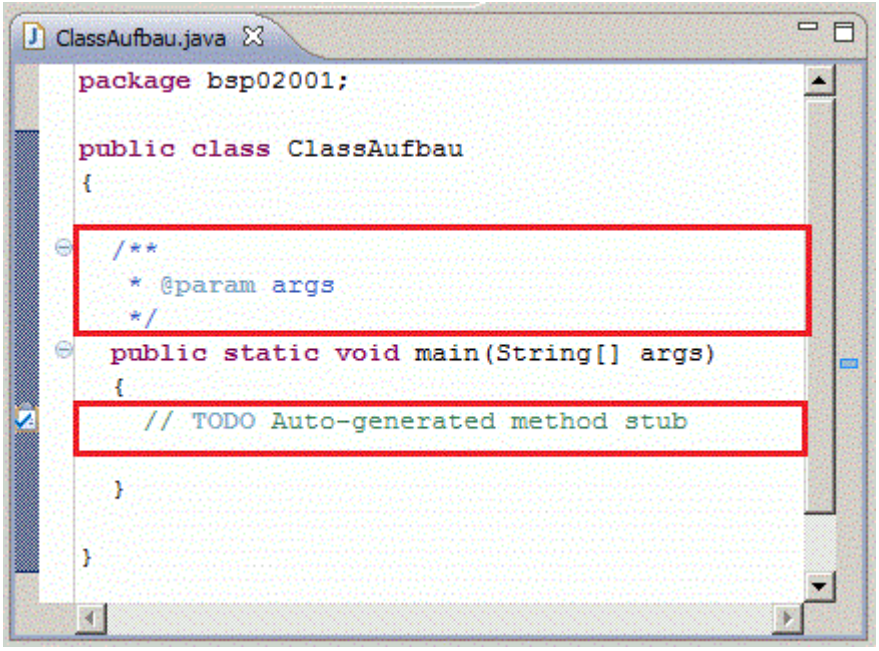
   Insgesamt ergeben sich somit folgende Einstellungen, die mit Klick auf den Button Finish bestätigt werden sollten:



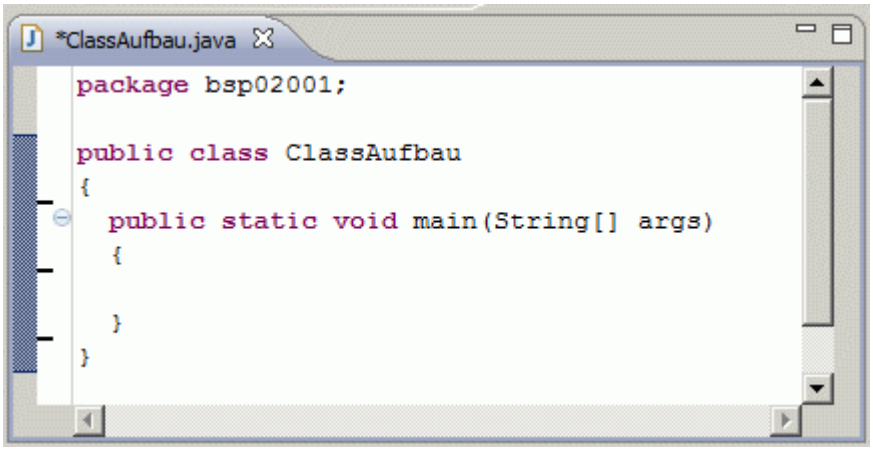
Eclipse erzeugt daraufhin die gewünschte Klasse und zeigt sie wie folgt an:



Im Package Explorer sieht man die korrekt angelegte Dateistruktur und rechts neben dem Package Explorer den erzeugten Java Quellcode der neuen Datei `ClassAufbau.java`. Der erzeugte Quellcode wird jedoch nicht vollständig benötigt. Man lösche daher stets die im folgenden Screenshot markierten Zeilen und ordne ferner die öffnenden geschweiften Klammern wie gezeigt in einer separaten Zeile an:



Nach Entfernen von Leerzeilen und Löschen der nicht benötigten Zeilen besteht unsere Klassendatei damit aus folgendem Programmcode:



2. Eine außerhalb der Eclipse Umgebung erstellte Datei kann wie folgt in ein Projekt eingefügt werden. Man kopiere die Datei zunächst in das Java-Basisverzeichnis. Anschließend markiere man im Package Explorer von Eclipse das Projekt, in das die Datei eingefügt wurde und wähle den Punkt Refresh aus dem File-Menü (oder betätige die Taste F5). Daraufhin wird die hinzugefügte Datei im Package Explorer angezeigt. Auf die gleiche Weise kann auch eine Datei aus Eclipse enfernt werden: Man beende Eclipse, lösche die Datei, starte Eclipse wieder, markiere das Projekt und betätige die Taste F5. Daraufhin wird die gelöschte Datei aus der Anzeige entfernt.

2.2.1.6 Zusammenfassung

Das einfachste Beispiel für eine Java Klasse (mit Paketbildung) ist die bereits oben erwähnte Klasse bsp02001.ClassAufbau, enthalten in der Datei `ClassAufbau.java` und compiliert in die Klasse `ClassAufbau.class`:

```
package bsp02001;

public class ClassAufbau
{

}
```

Wenn Sie bsp02001.ClassAufbau mit Eclipse übersetzen und ausführen, so wird Eclipse dieses Programm nicht als Konsolprogramm oder Hauptprogramm erkennen. Dies liegt daran, daß Klassendateien wie oben schon erwähnt neben Java-Anwendungen auch nicht als Programm lauffähige Klassen oder Applets enthalten können.

Der Begriff der Java-Klasse ist somit ein Oberbegriff für Java-Anwendungen, Applets etc. Er ist ein Begriff der objektorientierten Terminologie, dessen volle Bedeutung erst im Rahmen der objektorientierten Programmierung erkennbar wird.

Für die Kennzeichnung der Klasse ClassAufbau als Java-Anwendung fehlt insbesondere also noch ein wichtiges Element, nämlich die Kennzeichnung der Klasse als Hauptprogramm durch Einfügung der Methode **main()**. Diese Methode ist Thema des nächsten Kapitels.

2.2.2 Java-Anwendungen

2.2.2.1 Der Begriff der Java-Anwendung

Ein **Java-Anwendungsprogramm** oder auch nur kurz eine **Java-Anwendung** ist eine eigenständige und direkt von der JVM ausführbare Java-Klasse zusammen mit allen weiteren Klassen, auf die diese Klasse zugreift. Beispiele von Java-Anwendungen sind Konsolprogramme sowie GUI-Programme. Auf der anderen Seite sind Applets lediglich innerhalb eines Browsers lauffähig und daher kein Java-Anwendungsprogramm.

2.2.2.2 Die Methode main(String[])

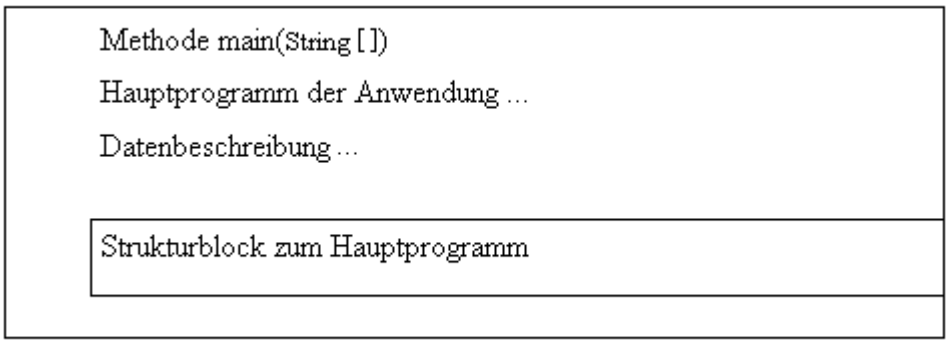
Java-Anwendungen werden gegenüber dem Compiler dadurch gekennzeichnet, daß es in der Klasse, die der JVM übergeben wird, einen Programmabschnitt gibt, der mit den Schlüsselworten

`public static void main(String[] ...)`

beginnt. Dieser Programmabschnitt muß innerhalb des mit dem Schlüsselwort `class` gekennzeichneten Abschnitts angeordnet sein. main(String[] ...) enthält die Anweisungen zu dem Modulblock, der nach Programmstart als erstes ausgeführt werden soll. Mit anderen Worten enthält der main(String[] ...)-Bereich die Anweisungen zum **Hauptprogramm-Modulblock** der Anwendung. Den Hauptprogramm-Modulblock werden wir im folgenden auch kurz als **Hauptprogramm** bezeichnen. Die das Hauptprogramm umgebende Klasse werden wir als **Startklasse** bezeichnen, da sich in dieser Klasse der Startpunkt der entsprechenden Java-Anwendung befindet.

Prinzipiell ist nach Java-Denkweise der Programmabschnitt main(String[] ...) eines Java-Programms eine Prozedur mit der besonderen Eigenschaft, daß sie ein möglicher Startpunkt des Programms ist. Im Rahmen der objektorientierten Programmierung wird statt des Begriffs Prozedur das Wort "Methode" verwendet, weshalb auch in diesem Text im Weiteren von der main-Methode die Rede ist. Jede Java-Klasse enthält maximal einen Abschnitt main(String[] ...), da es in einer Klasse niemals mehrere gleichnamige Methoden geben darf. Da das Hauptprogramm nach Java-Konzeption als spezielle Methode aufgefaßt wird, ist die Syntax (Gestaltung der Sprachelemente) von Java so angelegt, daß die main-Methode äußerlich wie jede andere Java-Prozedur gestaltet ist. Insbesondere tritt auch bei der main-Methode das für Methoden typische Klammerpaar (...) auf.

Setzt man die Sprachelemente von Java mit den in Kapitel 1.5.2 erklärten Elementen von Struktogrammen in Beziehung, so entspricht die Methode main(String[] ...) dem Modulblock zum Hauptprogramm:



Die Anweisungen der Methode main(String[] ...) - die also den Strukturblock zum Hauptprogramm realisieren - werden stets in einem Block zusammengefaßt,

1. der mit **`public static void main(String[] ...)`** **`{`** beginnt und
2. der mit einer schließenden geschweiften Klammer **`}`** endet.

Der obige Modulblock entspricht also folgendem Java Programmcode:

```
// Hauptprogramm der Anwendung ...
  public static void main(String[] argumente)
  {

// Datenbeschreibung ...

// Java Anweisungen zur Realisierung des Strukturblocks ...

  }
}
```

Das Schlüsselwort `public` muß stets eingefügt werden und erhält im Zusammenhang mit objektorientierter Programmierung seine Bedeutung. Das Schlüsselwort `static` kennzeichnet die Methode main(String[] ...) als statisch (vgl. Kapitel 2.3.1) und muß ebenfalls stets eingefügt werden. Auch das Schlüsselwort `void` muß stets erscheinen. Dieses Schlüsselwort kennzeichnet die main-Methode als "leer", d.h. die main-Methode hat keinen Rückgabewert (vgl Kap. 2.7).

2.2.2.3 Argumente der Methode main(String[] ...)

Es bleibt damit der Text `String[] argumente` zu erklären. Dieser Teil der sprachlichen Gestalt (Syntax) der Methode main(String[] ...) darf nicht fehlen. Er ermöglicht die Weitergabe von Werten an das Programm zum Zeitpunkt des Programmaufrufs und deklariert (erklärt also als Datentyp) hierzu

- einen Array (vgl. Kapitel 2.6.2)
- mit (auch anders aber gemäß den Syntaxregeln von Java zu wählendem) Namen "argumente",
- der aus Strings (Zeichenketten) (vgl. Kapitel 2.6.3)

besteht. Wie die Beispiele aus Kapitel 2.2.1 zeigen, verwendet Eclipse bei der automatischen Erstellung (Generierung) einer Startklasse statt des Namens `argumente` den Namen `args`. Die Erklärung des Textes `String[] argumente` zeigt, daß es sich hier um ein fortgeschrittenes Programmelement handelt, dessen Erklärung auf das Kapitel 2.6 verschoben werden muß. In diesem Abschnitt wird lediglich weiter unten anhand des Beispiels bsp02004.ProgErweitert die Funktionsweise erläutert.

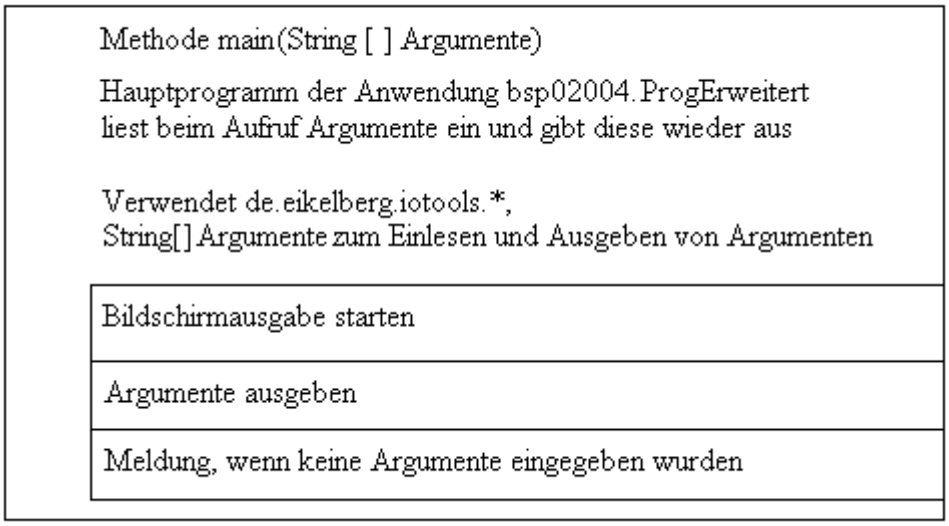
2.2.2.4 Beispiele

Ein sehr einfaches Beispiel für eine Startklasse einer Java Anwendung ist das Programm bsp02003.ProgAufbau, das ein Hauptprogramm ohne eine einzige Anweisung enthält:

```
package bsp02003;

public class ProgAufbau
{

// Hauptprogramm
  public static void main(String[] argumente)
  {

// leeres Hauptprogramm

  }
}
```

Dieses Programm ist dem Paket bsp02003 zugeordnet und enthält keine wirksame Anweisung.

Mit einem Text der Form `String[] argumente` wird wie bereits oben erwähnt die Weitergabe von Werten an das Programm zum Zeitpunkt des Programmaufrufs möglich. Diese Werte werden beim Programmaufruf in der Konsol-Box nach dem Programmnamen eingegeben und in dem durch `String[] argumente` deklarierten Bereich gespeichert. Dies sei an folgendem Beispiel bsp02004.ProgErweitert illustriert. Dieses Beispiel realisiert folgendes Struktogramm:

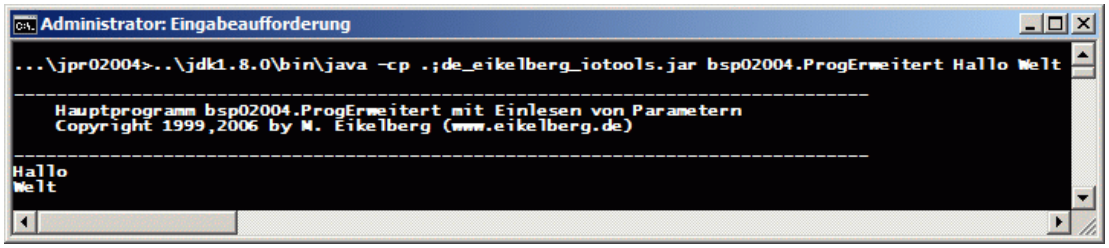


Der folgende Java-Programmcode ist eine Realisierung des obigen Stuktogramms:

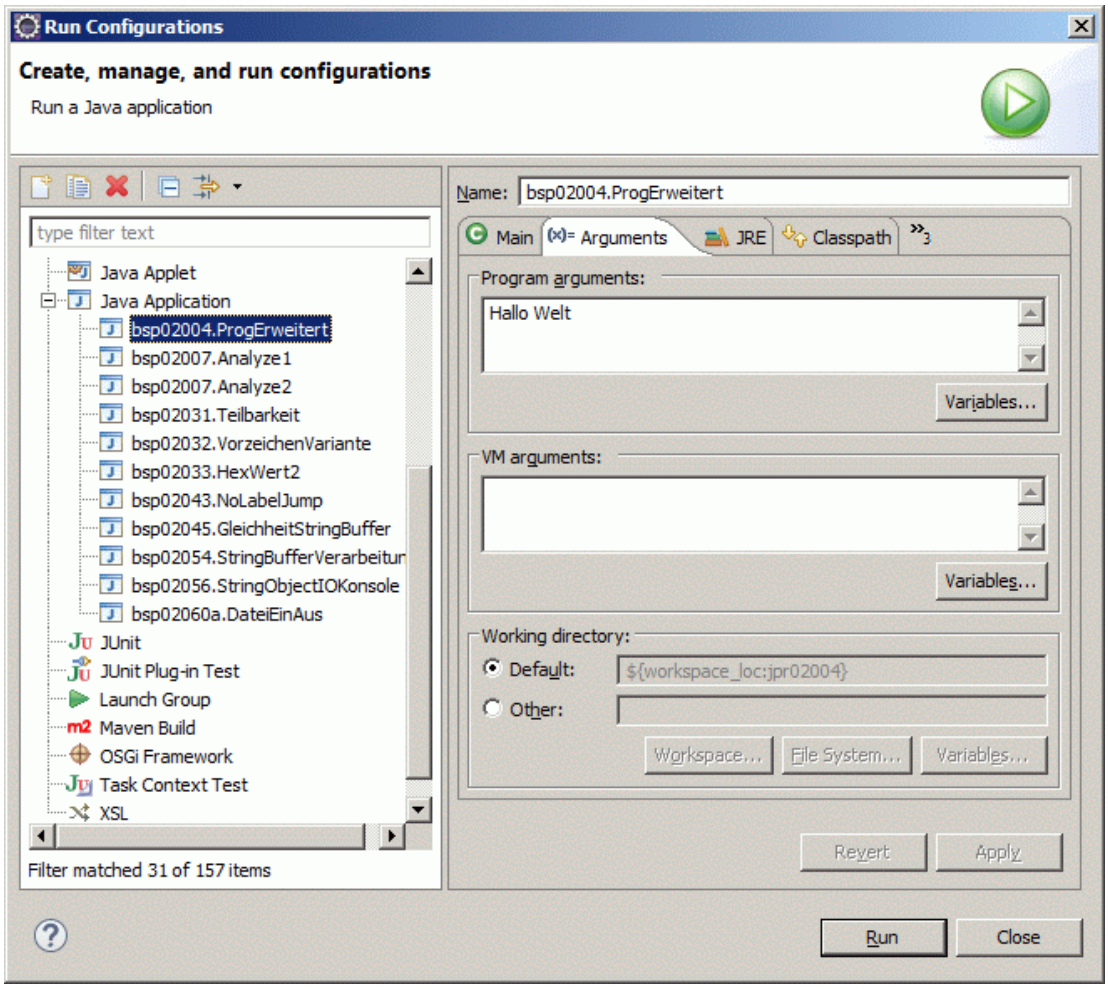
```
package bsp02004;
```

```
import de.eikelberg.iotools.*;

public class ProgErweitert
{

// Hauptprogramm
/* Programm liest beim Programmstart Parameter ein und gibt diese
   anschließend wieder aus.
*/
  public static void main(String[] argumente)
  {

// Startmeldung
    ScreenOut.kStartScreenOut(
      "Hauptprogramm bsp02004.ProgErweitert mit Einlesen von Parametern",
      "M. Eikelberg (www.eikelberg.de)",
      "1999,2006");

// Ausgabe der Argumente
    for (int i=0;i<argumente.length;i++)
      ScreenOut.kPrintln(argumente[i]);

// Meldung, wenn kein Argument eingegeben wurde
    if (argumente.length == 0)
      ScreenOut.kPrintln("Programm wurde ohne Parameter aufgerufen !");
  }
}
```

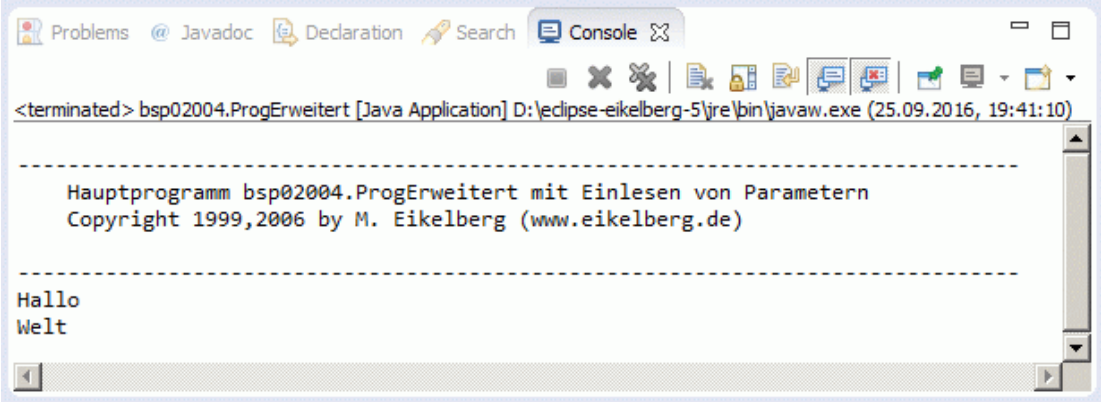
Den Bildschirmausdruck eines typischen Programmablaufs bei Verwendung der Java Virtual Machine zeigt der folgende Screenshot:



Bei einem Start von bsp02004.ProgErweitert innerhalb der Eclipse Umgebung werden die Argumente zuvor in dem Fenster "create, manage and run configurations" als Anwendungsparameter eingetragen. In das Fenster "create, manage and run configurations" gelangt man über die Auswahl Run-Menü / Run. Daraufhin öffnet sich das Fenster "create, manage and run configurations". Man erweitere bei Bedarf im linken Teil des Fensters den Zweig Java Application und markiere die Konfiguration zum jeweiligen Java-Projekt. Anschließend wechsle man im rechten Teil des Fensters in die Registerkarte "(*)=Arguments" und trage dort die Argumente (Program arguments) ein, die beim Programmstart verwendet werden sollen. Für das vorliegende Beispiel sieht das Fenster "create, manage and run configurations" nach erfolgter Eintragung wie folgt aus:



Die hinter dem Programmaufruf direkt in der Eingabezeile bzw. im Fenster "create, manage and run configurations" eingegebenen Worte 'Hallo Welt' werden beim Programmstart auf dem Array argumente gespeichert und anschließend mit Hilfe des Befehls `ScreenOut.kPrintln(...)` wieder ausgegeben. Dieser Befehl stammt aus dem bereits in Kapitel 2.2.1 kurz erwähnten Java-Paket `de.eikelberg.iotools` und führt die Ausgabe auf dem Bildschirm durch. Das Paket `de.eikelberg.iotools` und seine Nutzung für die vereinfachte Tastatureingabe und Bildschirmausgabe ist Thema von Kapitel 2.3.



2.2.3 Unicode und Codierung unter Windows

2.2.3.1 Betriebssysteme und Codierungen

Für jedes Betriebssystem gibt es mindestens eine Tabelle, gemäß der bei der Programmausgabe die auf der Festplatte oder im Hauptspeicher gespeicherten Bytes bei Bedarf in Buchstaben umgesetzt werden. Diese Tabelle heißt **Codierungstabelle** bzw. kurz **Codierung**. Diese Codierung wird natürlich auch beim Einlesen von Buchstaben verwendet.

Grundsätzlich wurde bei der Entwicklung von Java davon ausgegangen, dass das zugrundeliegende Betriebssystem **Unicode-Codierung** verwendet, also die Schriftzeichen bei der Ein- und Ausgabe gemäß der Unicode-Tabelle codiert und decodiert. Die Unicode-Tabelle ist die Codierung der Unix Betriebssysteme. Auch bei der Speicherung von Programmcode wird von einer Unicode Codierung des Programmcodes ausgegangen.

Verwendet das Betriebssystem, auf dem ein Java-Programm läft, eine andere Codierung als die Unicode-Codierung, so kann es Probleme bei der Ein- und Ausgabe von Daten geben und in Einzelfällen auch bei der Übertragung von Java Quelltext auf andere Betriebssysteme geben. Gleiches gilt, wenn andere Programme - zum Beispiel Editoren oder die Konsolbox - keine Unicode-Codierung verwenden. Diese Probleme treten auf jeden Fall unter Windows-Betriebssystemen auf. Zur Reduktion der in diesem Fall auftretenden Probleme kann man allerdings mit den Sprachmitteln von Java geeignete Vorkehrungen treffen.

Verwendet man Java auf Windows-Rechnern, so ist man in doppelter Weise von der Problematik betroffen:

- Bei der Verarbeitung von Schriftzeichen über Programme der Windows-Oberfläche (z. Bsp. notepad.exe / Editor) verwendet die deutsche Windows-Version die **Codierung gemäß Code Page 1252**.
- Bei der Ein- und Ausgabe in der Konsolbox sowie bei der Verwendung von DOS-Programmen (z. Bsp. Konsolbefehle echo und type) verwendet die deutsche Windows-Version die **Codierung gemäß Code Page 850**.
- Einzelne Windows-Programme (z. Bsp. der Editor notepad.exe oder der Internet Explorer) sind auch in der Lage, die **Unicode-Codierung** zu verwenden.

Die genannten Code Pages gelten für den deutschsprachigen Raum. Im Rahmen der **Lokalisierung** von Windows - also der Anpassung an den jeweiligen Sprach- und Kulturraum, in dem die jeweilige Windows-Version verwendet werden soll - werden viele unterschiedliche Code Pages verwendet, um zum Beispiel nationale Sonderzeichen wie die deutschen Sonderzeichen ä,ö,ü,Ä,Ö,Ü und ß in Windows verfügbar zu machen.

Die Java-Programmierung unter Windows ist also vom Problem unterschiedlicher Codierungen betroffen. Java bietet jedoch zweckmäßige und wohldurchdachte Sprachmittel an, die zur Lösung dieses Problems eingesetzt werden können. Jedoch fällt die Nutzung der entsprechenden Techniken in den Bereich der fortgeschrittenen Programmierung, so daß der erforderliche Programmcode für diesen Kurs zunächst im Paket `de.eikelberg.iotools` bereitgestellt wird. Dieses Paket und seine Nutzung wird in Kapitel 2.3 genauer erklärt, hier soll zunächst die Problematik erläutert werden, da ihre Kenntnis für den Programmierer sehr wichtig ist.

2.2.3.2 ASCII Zeichensätze

Wie im Grundlagen-Teil der Vorlesung erklärt werden Schriftzeichen, also Buchstaben, Zahlen und Sonderzeichen auf einem Rechner nach unterschiedlichen Regeln in Bit-Folgen übersetzt. Eine sehr weit verbreitete Codierungsregel beinhaltet der sogenannte ASCII-Zeichensatz, der - nicht nur im DOS-Bereich - von großer Bedeutung ist. Den ASCII-Zeichensatz gibt es in zwei Varianten:

- **7-Bit ASCII** besteht aus den 128 Zeichen der oberen 8 Reihen der nachfolgenden Tabelle und ist normiert.
- **8-Bit ASCII**, auch **erweiterter ASCII-Zeichensatz** bzw. **erweiterter ASCII-Code** genannt, besteht aus allen 256 Zeichen der nachfolgenden Tabelle. Es gibt jedoch mehrere nationale Varianten des 8-Bit ASCII Zeichensatzes, so dass die Umsetzung je nach Konfiguration eines Rechners variieren kann.

<u>8 Bit ASCII Code:</u>

rechtes Halbbyte

linkes Halbbyte	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
F	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	⌂

Tabelle nach: Horn/Kerner, Band 1, Seite 198. Leere Kästchen stehen für nicht druckbare Zeichen (Steuerzeichen)

2.2.3.3 Erweiterung des 7-Bit ASCII Zeichensatzes durch DOS Code Pages

Das wesentliche Problem der Erweiterungen des 7-Bit ASCII Zeichensatzes ist die unterschiedliche Normierung der Zeichen 128 bis 255 in diesen Erweiterungen. Unter anderem auch zur "Lokalisierung" der Betriebssysteme, also zur Anpassung an den jeweiligen kulturellen Bereich, in dem das Betriebssystem eingesetzt wird, ist in unterschiedlicher Weise eine Erweiterung des 7-Bit ASCII zu einem 8 Bit Zeichensatz erfolgt. Neben dem erweiterten ASCII-Code gibt es weitere Codierungen unterschiedlicher Hersteller und Institutionen, die sehr weit verbreitet sind und die *für die druckbaren Zeichen* des 7-Bit ASCII-Code mit diesen Zeichen übereinstimmen, im Bereich der nicht druckbaren Zeichen des 7-Bit ASCII Codes sowie im Bereich der Zeichen 128 bis 255 jedoch Erweiterungen bzw. Unterschiede aufweisen. Eine wichtige derartige Erweiterung ist die Microsoft Code Page 437, die gleich dem IBM-Zeichensatz 437 ist (Code Page = Zeichensatz):

Microsoft DOS Code Page 437 / IBM-Zeichensatz 437

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-		8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p		Ç	É	á	░	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q		ü	æ	í	▒	┴	╤	β	±
-2	☻	↕	"	2	B	R	b	r		é	Æ	ó	▓	┬	╥	Γ	≥
-3	♥	!!	#	3	C	S	c	s		â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t		ä	ö	ñ	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u		à	ò	Ñ	╡	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v		å	û	ª	╢	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w		ç	ù	º	╖	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x		ê	ÿ	¿	╕	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y		ë	Ö	⌐	╣	╔	┘	Θ	∙
-A	◙	→	*	:	J	Z	j	z		è	Ü	¬	║	╩	┌	Ω	·
-B	♂	←	+	;	K	[	k	{		ï	¢	½	╗	╦	█	δ	√
-C	♀	∟	,	<	L	\	l	|		î	£	¼	╝	╠	▄	∞	ⁿ
-D	♪	↔	-	=	M	]	m	}		ì	¥	¡	╜	═	▌	φ	²
-E	♫	▲	.	>	N	^	n	~		Ä	Pt	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂		Å	ƒ	»	┐	╧	▀	∩	

Tabelle aus: Jordan, Rüdiger; Kampmann, Wilfried, Einführung in die Datenverarbeitung und das Programmieren in Fortran 77, S.25. Man beachte, dass diese Tabelle die Zeichen nicht zeilenweise, sondern spaltenweise enthält!

Das Betriebssystems MS-DOS - die folgenden Ausführungen beziehen sich auf die letzte Version MS-DOS 6.22 - verwendet je nach Sprachversion insgesamt folgende Code Pages:

Nummer	Schriftzeichen für den Sprachraum
437	Amerikanisch
850	Mehrsprachig (Westeuropäisch)
852	Slawisch
860	Portugiesisch
863	Frankokanadisch
865	Norwegisch-Dänisch

In der deutschsprachigen Version von MS-DOS 6.22 stehen die Code Pages 437 und 850 zur Verfügung. Wird zum Beispiel ein EGA-Bildschirm verwendet, der hardwaremäßig (nur) die Zeichentabelle 437 darstellen kann, und soll die - im deutschen Bereich übliche - Code Page 850 bei der Ausgabe in der Konsolbox (con = Konsole) verwendet werden, so stehen in der Datei **config.sys** einer MS-DOS-Installation folgende Einträge:

```
country=049,,c:\dos\country.sys
device=c:\dos\device.sys con=(EGA,437,1)
```

Dadurch werden im Betriebssystem folgende Einstellungen vorgenommen:

- Der **Landescode** ist **049**, d.h. das Betriebssystem wird in Deutschland installiert (**049** entspricht der Telefonvorwahl ++49 vom Ausland nach Deutschland).
- Es ist ein (DOS Konsol-)Bildschirm (**con** = Konsole) installiert, der **EGA**-fähig ist und in dessen Hardware die Code Page **437** eingebaut ist. Es wird eine (**1**) Code Page für den Bildschirm geladen.

In der Datei **autoexec.bat** sollten entsprechend folgende Einträge eingefügt sein:

```
nlsfunc
mode con cp prep=((850)c:\dos\ega.cpi)
chcp 850
keyb gr,,c:\dos\keyboard.sys
```

Diese Anweisungen haben folgende Wirkung:

- Es wird das Programm NLSFUNC geladen, das ein Erkennen und Wechseln von Zeichensätzen ermöglicht.
- Der Bildschirm (**con**) wird eingestellt auf die Verwendung von Code Page 850 sowie Zeichen für die Bildschirmauflösung EGA. Die EGA und VGA-Zeichensätze sind in der Datei ega.cpi gespeichert.
- Der Befehl chcp ändert die aktive Code Page ab, in diesem Fall auf die Code Page 850.
- Der Befehl keyb sorgt für die Einstellungen passend zur deutschen (**gr** für Germany) Tastatur.

Der bereits oben gezeigte Zeichensatz 437 ist sehr häufig der bereits in der Hardware vorhandene (amerikanische) Zeichensatz. In deutschen MS-DOS-Systemen (und in den Konsolboxen von Windows) wird jedoch in der Regel der Zeichensatz 850 verwendet. Diesen Zeichensatz zeigt die folgende Tabelle:

Microsoft DOS Code Page 850 (westeuropäisch)

Die folgende Tabelle ist zeilenweise organisiert, das Zeichen NM befindet sich also in der N-ten Zeile und der M-ten Spalte

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
1	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	⌂
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»
B	░	▒	▓	│	┤	Á	Â	À	©	╣	║	╗	╝	¢	¥	┐
C	└	┴	┬	├	─	┼	ã	Ã	╚	╔	╩	╦	╠	=	╬	¤
D	ð	Ð	Ê	Ë	È	ı	Í	Î	Ï	┘	┌	█	▄	¦	Ì	▀
E	Ó	ß	Ô	Ò	õ	Õ	µ	þ	Þ	Ú	Û	Ù	ý	Ý	¯	´
F		±	‗	¾	¶	§	÷	¸	°	¨	·	¹	³	²	■	

2.2.3.4 Codierung unter Windows

Wegen der unterschiedlichen Zeichensätze der verschiedenen Betriebssysteme und Betriebssystemversionen ist es also nicht gesichert, dass Dateien oder Daten auf allen Rechnern gleich angezeigt werden. Es kann sogar - schlimmer noch - vorkommen, dass ein Rechner je nach Programm und Situation *unterschiedliche Zeichensätze* verwendet. Dies ist **bei Windows-Rechnern** der Fall:

1. Bei Ein- und Ausgabe über die Eingabeaufforderung bzw. Konsolbox von Windows wird nach **Code Page 850** ver- bzw. entschlüsselt.
2. Bei Ein- und Ausgabe über ein Windows-Programm verwendet Windows die **Code Page 1252**
3. Einzelne Windows-Programme (z. Bsp. Editor notepad.exe und der Internet Explorer iexplore.exe) sind in der Lage, auch mit der **Unicode-Tabelle** (vgl. Kap. 2.2.3.7) zu arbeiten.
4. Java-Programme bzw. die Java Virtual Machine verwenden unter Windows ab Java-Version 1.2 die Codierung **ISO 8859-1** (Teilmenge von Unicode, vgl. Kap. 2.2.3.7).
    - ISO 8859-1 und Code Page 1252 sind identisch für die Zeichen 0 - 127 sowie 160 - 255.
    - ISO 8859-1 enthält im Bereich der Zeichen 128-159 nicht durckbare Zeichen - für die Java das Zeichen 0x3F / 63 / '?' ausgibt - und ist unterschiedlich zu Code Page 1252

Die obige Aufstellung gilt für den deutschen Sprachraum.

Dieses Phänomen führt zu deutlich sichtbaren Problemen bei der Ein- und Ausgabe von Daten in einem Java-Programm, wenn die verwendeten Schriftzeichen nicht im 7-Bit ASCII-Code enthalten sind. Diese Probleme treten insbesondere auch deshalb auf, weil die deutschen Sonderzeichen ä,ö,ü,Ä,Ö,Ü und ß als nationale Besonderheit nicht im 7-Bit ASCII Code enthalten sind. Zur Illustration der Situation folgen in 2.2.3.8 einige Beispiele.

Die Codierung für Windows-Programme auf deutschen Windows-Systemen gemäß Code Page 1252 erfolgt entsprechend der folgenden Tabelle. Die rechteckigen Rahmen in der Tabelle wie zum Beispiel zu dem Zeichen 0x00 stehen für nicht druckbare Zeichen.

<u>Microsoft Windows Code Page 1252 (westeuropäisch)</u>

Die folgende Tabelle ist zeilenweise organisiert, das Zeichen NM befindet sich also in der N-ten Zeile und der M-ten Spalte

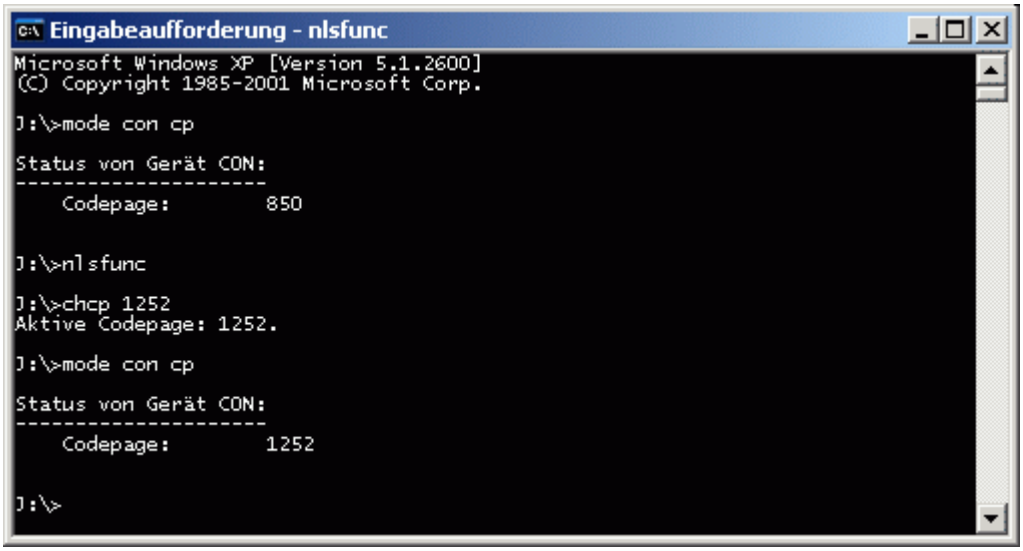
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	□
8	€	□	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	□	Ž	□
9	□	‘	’	“	”	•	–	—	˜	™	š	›	œ	□	ž	Ÿ
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	•	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

2.2.3.5 Exkurs - Einstellen des Zeichensatzes in der Konsolbox von Windows XP bzw. Windows 7

Für die Arbeit in der Konsolbox von Windows kann es interessant sein, die von der Konsolbox verwendete Code Page abzuändern. Diese Änderung geschieht für Windows XP und Windows 7 in leicht unterschiedlicher Weise.

Für **Windows XP** funktioniert das Einstellen des Zeichensatzes nicht für alle in der Konsolbox verwendbaren Schriftarten. Umgesetzt wird die Änderung oft nur dann, wenn man für die Konsolbox - über "Eigenschaften von Eingabeaufforderung" - den True Type (TT) Schrifttyp "Lucida Console" wählt.

Die Veränderung und Kontrolle der verwendeten Code Page geschieht mit den im folgenden Bildschirmausdruck gezeigten Befehlen:

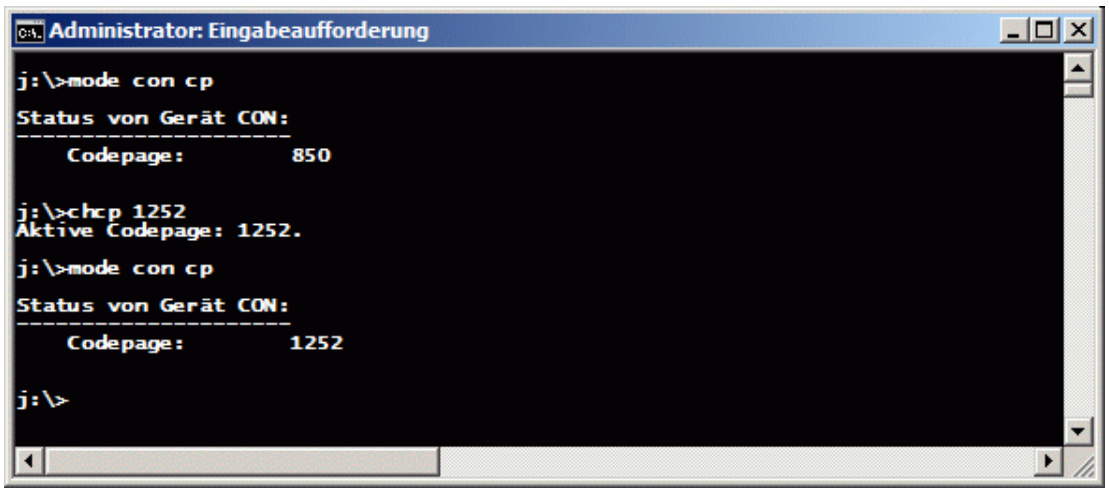


Die Befehle im Bildschirmausdruck haben folgende Bedeutung:

- Der Befehl mode con cp führt zur Anzeige des für die Konsolbox (**con**) verwendeten Zeichensatzes.
- nlsfunc lädt das bereits in 2.2.3.3 erklärte Programm NLSFUNC, das ein Erkennen und Wechseln von Zeichensätzen ermöglicht.
- chcp ändert den Zeichensatz ab auf den Windows-Zeichensatz 1252 (westeuropäisch).

Man beachte, dass die so vorgenommenen Änderungen der Codierung nur für die jeweilige Konsolbox gelten und mit dem Schließen der Konsolbox zurückgesetzt werden. Man beachte ferner, dass in der Konsolbox einer deutschen Windows-Version die deutsche MS-DOS Code Page 850 verwendet wird, obwohl ein deutsches Windows selbst die Code Page 1252 (westeuropäisch) verwendet. Auf diesen Punkt wird noch weiter unten eingegangen. Wir zeigen hier lediglich bereits an einem einfachen Beispiel, welche Folgen dies hat.

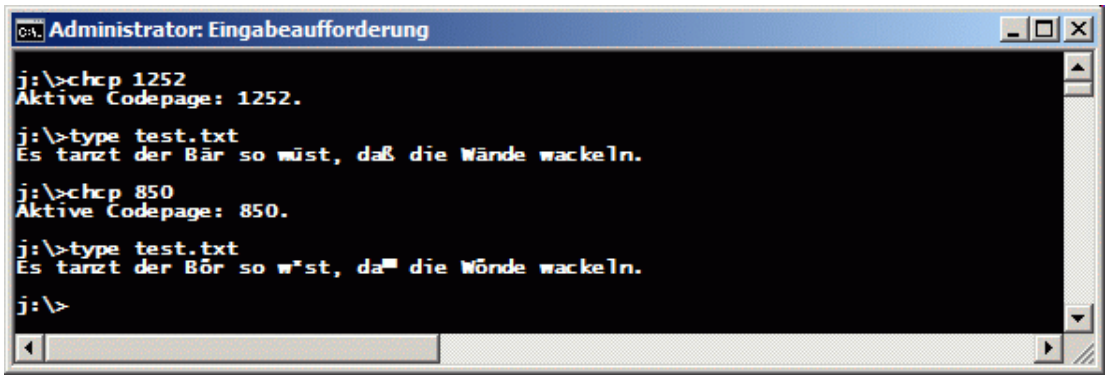
Unter **Windows 7** ist der Befehl `nlsfunc` nicht mehr verfügbar, er wird allerdings auch nicht mehr benötigt. Für dieses Betriebssystem kann der Zeichensatz direkt mit dem Befehl `chcp` eingestellt werden:



Als Beispiel für die Wirkung dieser Befehle wird (unter Windows 7) mit dem Windows-Programm "Editor" (notepad.exe) eine Datei mit Namen test.txt angelegt, die folgenden Satz enthält:

`Es tanzt der Bär so wüst, daß die Wände wackeln.`

Wir stellen die Codierung der Konsolbox zunächst auf die Windows-Codierung 1252 ein und drucken den Inhalt der Datei test.txt in der Konsolbox aus. Anschließend stellen wir wieder die Standard-Codierung 850 der Konsolbox ein und drucken den Inhalt von test.txt erneut aus. Wir erhalten dann folgende Anzeige in der Konsolbox:



Auf die hier sichtbar werdenden Probleme wird ab 2.2.3.7 eingegangen.

2.2.3.6 Weitere Erweiterungen des 7 Bit ASCII Zeichensatzes

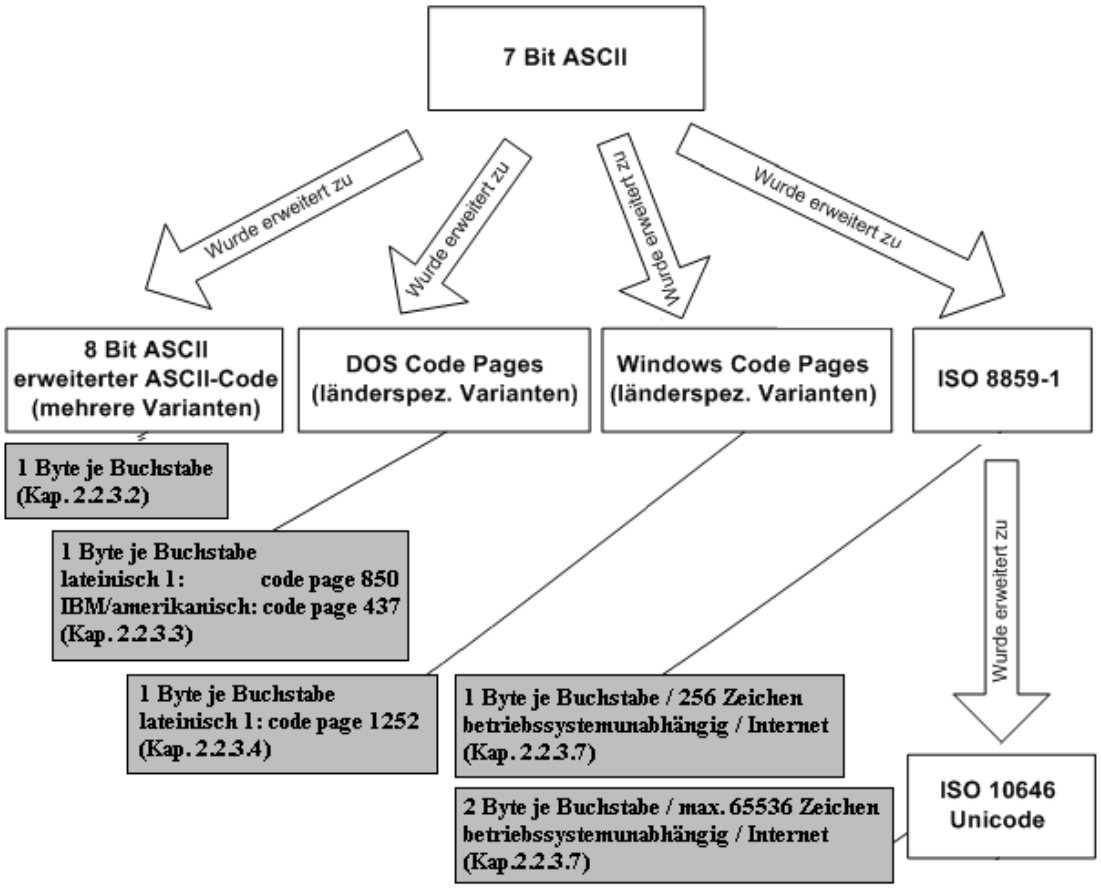
Neben den DOS und Windows Code Pages gibt es weitere Codierungen, die den 7-Bit ASCII Code erweitern. Dabei ist die Erweiterung im Bereich der ersten 128 Zeichen praktisch ohne Bedeutung, hier werden lediglich nicht druckbare ASCII-Zeichen mit zusätzlichen Zeichen für die Darstellung auf dem Bildschirm, in einer Datei oder auf dem Ausdruck versehen.

Die Erweiterung im Bereich der Zeichen größer hexadezimal x007F - also größer als dezimal 127 - kann jedoch dazu führen, dass solche Zeichen auf verschiedenen Rechnern in verschiedenen Ländern unter verschiedenen Programmen unterschiedlich angezeigt werden.

Für den deutschsprachigen Bereich sind neben dem 'reinen' 8-Bit ASCII gemäß 2.2.3.2 insgesamt die folgenden ASCII-Erweiterungen von Interesse:

- Code Page 850 (CP850) - MS-DOS, DOS-Programme und Konsolbox/Eingabeaufforderung von Microsoft Windows (DOS westeuropäisch)
- Code Page 1252 (CP1252) - Microsoft Windows und alle Windows-Programme (Windows westeuropäisch)

- ISO 8859-1 (Latin-1)
- ISO 10646 (Unicode)



In Bezug auf Übereinstimmungen und Unterschiede gilt Folgendes:

- Für die Zeichen 0 bis 127 umfassen alle vier genannten Zeichensätze die druckbaren Zeichen des 7-Bit ASCII-Zeichensatzes und stimmen für diese Zeichen mit dem 7-Bit ASCII Zeichensatz überein.
- Für die Zeichen 128 bis 255 stimmen ISO 8859-1 und ISO 10646/Unicode überein.
- Für die Zeichen 128 bis 255 gibt es keinerlei Übereinstimmung vom CP850 mit ISO 10646/Unicode.
- Für die Zeichen 128 bis 159 gibt es keine Übereistimmung zwischen CP1252 und ISO 10646/Unicode, für die Zeichen 160 bis 255 stimmen CP1252 und Unicode überein.

Zeichenbereich	Code Page 850	Code page 1252	ISO 8859-1	ISO 10646
0-127	**Zeichensätze identisch**			
128-159	Ç ü é â ä à å ç ê ë è ï î ì Ä Å É æ Æ ô ö ò û ù ÿ ö Ü ø £ Ø × ƒ	€ ‚ ƒ „ … † ‡ ˆ ‰ Š ‹ Œ Ž ‘ ’ “ ” • – — ™ š › œ Ž Ÿ	**identische Zeichen** (nicht druckbar)	
160-255	Code Page 850 spezifische Zeichen	**identische Zeichen**		
256-65535	keine Zeichen			Zeichen lt. Unicode-Tabelle

2.2.3.7 Der Unicode Zeichensatz - ISO 8859-1 und ISO 10646

Prinzipiell basiert Java auf der **Unicode** Verschlüsselung. Daher soll der Unicode-Zeichensatz Basis der weiteren Überlegungen sein. Der Unicode-Zeichensatz ist ein 16-Bit Zeichensatz und bietet damit Platz für maximal 65536 Zeichen - mehr als genug, so dass noch nicht alle Schriftzeichen festgelegt sind. Die ersten 256 Zeichen der Unicode-Verschlüsselung sind gleich den Zeichen 0 bis 255 der Verschlüsselung nach ISO 8859-1 und können aus folgender Tabelle entnommen werden:

Codierung ISO 8859-1 (Zeichen 0 bis 255 Codierung ISO 10646)

	000	001	002	003	004	005	006	007	008	009	00A	00B	00C	00D	00E	00F
0	NUL	DLE	SP	0	@	P	`	p	CTRL	CTRL	NB SP	°	À	Ð	à	ð
1	STX	DC1	!	1	A	Q	a	q	CTRL	CTRL	¡	±	Á	Ñ	á	ñ
2	SOT	DC2	"	2	B	R	b	r	CTRL	CTRL	¢	2	Â	Ò	â	ò
3	ETX	DC3	#	3	C	S	c	s	CTRL	CTRL	£	3	Ã	Ó	ã	ó
4	EOT	DC4	$	4	D	T	d	t	CTRL	CTRL	¤	´	Ä	Ô	ä	ô
5	ENQ	NAK	%	5	E	U	e	u	CTRL	CTRL	¥	µ	Å	Õ	å	õ
6	ACK	SYN	&	6	F	V	f	v	CTRL	CTRL	¦	¶	Æ	Ö	æ	ö
7	BEL	ETB	'	7	G	W	g	w	CTRL	CTRL	§	·	Ç	×	ç	÷
8	BS	CAN	(	8	H	X	h	x	CTRL	CTRL	¨	¸	È	Ø	è	ø
9	HT	EM	)	9	I	Y	i	y	CTRL	CTRL	©	1	É	Ù	é	ù
A	LF	SUB	*	:	J	Z	j	z	CTRL	CTRL	ª	º	Ê	Ú	ê	ú
B	VT	ESC	+	;	K	[	k	{	CTRL	CTRL	«	»	Ë	Û	ë	û
C	FF	FS	,	<	L	\	l	|	CTRL	CTRL	¬	¼	Ì	Ü	ì	ü
D	CR	GS	-	=	M	]	m	}	CTRL	CTRL	-	½	Í	Ý	í	ý
E	SO	RS	.	>	N	^	n	~	CTRL	CTRL	®	¾	Î	Þ	î	þ
F	SI	US	/	?	O	_	o	DEL	CTRL	CTRL	¯	¿	Ï	ß	ï	ÿ

Die obige Tabelle ist zusammengesetzt aus zwei Tabellen der Unicode Inc. zur Unicode-Verschlüsselung. Diese sowie weitere Tabellen finden Sie, wenn Sie dem Link zum **Überblick über die wichtigsten Zeichen der Codierung Unicode 2.1** folgen.

Für die Java-Programmierung ist es jetzt wichtig, die Unterschiede zwischen Unicode und den anderen gebräuchlichen Codierungen zu kennen. Diese Unterschiede sind in **Dokumenten auf dem FTP-Server der Unicode Inc.** beschrieben. Dort finden Sie auch Aufstellungen der Unterschiede zwischen Unicode und weiteren Codierungen der verschiedenen Hersteller wie zum Beispiel Apple und IBM. Die Gegenüberstellungen von Unicode mit CP850, CP1252 und ISO 8859-1 finden Sie in folgenden Aufstellungen:

- **Unicode vs. Code Page 850 (CP850) - MS-DOS und Konsolfenster von Windows (DOS westeuropäisch)**
- **Unicode vs. Code Page 1252 (CP1252) - Microsoft Windows (Windows westeuropäisch)**
- **Unicode vs. ISO 8859-1 (Latin-1)**

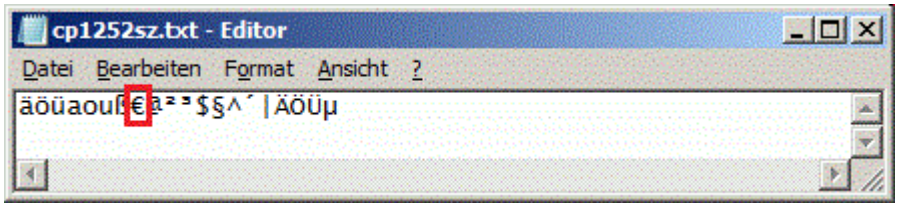
2.2.3.8 Beispiel zur Codierung unter Windows

Mit Hilfe des Programms "Hacker's View" (HIEW), das Sie auf dem FTP-Server PC132W.bsnw.fb4.fh-bochum.de und auf der DVD finden, können die in einer Datei gespeicherten Bits angesehen und ihre hexadezimale Darstellung angezeigt werden. Angewendet auf die Datei **cp1252sz.txt** erhält man unter Windows 7 folgende Ansicht:

Hinweis: Hinweise zur Installation und zur Verwendung von HIEW finden Sie im Anhang in der **Anleitung zur Installation und Verwendung des Programms HIEW**.

Lässt man sich diese Datei mit dem Konsolbefehl type im Konsolfenster von Windows 7 anzeigen, so erhält man folgende Ansicht:

Bei der Betrachtung der Datei mit dem Windows Editor NOTEPAD erhält man dagegen folgende Schriftzeichen:



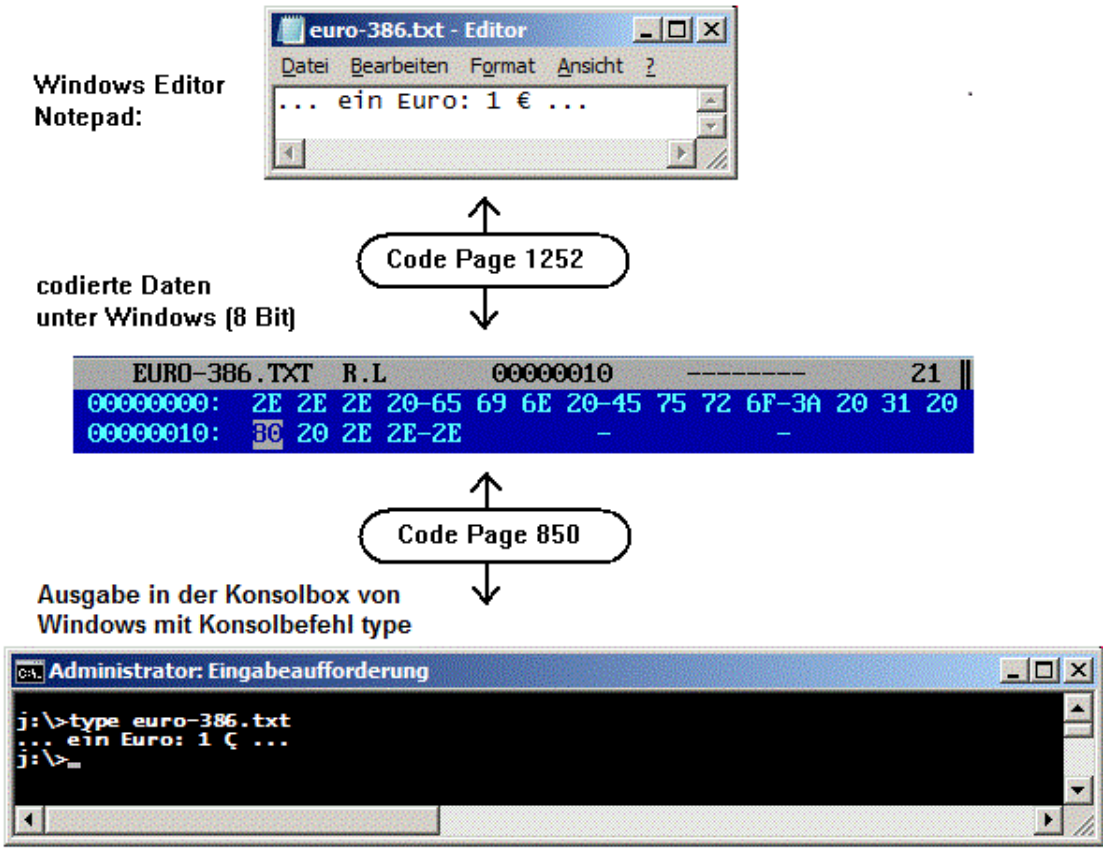
Wir schauen uns jetzt zunächst das 8. Zeichen der Datei an, es hat den hexadezimalen Wert x0080. Im Bereich x0080 bis x009F unterschieden sich sowohl CP850 als auch CP1252 von Unicode,. ISO 8859-1 ist in diesem Bereich mit Unicode identisch. Man erhält zum hexadezimalen Wert x0080 folgende Zeichen:

- Unicode / ISO 8859-1: nicht druckbares Steuerzeichen
- Code Page 850: Ç (Unicode Zeichen x00C7)
- Code Page 1252: € (Unicode Zeichen x20AC)

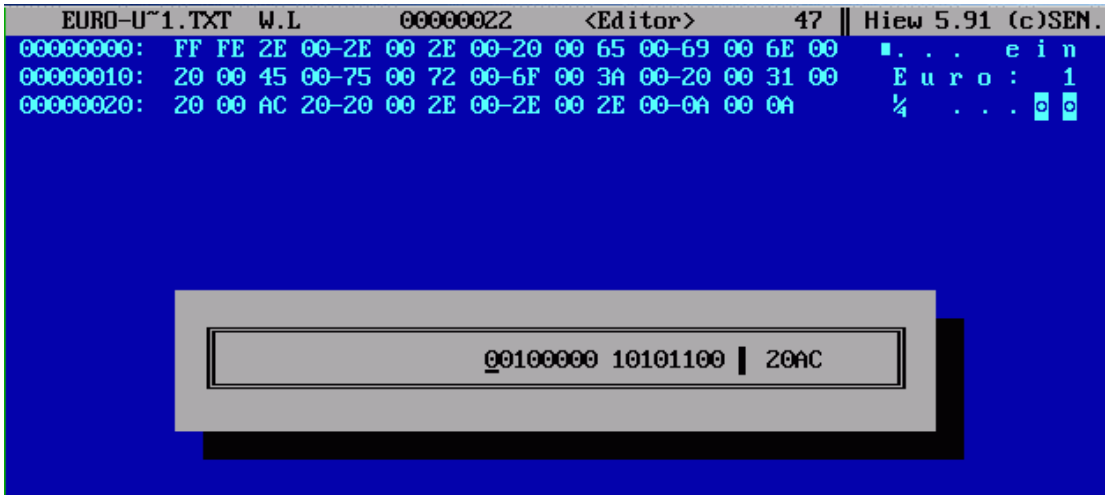
Dementsprechend zeigt der Konsolbefehl type in der Konsole für das 8. Zeichen das Zeichen Ç an, das Windows-Programm NOTEPAD jedoch das Euro-Zeichen €. Die folgende Tabelle beantwortet jetzt die umgekehrte Fragestellung, ob und wie das Euro-Zeichen mit Hilfe der verschiedenen Code-Tabellen verschlüsselt werden kann:

Verschlüsselung des Euro-Zeichens € gemäß			
CP 850	**CP 1252**	**ISO 8859-1**	**Unicode**
nicht vorhanden	hex. x0080 dez. 128	nicht vorhanden	hex. x20AC dez. 8364

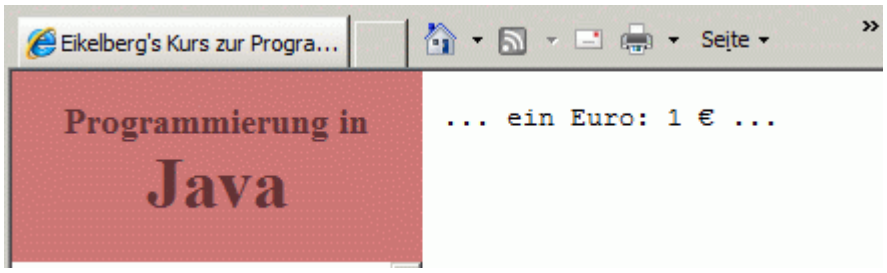
Das Euro-Zeichen fällt also in einen Bereich, für den die Unterschiede der Codierungen relativ groß sind. Die folgende Graphik ist unter Verwendung der Datei euro-386.txt erstellt worden und illustriert noch einmal den Sachverhalt unter Windows:



Erstellt man unter Unix eine Datei mit gleichem Text wie im Windows-Fenster der letzten Graphik, so ergibt sich folgender hexadezimaler Code:



Die obige Graphik ist unter Verwendung der Unicode-verschlüsselten Datei euro-unix.txt erstellt worden. Man beachte zunächst, dass die beiden zusammengehörenden Bytes der 16 Bit Unicode Verschlüsselung in Maschinensprache stets in umgekehrter Richtung codiert werden. Zu Beginn der Datei steht die Unicode Präambel. Sie lautet eigentlich **FE FF** und ist am Anfang der Datei in umgekehrter Reihenfolge als FF FE zu finden. Ebenso wird das Leerzeichen x0020 in der Datei als 20 00 codiert. Der Screenshot zeigt für das Eurozeichen im kleinen Fenster die Anzeige der einzelnen Bits. Dabei hat das Programm HIEW richtig die Reihenfolge der beiden Bytes bei der Anzeige umgekehrt und zeigt im kleinen Fenster korrekt **20 AC** als Codierung an. Folgen Sie dem Link zur Datei euro-unix.txt und lassen Sie sich die Datei in Ihrem Browser anzeigen. Ihr Browser sollte anhand der Unicode-Präambel erkennen, dass die Datei Unicode codiert ist und den Text wie im folgenden Ausschnitt eines Screenshots korrekt anzeigen:



Wie weiter oben angemerkt sind die deutschen Umlaute ebenfalls von dem Problem unterschiedlicher Codierung betroffen sind. Die folgende Tabelle zeigt die hexadezimalen Code-Nummern der deutschen Umlaute und des ß in den verschiedenen Codierungen. Dabei ergeben CP 1252, ISO 8859-1 und Unicode die gleichen Werte, da die Umlaute und das ß im Bereich x0090 bis x00FF (also dezimal 180 bis 255) liegen:

Buchstabe	hexadezimaler / dezimaler Wert gemäß	
	CP 850	CP 1252 , ISO 8859-1 , Unicode
ä	0x84 / 132	0x00E4 / 228
ö	0x94 / 148	0x00F6 / 246
ü	0x81 / 129	0x00FC / 252
Ä	0x8E / 142	0x00C4 / 196
Ö	0x99 / 153	0x00D6 / 214
Ü	0x9A / 154	0x00DC / 220
ß	0xE1 / 225	0x00DF / 223

Man sieht, daß sich die Problematik in Bezug auf die deutschen Sonderzeichen auf den Unterschied zwischen Code Page 850 und Unicode reduziert. Eine in Java codierte Lösung dieses Problems - die alle Schriftzeichen im Bereich x0080 bis x00FF berücksichtigt - ist im Paket `de.eikelberg.iotools` implementiert (in Programmform realisiert).

2.2.3.9 Hilfsmittel zur Bestimmung der Codierung und Decodierung von Zeichen

Wie man also sieht, ist es beim Programmieren stellenweise erforderlich, zu einem Zeichen des aktuellen Betriebssystems die zugehörige dezimale oder hexadezimale Uniocode-Verschlüsselung zu ermitteln. Auch der umgekehrte Weg kann erforderlich sein. Bei der Lösung dieser Fragen hilft neben den bereits oben gezeigten Tabellen auch das Programm bsp02005.Encoding, das *derzeit nur implementiert ist für Windows 9x/NT/2000/XP sowie Betriebssysteme mit Unicode-Verschlüsselung wie zum Beispiel UNIX-Varianten*:

- **"Zeichen des Betriebssystems -> Dezimalzahl des gleichen Zeichens in der Unicode-Codierung"**:
  Die Unicode Codierung, also den dezimal oder hexadezimal verschlüsselten Wert eines Zeichens können Sie für die ersten 256 Zeichen der Codierung Ihres Betriebssystems mit Hilfe des Programms bsp02005.Encoding / Auswahl 'u' bestimmen.
- **"Dezimalzahl eines Zeichens in der Unicode-Codierung -> gleiches Zeichen gemäß Codierung des Betriebssystems"**:
  Das sich aus einer Dezimalzahl ergebende Zeichen gemäß der Codierung Ihres Systems liefert das Programm bsp02005.Encoding / Auswahl 'i'.
- **"Hexadezimalzahl -> Unicode-Zeichen zur Hexadezimalzahl"**:
  Das Unicode Zeichen zu einem hexadezimalen Wert zwischen x0000 und xFFFF finden Sie ausgehend vom **Überblick über die wichtigsten Zeichen der Codierung Unicode 2.1.**

Als Beispiel seien hier die Bildschirmausgaben von zwei Läufen des Programms bsp02005.Encoding eingefügt:

```
Administrator: Eingabeaufforderung

...\jpr02005>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02005.Encoding
--------------------------------------------------------------------------------
    Programm zur Codierung und Decodierung von Zeichen
    Copyright 1999,2006 by W. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------

Dieses Programm ist nur zur Verwendung auf Windows Systemen
oder Systemen mit Unicode Verschluesselung konzipiert!

Wird Umwandlung von (Unicode-)Integercodierung
   nach Zeichen gemaess Codierung Ihres Systems [i]
oder Umwandlung Zeichen gemaess Codierung Ihres Systems
   nach (Unicode-)Integercodierung [u]
gewuenscht ? : u

Geben Sie die umzuwandelnden Zeichen in der Datei daten\eingabe.txt ein.

*********************************************************************
* Schreiben Sie alle umzuwandelnden Zeichen in die erste Zeile!     *
*                                                                   *
* Das Programm stellt fest, ob es auf einem Windows-Rechner laeuft  *
* und interpretiert die Eingabedatei in diesem Fall als mit einem   *
* Windows Editor erstellt, also nach Code Page 1252 verschluesselt. *
* Bei Verwendung von DOS-Editoren unter Windows erhaelt man somit   *
* ein falsches Ergebnis bei Eingabe der Zeichen x0080 (128) bis     *
* x00FF (255) der Code Page 850.                                    *
*                                                                   *
* Laeuft das Programm nicht unter Windows, wird von einer Unicode-  *
* Codierung der Datei ausgegangen. Falls Ihr System also kein       *
* Windows-System ist und nicht Unicode codiert, erhalten sie        *
* moeglicherweise ein falsches Ergebnis fuer die Zeichen x0080 (128) *
* bis x00FF (255).                                                  *
*                                                                   *
* Verwenden Sie dieses Programm also moeglichst unter Windows oder  *
* mit einer Unicode-faehigen Kombinatinon von Editor und Betriebs-  *
* system!                                                           *
*********************************************************************

Haben Sie alle Zeichen in die Datei eingabe.txt geschrieben? [j/n]: j

Unterschied Unicode / CP1252 bemerkt - CP1252 Zeichen Nr 128 gelesen!

Unterschied Unicode / CP1252 bemerkt - CP1252 Zeichen Nr 130 gelesen!

Unterschied Unicode / CP1252 bemerkt - CP1252 Zeichen Nr 131 gelesen!

Das Ergebnis finden Sie in der Datei daten\ausgabe.txt
```

In der Datei `eingabe.txt` im Verzeichnis `daten` befand sich bei diesem Durchlauf des Programms folgender Text:

```
eingabe.txt - Editor
Datei  Bearbeiten  Format  Ansicht  ?
äöüÄÖÜß€@²³$§^´|µ
Zeichen
0x82:‚
0x83:ƒ
```

Als Ergebnis wurde in die Datei `ausgabe.txt` folgender Text geschrieben:

ausgabe.txt - Editor

Datei Bearbeiten Format Ansicht ?

```
*********************************************************************
* Haben Sie diese Datei unter Windows erstellt, so muessen Sie      *
* sie mit einem Windows-Editor betrachten. Haben Sie die Datei      *
* unter einem anderen Betriebssystem erstellt, so erhalten Sie      *
* nur dann eine korrekte Anzeige, wenn Betriebssystem und Editor*
* gemaess Unicode entschluesseln. Ansonsten muessen Sie wegen       *
* unterschiedlicher Codierungen mit fehlerhaften Anzeigen fuer      *
* Zeichen zwischen hexadezimal x007F (dezmal 128) und x00FF         *
* (dezimal 255) rechnen!                                            *
*********************************************************************


Unicode Dezimal-Wert zu 'ä' ist: 228
Unicode Dezimal-Wert zu 'ö' ist: 246
Unicode Dezimal-Wert zu 'ü' ist: 252
Unicode Dezimal-Wert zu 'Ä' ist: 196
Unicode Dezimal-Wert zu 'Ö' ist: 214
Unicode Dezimal-Wert zu 'Ü' ist: 220
Unicode Dezimal-Wert zu 'ß' ist: 223
Unicode Dezimal-Wert zu '€' ist: 8364
Unicode Dezimal-Wert zu '@' ist: 64
Unicode Dezimal-Wert zu '²' ist: 178
Unicode Dezimal-Wert zu '³' ist: 179
Unicode Dezimal-Wert zu '$' ist: 36
Unicode Dezimal-Wert zu '§' ist: 167
Unicode Dezimal-Wert zu '^' ist: 94
Unicode Dezimal-Wert zu '´' ist: 180
Unicode Dezimal-Wert zu '|' ist: 124
Unicode Dezimal-Wert zu 'µ' ist: 181
Unicode Dezimal-Wert zu '' ist: 13
Unicode Dezimal-Wert zu '' ist: 10
Unicode Dezimal-Wert zu 'Z' ist: 90
Unicode Dezimal-Wert zu 'e' ist: 101
Unicode Dezimal-Wert zu 'i' ist: 105
Unicode Dezimal-Wert zu 'c' ist: 99
Unicode Dezimal-Wert zu 'h' ist: 104
Unicode Dezimal-Wert zu 'e' ist: 101
Unicode Dezimal-Wert zu 'n' ist: 110
Unicode Dezimal-Wert zu ' ' ist: 32
Unicode Dezimal-Wert zu '' ist: 13
Unicode Dezimal-Wert zu '' ist: 10
Unicode Dezimal-Wert zu '0' ist: 48
Unicode Dezimal-Wert zu 'x' ist: 120
Unicode Dezimal-Wert zu '8' ist: 56
Unicode Dezimal-Wert zu '2' ist: 50
Unicode Dezimal-Wert zu ':' ist: 58
Unicode Dezimal-Wert zu '‚' ist: 8218
Unicode Dezimal-Wert zu '' ist: 13
Unicode Dezimal-Wert zu '' ist: 10
Unicode Dezimal-Wert zu '0' ist: 48
Unicode Dezimal-Wert zu 'x' ist: 120
Unicode Dezimal-Wert zu '8' ist: 56
Unicode Dezimal-Wert zu '3' ist: 51
Unicode Dezimal-Wert zu ':' ist: 58
Unicode Dezimal-Wert zu 'ƒ' ist: 402
```

Die zweite Auswahlmöglichkeit im Programm bsp02005.Encoding ergibt einen Programmablauf der folgenden Art:

```
Administrator: Eingabeaufforderung
...\jpr02005>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02005.Encoding
--------------------------------------------------------------------------------
    Programm zur Codierung und Decodierung von Zeichen
    Copyright 1999,2006 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------

Dieses Programm ist nur zur Verwendung auf Windows Systemen
oder Systemen mit Unicode Verschluesselung konzipiert!

Wird Umwandlung von (Unicode-)Integercodierung
   nach Zeichen gemaess Codierung Ihres Systems [i]
oder Umwandlung Zeichen gemaess Codierung Ihres Systems
   nach (Unicode-)Integercodierung [u]
gewuenscht ? : i

Das Programm wird beendet bei Eingabe einer negativen Zahl !
Die Ausgabe finden Sie in der Datei ausgabe.txt!

Geben sie den Integer-Wert des Unicode-Zeichens ein : 7
Eingegeben wurde: 7
i = 7 ergibt einen Piepton!

Geben sie den Integer-Wert des Unicode-Zeichens ein : 65
Eingegeben wurde: 65


Geben sie den Integer-Wert des Unicode-Zeichens ein : 128
Eingegeben wurde: 128
Der Dezimalwert 128 ergibt in Unicode und Code Page 1252 verschiedene Zeichen!


Geben sie den Integer-Wert des Unicode-Zeichens ein : 228
Eingegeben wurde: 228


Geben sie den Integer-Wert des Unicode-Zeichens ein : 196
Eingegeben wurde: 196


Geben sie den Integer-Wert des Unicode-Zeichens ein : 8364
Eingegeben wurde: 8364


Geben sie den Integer-Wert des Unicode-Zeichens ein : -1
```

Bei diesem Programmdurchlauf wurde in der Datei ausgabe.txt folgender Text erzeugt:

```
ausgabe.txt - Editor
Datei  Bearbeiten  Format  Ansicht  ?
*********************************************************************
* Haben Sie diese Datei unter Windows erstellt, so muessen Sie      *
* sie mit einem Windows-Editor betrachten. Haben Sie die Datei      *
* unter einem anderen Betriebssystem erstellt, so erhalten Sie      *
* nur dann eine korrekte Anzeige, wenn Betriebsystem und Editor     *
* gemaess Unicode ver- bzw. entschluesseln.                         *
* Ansonsten muessen Sie wegen unterschiedlicher Codierungen mit     *
* fehlerhaften Anzeigen fuer Zeichen zwischen hexadezimal           *
* 0x007F (dezimal 128) und 0x00FF (dezimal 255) rechnen!            *
*********************************************************************


Der Unicode-Dezimalwert 7 ergibt das Zeichen: '•'
Der Unicode-Dezimalwert 65 ergibt das Zeichen: 'A'
Der Dezimalwert 128 ergibt in Unicode und Code Page 1252 verschiedene Zeichen!
Der Unicode-Dezimalwert 128 ergibt das Zeichen: '?'
Der Unicode-Dezimalwert 228 ergibt das Zeichen: 'ä'
Der Unicode-Dezimalwert 196 ergibt das Zeichen: 'Ä'
Der Unicode-Dezimalwert 8364 ergibt das Zeichen: '€'
```

Das zum Zeichen 128 angezeigte Fragezeichen entsteht daraus, dass die virtuelle Maschine in dem Fall, dass man aus einem Programm ein Zeichen aus dem Bereich 128 bis 159 ausgibt, statt des angegebenen Zeichens stets das Zeichen 0x3F (Fragezeichen) ausgibt. Dies entspricht der Tatsache, dass die Unicode-Codierung im Bereich der Zeichen 128 bis 159 keine druckbaren Zeichen kennt und dass dieser Bereich der einzige Bereich ist, in dem sich Code Page 1252 und Unicode unterscheiden.

2.2.3.10 Portierung von Java Quelltext

Was bedeuten diese Beobachtungen

- *für die Codierung von Java-Quelltext sowie*
- *für die Wahl von Verzeichnis-/Paketnamen und Datei-/Klassennamen*

für Java-Programme? Zunächst zeigt sich, dass es prinzipiell Probleme bei der Portierung (Übertragung auf einen anderen Rechnertyp oder ein anderes Betriebssystem) von Java Quellcode und Java Anwendungen geben kann. Ferner kann es wegen der Verwendung zweier unterschiedlicher Codierungen unter Windows zusätzliche Probleme geben. Diese Probleme können - sowohl in Bezug auf die Verwendung

von Zeichen im Java Quelltext als auch bei der Wahl von Verzeichnis- und Dateinamen - durch Beachtung folgender **Regeln** vermieden werden:

1. Bei einer Verwendung der druckbaren Unicode Zeichen x0000 bis x007F - dies ist 7-Bit ASCII - ergibt sich nie ein Problem.
2. Die Zeichen x0080 bis x009F der Windows-Codierung Code Page 1252 sollten möglichst nicht verwendet werden, da ihnen bei Unicode-Decodierung nicht druckbare Zeichen entsprechen. Zudem verwendet Java ab Version 1.2 für diese Zeichen stellvertretend das Fragezeichen (Zeichen 0x3F / 63).
3. Die Zeichen x00A0 bis x00FF von Unicode bzw. Code Page 1252 können problemlos verwendet werden, sofern eine Portierung des Quelltextes lediglich zwischen Windows- und Unix-Rechnern erfolgt. In anderen Fällen muss hier auch mit Inkompatibilitäten gerechnet werden.
4. Von der Verwendung von Unicode-Zeichen oberhalb x00FF sollte Abstand genommen werden, wenn der Quelltext auch unter Windows lesbar sein sollte.
5. Werden Programme mit Konsol-Editoren erstellt, so sollten stets nur die druckbaren Zeichen des 7-Bit ASCII Code verwendet werden. Am besten man verwendet Konsol-Editoren bei der Java-Programmierung unter Windows nicht mehr. Unter Windows 7 ist standardmäßig auch kein Konsol-Editor verfügbar.

Insbesondere Regel 1 ist eine in der Praxis auch in anderen Zusammenhängen oft beachtete und bewährte Regel. Neben diesen Regeln gibt es weitere Einschränkungen zur Verwendung von Schriftzeichen bei der Erstellung von Java Quellcode. Diese Einschränkungen sind Thema von Kapitel 2.2.4.

2.2.4 Lexikalische Elemente

Wir kehren zur Betrachtung der Methode main() zurück. Soll die Methode main() mit Java-Anweisungen gefüllt werden, so müssen einige sprachlichen Regeln von Java beachtet werden, die **Syntaxregeln** genannt werden. Die somit erforderliche Besprechung der Syntaxregeln beginnt mit der Betrachtung der **lexikalischen Eigenschaften**, also den Eigenschaften von Java in Bezug auf die Verwendung von Schriftzeichen bei der Codierung eines Programms. Fundamental für die sprachliche Gestaltung von Java ist die Verwendung des Unicode-Zeichensatzes, weshalb er im vorangegangenen Kapitel besprochen wurde.

2.2.4.1 Bezeichner

In Programmcode fast jedes Programms treten sogenannte **Bezeichner** auf. Bezeichner sind Namen, die vom Programmierer zur Benennung von Variablen (s.u.), Methoden, Klassen und Paketen gewählt werden. Ein spezielles Beispiel für einen Bezeichner ist zum Beispiel der Klassenname zum Hauptprogramm. Jeder Bezeichner kann aus einer beliebig langen Folge von Schriftzeichen bestehen, wobei folgende Einschränkungen beachtet werden müssen:

- Das erste Schriftzeichen muß ein sogenannter **Unicode-Buchstabe** sein. Diese sind die Klein- und Großbuchstaben des Alphabets sowie der Unterstrich "_" und das Dollarzeichen. Neben diesen Zeichen sind einige weitere Zeichen möglich, von deren Nutzung aber aus den in Kapitel 2.2.3 erläuterten Gründen abgeraten werden muß.
- Die folgenden Schriftzeichen können unter Windows beliebige **Unicode-Zeichen** sein. Ausgenommen sind hier lediglich die Zeichen, die in der Syntax von Java eine besondere Bedeutung haben wie zum Beispiel +,-,*,/,%,| usw. sowie einige Sonderzeichen wie zum Beispiel §, die hochgestellte 2 bzw. 3, der unterbrochene senkrechte Strich sowie das Copyright-Zeichen.
- Insgesamt darf der Bezeichner nicht einen Namen haben, dessen Zeichenfolge identisch ist mit der Zeichenfolge eines reservierten Java-Wortes. So ist es zum Beispiel nicht möglich, als Namen für eine Variable den Namen `true` oder `false` zu wählen.

Eine Liste der nicht als Variablennamen wählbaren Zeichenfolgen finden Sie im **Anhang**. Die Verwendung von Schriftzeichen in Bezeichnern können Sie mit Hilfe des Programms `bsp02008.Sonderzeichen` untersuchen. Die Ausgabe dieses Programms erfolgt in eine Datei. Das Ergebnis eines Laufs können Sie in der Datei **sonderz.txt** betrachten.

2.2.4.2 Ausdrücke

Die kleinste ausführbare Einheit im Quellcode eines Java Programms ist der **Ausdruck**. Ein Ausdruck kann zum Beispiel eine mathematische Berechung oder einen logischen Ausdruck enthalten und dient zur Formulierung der im Programm vorzunehmenden Aufgaben. So führt zum Beispiel der folgende Ausdruck zur Berechnung von 256 mal 16:

```
256 * 16
```

In der Regel treten in Ausdrücken allerdings nicht nur fest vorgegebene Zahlen auf, sondern zumindest teilweise auch Variablen (siehe 2.2.4.3). So führt zum Beispiel

```
256 * x1
```

zur Multiplikation des Inhaltes der Variablen `x1` mit der Zahl 256.

2.2.4.3 Anweisungen

Der kleinste Bereich in einem Java-Programm ist die sogenannte **Anweisung**. Eine Anweisung ist oft nur ein einzelner Befehl oder eine Codezeile für den Compiler, kann sich aber auch über mehrere Zeilen erstrecken. Daher ist es erforderlich, dem Java-Compiler anzuzeigen, an welcher Stelle jeweils eine Anweisung beendet ist. In Java wird daher - wie in einigen anderen Programmiersprachen auch - am Ende einer Anweisung stets ein Semikolon geschrieben.

Grundsätzlich wird bei einer Anweisung im Java Quellcode ein Schritt für die Verarbeitung des Java-Programms formuliert. Den einfachsten Fall - manchmal aufgrund der Java Syntax erforderlich - bildet die **leere Anweisung**, die schlichtweg dazu führt, daß das Programm *nichts* tut. Sie besteht also lediglich aus dem die Anweisung abschließenden Semikolon:

```
;
```

Eine weitere Möglichkeit, eine Anweisung zu formulieren, ist die Codierung einer **Ausdrucksanweisung**. Bei einer Ausdrucksanweisung wird ein Ausdruck mit abschließendem Semikolon codiert. Dies führt zur Auswertung des Ausdrucks an dieser Stelle bei der Abarbeitung des Programms. Eine häufig auftretende Form der Ausdrucksanweisung ist der Aufruf einer Methode. Eine Methode wird aufgerufen, indem man ihren Namen und den Namen des zugehörigen Objektes - manchmal statt des Objektes den Namen der Klasse, vgl. Kapitel 2.3.3 - in das Programm schreibt. So gibt es im Paket de.eikelberg.iotools zum Beispiel eine Methode `ScreenOut.kPrint()`, die eine als Argument - also zwischen den runden Klammern - mitgegebene Zeichenkette auf dem Konsolbildschirm ausgibt. Eine solche Anweisung lautet also zum Beispiel wie folgt:

```
ScreenOut.kPrintln();
ScreenOut.kPrint("Geben Sie eine ganze Zahl ein: ");
```

Zeichenketten werden stets von doppelten Anführungszeichen `"` umschlossen. Ausgegeben wird der Text `Geben Sie eine ganze Zahl ein: `. Auf diese Weise kann man in einfacher Weise im Programm eine Eingabeaufforderung codieren.

Eine wichtige Form der Anweisung ist die sogenannte **Deklaration**. Mit Hilfe der Deklaration wird der Java Compiler bzw. der JVM bekanntgegeben, dass im weiteren Programmverlauf eine Variable verwendet werden soll. **Variablen** sind Speicherplätze, die während des Programmablaufs im Hauptspeicher eingerichtet werden und für das Speichern von Daten während des Programmablaufs bereitstehen. Zur Kenntlichmachung dieser Speicherplätze erhalten die Variablen im Programm einen Variablennamen. Oft wird der Begriff Variable auch umgangssprachlich für den Variablennamen verwendet.

Die Bereitstellung der Speicherplätze zu Variablen muss in Java vom Programmierer durch die Deklaration angestoßen werden - manchmal ist hier noch ein zweiter Schritt notwendig, wenn die Informationen in der Deklaration nicht ausreichen, um den Speicherplatz wirklich anzulegen (vgl. Kapitel 2.6).

Variablennamen müssen nach den oben genannten Regeln für die Bezeichner gewählt werden. Für die korrekte Einrichtung dieses Speicherplatzes ist es ferner notwendig, der JVM mitzuteilen, welche Art von Daten in diesem Speicherplatz abgelegt werden sollen. Daher besteht die Deklaration aus der Angabe des Datentyps, gefolgt von dem Variablennamen und dem abschliessenden Semikolon. Beispiele für Deklarationen sind die folgenden beiden Programmzeilen:

```
String c1;
int x1;
```

In der ersten Zeile wird eine Variable mit Namen `c1` deklariert, die Daten vom Typ String - also Zeichenkette - aufnehmen soll. Die zweite Zeile deklariert eine Variable mit Namen `x1`, die ganze Zahlen (integer) aufnehmen soll. Die elementaren für die Deklaration erforderlichen Datentypen werden in Kapitel 2.3.1 erklärt.

Eine sehr häufige Form der Anweisung ist die sogenannte **Zuweisung**. Sie besteht aus einem Ausdruck sowie einer Variablen sowie einem dazwischen angeordneten Gleichheitszeichen. Rechts vom Gleichheitszeichen steht der Ausdruck, der ausgewertet und damit berechnet werden soll, und links vom Gleichheitszeichen steht die Variable, zu der ein Speicherplatz gehört, auf dem das Ergebnis der Auswertung des Ausdrucks gespeichert werden soll.

Beispiele für Anweisungen sind in den bisherigen Beispielprogrammen naturgemäß bereits in größerer Zahl vorgekommen, so zum Beispiel auch Programmcode, der folgenden Anweisungen ähnlich ist:

```
String c1;
c1 = KeyboardIn.kReadLine();
int x1;
x1 = Integer.parseInt(c1);
```

In der ersten Zeile des obigen Source Code Ausschnitts wird zunächst wieder eine Variable mit Name `c1` und vom Datentyp `String` - deutsch Zeichenkette - deklariert. In der zweiten Zeile wird die Methode `KeyboardIn.kReadLine()` ausgeführt, um von der Tastatur eine Zeichenkette einzulesen und auf dem Speicherplatz zur Variablen `c1` abzulegen. Die Methode `KeyboardIn.kReadLine()` stammt aus dem Paket de.eikelberg.iotools (vgl. Kapitel 2.3). In der dritten Zeile wird eine Variable mit Name `x1` deklariert. Die Anweisung `Integer.parseInt(c1);` in der vierten Zeile wandelt die Zeichenkette in `c1` in eine vom Programm verarbeitbare Zahl um. Dabei wird vorausgesetzt, dass die Zeichenkette aus einer Folge von Ziffern besteht und keine anderen Schriftzeichen in der Zeichenkette auftreten. Ist diese Bedingung nicht erfüllt, so wird es an dieser Stelle bei der Ausführung einen Programmabbruch geben! Eigentlich sind an dieser Stelle des Programms somit entsprechende programmtechnische Maßnahmen erforderlich!

Insgesamt lautet also ein Programmstück zum Einlesen einer ganzen Zahl zum Beispiel wie folgt:

```
ScreenOut.kPrintln();
ScreenOut.kPrint("Geben Sie eine ganze Zahl ein: ");
String c1;
c1 = KeyboardIn.kReadLine();
int x1;
x1 = Integer.parseInt(c1);
```

2.2.4.4 Blöcke

Wie das letzte Beispiel zeigt, treten Anweisungen häufig gruppiert auf. Darüber hinaus fördert auch die Planung eines Programms mit Hilfe der Struktogramme die sinnvolle Gruppierung von Anweisungen zu logisch zusammenhängenden Bereichen. Damit diese Gruppierung im Programm auch geeignet erfolgen kann, gibt es in Java das Sprachelement **Block**. Ein solcher Block kann zum Beispiel einer der bei den Struktogrammen auftretenden Formen der Blöcke sein. Aber auch kleinere Bereiche können als Blöcke formuliert werden.

In Java wird ein Block - wie in einigen anderen Programmiersprachen auch - von geschweiften Klammern umschlossen und hat damit folgende Form

```
{
// hier stehen die Anweisungen des Blocks
}
```

Dabei ist es eine gute Tradition, die die Übersichtlichkeit des Quelltextes erhöht und zudem später auch beim Programmieren hilft, dass nach jedem Anfang eines Blocks die folgenden Zeilen des Programmcodes - ausgenommen Anweisungen - eingerückt werden und die Einrückung beibehalten wird bis zum Ende des Blocks. Bei allen Beispielen dieses Kurses wird der Programmcode entsprechend dieser Tradition formuliert und dabei eine Einrückung um zwei Stellen vorgenommen. So könnte das oben angeführte Stück auch wie folgt aufgeschrieben werden:

```
{
  ScreenOut.kPrintln();
  ScreenOut.kPrint("Geben Sie eine ganze Zahl ein: ");
  {
    String c1;
    c1 = KeyboardIn.kReadLine();
  }
  {
    int x1;
    x1 = Integer.parseInt(c1);
  }
}
```

In diesem Beispiel macht die Formulierung der Blöcke noch keinen richtigen Sinn. Bei der Formulierung von Steuerungselementen wird die Blockbildung später allerdings sehr wichtig.

2.2.4.5 Kommentare

Wie bereits in Kapitel 1 erwähnt ist es außerordentlich wichtig und auch im Eigeninteresse des Programmierers, daß jedes Java-Programm ausreichend dokumentiert wird. Hierzu gibt es in Java drei verschiedene Kommentarzeichen:

- Text rechts von einem doppelten Querstrich (slash) // wird bis zum Ende der Zeile nicht vom Java-Compiler ausgewertet und vielmehr als Kommentar angesehen.
- Ein auch mehrzeiliger Bereich für Kommentare wird mit /* begonnen und mit */ beendet.
- Sogenannte **Dokumentationskommentare** beginnen mit /** und enden mit */. Sie müssen nach bestimmten Regeln aufgebaut sein, vor dem zu dokumentierenden Bereich stehen und können anschließend mit Hilfe des JDK-Programms `javadoc` aus dem Programmcode heraugelöst und in ein HTML-Dokument umgewandelt werden. Jeder Zeilenanfang im Bereich eines Dokumentationskommentars darf zusätzlich durch ein Sternchen gekennzeichnet werden, das von javadoc überlesen wird.

Mit den Dokumentationskommentaren und dem Programm `javadoc` liegt ein einfach zu handhabendes und leistungsfähiges Werkzeug zur Kommentierung von Klassen außerhalb des Quelltextes vor. Es wurde auch bei der Entwicklung der Klassen des Paketes de.eikelberg.iotools verwendet und erzeugte die in das nächsten Kapitel 2.3 integrierte Dokumentation dieses Paketes.

Im Projektverzeichnis `iotools` zum Paket de.eikelberg.iotools finden Sie die Batch-Datei **iotools_javadoc.bat**, die ein Beispiel für den Aufruf des Programms `javadoc` enthält. In den Java Quelldateien zu den Klassen des Pakets de.eikelberg.iotools finden Sie Dokumentationskommentare, die von `javadoc` zu **dieser Dokumentation** zusammengefügt wurden.

Ergänzt man in dem zuletzt genannten Beispiel einzeilige Kommentare und entfernt man gleichzeitig die nicht erforderliche Blockbildung, so erhält man folgenden Java Quellcode:

```
{
// Eingabeaufforderung
  ScreenOut.kPrintln();
  ScreenOut.kPrint("Geben Sie eine ganze Zahl ein: ");

// eingegebene Zeichenkette einlesen
  String c1;
  c1 = KeyboardIn.kReadLine();

// eingelesene Zeichenkette in eine ganze Zahl umwandeln
  int x1;
  x1 = Integer.parseInt(c1);
}
```

Dieser Block könnte zum Beispiel geringfügig ergänzt als Modulblock zu einer Methode verwendet werden, die ganze Zahlen einliest und diese eingelesene Zahl als Rückgabewert der Methode weitergibt. Gibt man dieser Methode den Namen `leseInteger()`, so hat der Programmcode der Methode insgesamt folgende Form:

```
public static int leseInteger()
```

```
{
// Eingabeaufforderung
  ScreenOut.kPrintln();
  ScreenOut.kPrint("Geben Sie eine ganze Zahl ein: ");

// eingegebene Zeichenkette einlesen
  String c1;
  c1 = KeyboardIn.kReadLine();

// eingelesene Zeichenkette in eine ganze Zahl umwandeln
  int x1;
  x1 = Integer.parseInt(c1);

// gelesene Zahl an das rufende Programm als Rückgabewert
// der Methode weitergeben
  return x1;
}
```

Das Beispielprogramm bsp02009.IntegerEinAus ist ein einfaches Programm, das diese Methode verwendet. In diesem Beispielprogramm werden auch die verschiedenen Kommentarformen genutzt. Die Klasse bsp02009.IntegerEinAus enthält insgesamt folgenden Java Quellcode:

```
/* Kapitel 2 Beispiel 9
   Verwendung der Methode leseInteger() und
   Dokumentation von Source Code mit den verschiedenen Formen
   des Kommentars.
*/

package bsp02009;

import de.eikelberg.iotools.*;

/**Das Hauptprogramm der Klasse IntegerEinAus liest eine ganze Zahl
   ein und gibt sie anschließend wieder aus. Das Programm besteht
   aus der Methode main(), in dem für das Einlesen der ganzen Zahl
   die Methode leseInteger() verwendet wird.
*/
public class IntegerEinAus
{

/* Hauptprogramm
   Man beachte die Anordnung der öffnenden Klammern zur
   Klasse und zur Methode main()
*/
/**Die Methode main() liest eine ganze Zahl ein und gibt sie
   anschließend wieder aus. main() dient als Testprogramm
   für die Methode leseInteger().
*/
  public static void main(String[] args)
  {

// Startmeldung des Programms
    ScreenOut.kStartScreenOut(
      "Testprogramm zur Verwendung der Methode leseInteger()",
      "M. Eikelberg (www.eikelberg.de)",
      "2000");

// integer Variable deklarieren und ganze Zahl einlesen
    int i;
    i = leseInteger();

// eingelesene Zahl wieder ausgeben
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("Eingegeben wurde die Zahl : "+i);
  }

/**Die Methode leseInteger() liest eine ganze Zahl ein. Sie verwendet
   die Tastatureingabe in der Konsolbox. Die eingelesene Zahl wird als
   Rückgabewert der Methode an die rufende Methode weitergegeben.
*/
  public static int leseInteger()
  {
// Eingabeaufforderung
    ScreenOut.kPrintln();
```

```
    ScreenOut.kPrint("Geben Sie eine ganze Zahl ein: ");

// eingegebene Zeichenkette einlesen
    String c1;
    c1 = KeyboardIn.kReadLine();

// eingelesene Zeichenkette in eine ganze Zahl umwandeln
    int x1;
    x1 = Integer.parseInt(c1);

// gelesene Zahl an das rufende Programm als Rückgabewert
// der Methode weitergeben
    return x1;
  }
}
```

Im Verzeichnis `jpr02009` der DVD finden Sie zu den Dokumentationskommentaren eine geeignet angepaßte Datei iotools_javadoc.bat, die aus den Dokumentationskommentaren HTML-Dateien im Unterverzeichnis `jpr02009\doc` erzeugt. Die erzeugte Dokumentation können Sie betrachten, wenn Sie **diesem Link** folgen.

2.3 Elementare Datentypen und ihre Verarbeitung

2.3.1 Elementare Datentypen

2.3.1.1 Deklaration und Wertebereich elementarer Datentypen

Im letzten Kapitel ist bereits mehrfach der Begriff Variable gefallen. Eine Variable wurde dabei wie folgt definiert:

Variablen sind Speicherplätze, die während des Programmablaufs im Hauptspeicher eingerichtet werden und für das Speichern von Daten während des Programmablaufs bereitstehen. Zur Kenntlichmachung dieser Speicherplätze erhalten die Variablen im Programm einen **Variablennamen**. Die Variablennamen gehören zu den Bezeichnern und unterliegen damit auch den Bedingungen an die sprachliche Gestaltung von Bezeichnern. Oft wird der Begriff Variable auch umgangssprachlich für den Variablennamen verwendet.

Da es verschiedene Typen von Speicherplätzen gibt, die für Variablen genutzt werden können, ist es im Programm erforderlich, vor der Verwendung der Variablen dem Compiler und der JVM anzukündigen, welche Art Speicherplatz für die Variable verwenden soll. Java stellt hierzu 8 verschiedene Grundtypen von Speicherplätzen zur Verfügung, die **elementare Datentypen** genannt werden.

Der Grund für die Spezifikation unterschiedlicher Datentypen liegt darin, dass Daten im Rechner intern je nach vorgesehener Verarbeitung unterschiedlich gespeichert werden. Dabei unterliegt die Speicherung während des Programmablaufs im Hauptspeicher und die dauerhafte Speicherung auf einem Datenträger den gleichen Prinzipien bis auf die Ausnahme, daß standardmäßig von den elementaren Datentypen auf einem Datenträger lediglich der Typ der Schriftzeichen gespeichert werden kann. Die entsprechende Speicherungsformen Unicode, Code Page 1252 etc. für **Schriftzeichen** sind in Kapitel 2.2.3 ausführlich diskutiert worden. Der entsprechende Datentyp hat im Java Quellcode den Namen **`char`**. Dieser Typ wird im Sprachgebrauch auch als **character** (Schriftzeichen) bezeichnet.

Anders als mit Schriftzeichen verhält es sich zum Beispiel mit Zahlen. Auf Datenträgern müssen diese Zahlen stets auch als Folge von Ziffern und damit als Schriftzeichen gespeichert werden. Soll eine Folge von Ziffern in Form einer Zahl während des Programmablaufs jedoch nicht als Text, sondern - zum Beispiel bei Berechungen - als Zahl verwendet werden, so muss diese Folge von Ziffern im Hauptspeicher des Rechners grundsätzlich anders gespeichert werden. Damit ein Programm zum Beispiel vorzeichenbehaftete **ganze Zahlen** verarbeiten kann, werden diese intern als Dualzahl gespeichert. **Gleitkommazahlen**, auch Gleit*punkt*zahlen genannt - also Zahlen mit Stellen nach dem Komma und vielleicht sogar multipliziert mit einer Zehnerpotenz - werden nach der Norm **IEEE 754** intern gespeichert (IEEE - Institute of Electrical and Electronic Engineers, amerikanischer Fachverband mit der Hauptaufgabe der Standardisierung und Normierung zum Beispiel auch im Netzwerkbereich).

Neben diese unterschiedliche interne Speicherungsart tritt ferner die Frage, wie groß der Speicherplatz sein soll, der die zu speichernde Zahl aufnehmen soll. Dies bestimmt natürlich direkt, wie groß bzw. wie klein die im Speicherbereich abzuspeichernde Zahl werden darf. Entsprechend gibt es in Java für ganze Zahlen je nach Größe die Datentypen

- **`byte`**,
- **`short`** (gemeint ist short integer - intern halb so groß wie int).
- **`int`** (sprachlich **integer** (ganzzahlig)) und
- **`long`** (gemeint ist long integer - intern doppelt so groß wie int).

Für Gleitkommazahlen gibt es die beiden Datentypen

- **`float`** (Gleitpunkt) und
- **`double`** (intern doppelt so groß wie float).

Neben den Datentypen zu Schriftzeichen und Zahlen kennt Java noch einen weiteren elementaren Datentyp, den Datentyp **`boolean`** zur Speicherung der Wahrheitswerte **`true`** (wahr) und **`false`** (falsch). Hier genügt bei der Speicherung intern ein Bit.

Die folgende Tabelle listet die elementaren Datentypen von Java auf und zeigt gleichzeitig, wie eine Deklaration einer Variablen mit Name `vname` zum entsprechenden Datentyp auszusehen hat.

Typ der Variablen	Wertebereich	Größe in Byte	Deklaration in Java
boolean	`true, false`	1	`boolean vname`
char	intern: alle Unicode-Zeichen bei Ein- und Ausgabe: siehe Kapitel 2.2.3	2	`char vname`
byte	-128..+127	1	`byte vname`
short	-32.768..+32.767	2	`short vname`
int	-2.147.483.648..+2.147.483.647	4	`int vname`
long	- 9.223.372.036.854.775.808..+9.223.372.036.854.775.807	8	`long vname`

float	±1,175494351e-38..±3,4028235e+38	4	`float vname`
double	±2,225073859e-308..±1,797693134e+308	8	`double vname`

Hinweis: In der obigen Tabelle ist ein Punkt ein Tausendertrennzeichen und ein Komma ein Dezimalkomma.

Die Darstellung -1,797693134e+308 in der Tabelle meint die Zahl "-1,797693134 mal 10 hoch +308" und ist eine der möglichen Darstellungen einer Gleitpunktzahl bei der Ausgabe aus einem Java-Programm. In den nachfolgenden Berechnungen ist mit "Länge" die Anzahl der ***Bits*** gemeint, die intern zur Speicherung der entsprechenden Variablen benötigt werden. Für die ganzzahligen Datentypen ergeben sich die minimalen und maximalen Werte wie folgt:

$$Minimum = -2^{(Länge - 1)} \qquad Maximum = 2^{(Länge - 1)} - 1$$

Für die Gleitkommazahlen ergeben sich die minimalen und maximalen Werte aus der Norm IEEE 754 bzw. ANSI/IEEE 754-1985. Hier erklären sich sich zum Beispiel die maximalen Werte wie folgt:

$$2^{128} = 3.402823669\ 10^{38} \qquad 2^{1024} = 1.797693135\ 10^{308}$$

Man beachte, dass bei Gleitkommazahlen natürlich auch die Frage interessiert, wie genau eine Zahl verarbeitet werden kann, wie nahe sie also gegen Null gehen kann, ohne gleich Null gesetzt zu werden. Hier werden beim Datentyp float etwa Werte bis ca. 10^{-45} und beim Datentyp double etwa Werte bis ca. 10^{-324} verarbeitet.

Variablen werden vom Programmierer wie in der Tabelle aufgelistet deklariert. Dabei ist es auch möglich, mehrere Variablen - von Kommatas getrennt - gleichzeitig zu deklarieren wie in folgendem Beispiel:

```
    boolean windows;
    char c1,auswahl,euroZeichen;
    byte zaehler;
/* Variablen beginnend mit einem Kleinbuchstaben, sinnhaft gewählt,
   aus mehreren Worten oder Abkürzungen bestehend mit Großbuchstaben
   innerhalb des Variablennamens zur Kenntlichmachung der Sinnhaftigkeit
   des Variablennamens sind eine gute Angewohnheit:
*/
    short kleineZahl,kZahl;
// oft gewählt, aber nicht immer sinnvoll ist folgende Namenswahl
    int i1,i2,i3;
// L2 ist nicht schön, Variablen sollten mit einem Kleinbuchstaben beginnen
    long l1,L2,langeZahl;
    float f1,f2,ergebnis;
    double fehlerGrenze,naheNull;
```

2.3.1.2 Initialisierung elementarer Datentypen

Bei dem Zugriff im Programm kann die Programmlogik dazu führen, dass eine Variable erstmalig verwendet wird, ohne dass vorher ein Wert auf der Variablen gespeichert wurde. Dies führt in anderen Programmiersprachen als Java oft zu Programmfehlern oder Abstürzen. Daher wird diese Situation vom Java Compiler geprüft und man erhält gegebenenfalls einen Compilerfehler. Um diesen Fehler zu vermeiden, ist es möglich, einer Variablen direkt bei der Deklaration einen Wert zuzuweisen. Man spricht dann auch davon, dass die Variable **initialisiert** wird. Diese Initialisierung geschieht durch eine Zuweisung, die den Speicherplatz zur Variablen quasi mit einen Startwert belegt. Einzeln codierte Zuweisungen haben für die 8 elementaren Datentypen eine Form wie in den folgenden Beispielen

```
    windows = true;
    auswahl = 'x';
    euroZeichen = '\u20AC';
    zaehler = 0;
    kZahl = -32768;
    i1 = -2000000000;
    langeZahl = 9000000000000000000l;
    langeZahl = 9000000000000000000L;
    ergebnis = 0.0f;
    ergebnis = 0.0F;
    naheNull = 1e-300d;
    naheNull = 1e-300D;
```

Die Werte rechts von der Zuweisung sind feste Werte, die man **Konstanten** nennt. Man sieht folgende Regeln bei der Codierung von Konstanten:

- Bei boolean Variablen - auch boolesche Variablen genannt - werden für die Zuweisung der Wahrheitswerte 'wahr' und 'falsch' die Schlüsselörter **true** und **false** verwendet.
- character Konstanten werden zur Abgrenzung von Bezeichnern in einfache Hochkommata eingeschlossen. Alternativ kann statt der direkten Angabe des Schriftzeichens zwischen den Hochkommata auch die Eingabe des Hexadezimalwertes erfolgen, den das

Zeichen laut Unicodetabelle hat. In diesem Fall muss vor dem Hexadezimalwert noch die Zeichenfolge \u codiert werden.
- byte, short und integer Zahlkonstanten können einfach als Zahlwert eingegeben werden.
- Ganzzahlige long Zahlkonstanten werden durch ein l oder L nach der letzten Ziffer gekennzeichnet.
- float Gleitpunktzahlen werden durch ein f oder F nach der letzten Ziffer gekennzeichnet.
- double Gleitpunktzahlen werden durch ein d oder D nach der letzten Ziffer gekennzeichnet.

char Variablen können also auf zwei verschiedene Weise initialisiert werden. Kann das Zeichen, das der char Variablen zugewiesen werden soll, über die Tastatur eingegeben werden, so kann das Zeichen - wie im Beispiel zur Variablen auswahl - in Hochkommata eingeschlossen im Quelltext eingegeben werden. Kann das Zeichen jedoch - wie im Beispiel zur Variablen return - nicht über die Tastatur im Quelltext direkt eingegeben werden, so bestimmt man den Hexadezimalwert der Unicode-Verschlüsselung des Zeichens, ergänzt diesen Wert auf eine vierstellige Anzeige und stellt dieser Hexadezimalzahl die Zeichen \u voran. Die so entstehende Zeichenfolge wird anschließend in Hochkommata eingeschlossen. Im obigen Beispiel soll der Variablen return das nicht im Editor darstellbare Return-Zeichen zugewiesen werden. Hierzu wird zunächst der Hexadezimalwert des Return-Zeichens bestimmt. Dieser Wert ist gleich hexadezimal 13 und wird auf die vierstellige Darstellung 0013 ergänzt. Vor diesen hexadezimalen Zahlwert schreibt man jetzt noch die Zeichen \u und erhält \u0013 als Umschreibung des Return-Zeichens. Diese Umschreibung muss noch in Hochkommata eingeschlossen werden, damit sie einer char Variablen zugewiesen werden kann.

float und double Zahlwerte können als Fließpunktzahlen (Fließkommazahlen) oder in wissenschaftlicher Darstellung codiert werden. Generell gilt, dass - wie im gesamten englischsprachigen Bereich - stets statt eines Dezimalkommas ein Punkt verwendet wird, weshalb man auch vom **Dezimalpunkt** spricht. Die **Fließpunktdarstellung** besteht aus

1. Vorzeichen - das auch fehlen kann -,
2. Dezimalstellen vor dem Dezimalpunkt,
3. einem Dezimalpunkt,
4. Dezimalstellen nach dem Dezimalpunkt sowie
5. den Buchstaben d bzw. D (für double) oder f bzw. F (für float).

Die **wissenschaftliche Darstellung** besteht aus

1. dem Vorzeichen der Mantisse - das auch fehlen kann -,
2. Dezimalstellen der Mantisse vor dem Dezimalpunkt,
3. einem Dezimalpunkt,
4. Dezimalstellen der Mantisse nach dem Dezimalpunkt,
5. einem Buchstaben e zur Abtrennung von Mantisse und Exponent
6. dem Vorzeichen des Exponenten - das auch fehlen kann -,
7. dem ganzzahligen Exponenten sowie
8. den Buchstaben d bzw. D (für double) oder f bzw. F (für float).

Beispiele für Gleitpunktzahlen in Fließpunktdarstellung sind folgende Zahlen:

1.2345f -0.045f -.1111d -99999.99d

Beispiele für Zahlen in wissenschaftlicher Darstellung sind folgende Zahlen

-0.1111e-30f -.0011e-30f 1243.56e2d .12e2d

Deklaration und Initialisierung von Variablen können auch in einem einzigen Schritt geschehen. Hierzu zieht man die beiden Codebestandteile wie in folgendem Beispiel zusammen:

```
boolean booleanVar = false;
char charVar = 'c';
byte byteZahl = 100;
short shortZahl = 10000;
int integerZahl = 1000000000;
long longZahl = 1000000000000000000l;
float floatZahl = -0.1111e-30f;
double doubleZahl = 11.11e300d;
```

Ein Beispielprogramm zur Variablendeklaration und Initialisierung ist das Programm bsp02010.VariablenDef.

2.3.1.3 Lokale Deklaration

In Java ist es üblich, Variablen möglichst **lokal** zu deklarieren. Damit ist gemeint, dass Variablen innerhalb des kleinstmöglichen Blocks deklariert werden, so dass die Variable noch verwendbar ist. Es gilt dann:

- Eine Variable ist nur innerhalb des Blocks verwendbar, in dem sie deklariert ist!
- Variablennamen - also die Bezeichner - können in unterschiedlichen Blöcken unterschiedlich verwendet werden.

Der Vorteil der lokalen Deklaration liegt darin, dass bei der Java-Programmierung die in anderen Programmiersprachen zu Beginn des Programmcodes oft auftretenden großen Listen von Variablen mit vielen verschiedenen und daher auch komplizierteren Namen nicht in einem solchen Maße auftreten.

Als Beispiel für die lokal beschränkte Gültigkeit der Variablendeklaration dient das Programm bsp02011.LokaleVariablen:

```
package bsp02011;

import de.eikelberg.iotools.*;

public class LokaleVariablen
{

  public static void main(String[] args)
  {
    ScreenOut.kStartScreenOut(
      "Beispielprogramm zur lokalen Gültigkeit von Variablen",
      "M. Eikelberg (www.eikelberg.de)","2000");

    int i1 = 1;

// Block, in dem i2 deklariert ist
    {
      int i2 = 2;
      ScreenOut.kPrintln
        ("Der Wert der Variablen i2 ist gleich: "+i2);
    }

    ScreenOut.kPrintln
      ("Der Wert der Variablen i1 ist gleich: "+i1);

// Die folgende Anweisung ergibt einen Compiler-Fehler, da die
// Variable i2 hier bereits nicht mehr deklariert ist:
//    ScreenOut.kPrintln
//      ("Der Wert der Variablen i2 ist gleich: "+i2);
  }
}
```

2.3.1.4 Klassenvariablen

Die beschränkte Gültigkeit der Variablendeklaration ist zunächst von Vorteil. Da lokale Variablen ferner am Ende des kleinsten ihre Deklaration umgebenden Blocks wieder ungültig werden, endet die Gültigkeit einer Variablen somit stets spätestens am Ende der Methode, in der sie deklariert wurde. Manchmal ist es jedoch erforderlich Variablen anzulegen, die in mehr als einer Methode verfügbar sind. Derartige Variablen nennt man Klassenvariablen oder Instanzvariablen:

- Klassen- und Instanzvariablen werden direkt in der Klasse - also außerhalb der Methoden - deklariert.
- Die Deklaration einer Klassen- oder Instanzvariablen geschieht syntaktisch - also nach den gleichen Sprachregeln - wie die Deklaration eine lokalen Variablen.
- Wird vor einer außerhalb der Methoden deklarierten Variablen das Schlüsselwort `static` eingefügt, ist die Variable eine **Klassenvariable**.
- Fehlt das Schlüsselwort `static`, so ist die Variable eine **Instanzvariable**.

Die main-Methode ist mit dem Schlüsselwort static gekennzeichnet. Dies bedeutet, dass man aus der main-Methode nur auf Variablen zugreifen kann, die ebenfalls als static deklariert wurden - also die Klassenvariablen. Ein Zugriff auf Instanzvariablen von der main-Methode aus ist nicht möglich. Entsprechend werden Instanzvariablen erst später in Kapitel 2.8.3 behandelt.

> Hier werden zunächst nur Klassenvariablen betrachtet, denn aus der main-Methode heraus kann nur auf Klassenvariablen zugegriffen werden. Entsprechend muss bis auf Weiteres vor Variablen außerhalb von Methoden das Schlüsselwort **`static`** eingefügt werden.

Auf Klassen- und Instanzvariablen kann man gegebenenfalls auch aus anderen Klassen zugreifen. Hierzu kann die Gültigkeit - also die Frage, wo die Variable verwendet werden kann - mit den folgenden Schlüsselworten gesteuert werden.

- **private** bedeutet, dass die Variable nur in der sie umgebenden Klasse verwendbar ist, also nur in der Klasse gültig ist, in der sie codiert ist.
- **protected** bedeutet, dass die Variable
    - in der sie umgebenden gesamten Klasse,
    - in abgeleiteten Klassen (siehe objektorientierte Programmierung) dieser Klasse und
    - im gesamten diese Klasse umgebenden Paket verwendbar ist.
- **public** bedeutet, dass die Variable unbeschränkt, also auch außerhalb der Klasse und des Paketes verwendbar ist.

In den Sätzen der obigen Liste kann gemäß Java Terminologie bzw. objektorientierter Terminologie statt des Wortes 'verwendbar' auch das Wort **sichtbar** verwendet werden. Man spricht dann auch von der **Sichtbarkeit** der Variablen. Die Schlüsselwörter private, protected und public werden wie in den folgenden Beispielen bei der Deklaration vor das Schlüsselwort zum Variablentyp sowie in der Regel, aber nicht notwendig, vor das Schlüsselwort `static` geschrieben:

```
  private static int zaehler = 0;
  protected static float faktor = 0.5f;
  public static char x = 'X';
```

Die Schlüsselworte private, protected, public und static nennt man **Modifizierer**. Zusätzlich zu den genannten vier Modifizierern gibt es weitere, von denen **final** der wichtigste ist. **final** bewirkt, dass der Wert der entsprechenden Variablen nicht mehr verändert werden kann. Damit wird der Variablenname quasi zum *Alias* des Wertes, mit dem die Variable ursprünglich belegt worden ist. Die Variable selber hat dann die Funktion eine Konstanten. Ein Beispiel hierzu ist die folgenden Deklaration:

```
  public static final double e=2.7182818284590452353602874713527d;
```

Ein Beispielprogramm zur Verwendung von Modifizierern ist das Programm bsp02012.ModifVar:

```
package bsp02012;

import de.eikelberg.iotools.*;

public class ModifVar
{

  static private int zaehler = 0;
  static protected float faktor = 0.5f;
  static public char x = 'X';

  public static final double e=2.7182818284590452353602874713527d;

  public static void main(String[] args)
  {
    ScreenOut.kStartScreenOut(
      "Beispielprogramm zur Sichtbarkeit von Variablen",
      "M. Eikelberg (www.eikelberg.de)","2000");

// Die folgende Anweisung ergibt einen Compilerfehler
// e = 0.0d;

    ScreenOut.kPrintln
      ("Der Wert der Variablen zaehler ist gleich: "+zaehler);
    ScreenOut.kPrintln
      ("Der Wert der Variablen faktor  ist gleich: "+faktor);
    ScreenOut.kPrintln
      ("Der Wert der Variablen x       ist gleich: "+x);
    ScreenOut.kPrintln
      ("Der Wert der Variablen e       ist gleich: "+e);
  }
}
```

Man sieht, dass die Reihenfolge bei mehreren Modifizierern unterschiedlich sein kann.

2.3.1.5 Typkonvertierung

Mit der Deklaration der Variablen ist zunächst festgelegt, von welchem Datentyp die auf den entsprechenden Speicherplätzen abzulegenden Daten sind. Damit ist nicht nur die Codierung der Daten festgelegt sondern auch die Frage, in welcher Weise die entsprechenden Daten im Programm verarbeitet werden können. Dies liegt daran, dass zu jedem Verarbeitungsschritt festgelegt ist, auf welche Art von Daten dieser Verarbeitungsschritt angewendet werden kann. Damit kann die Situation auftreten, dass die Daten in einem ungeeigneten Datentyp vorliegen. Zur Bewältigung dieser Situation ist also die Überführung der Daten in einen anderen Datentyp - die sogenannte **Typumwandlung** oder **Typkonversion** bzw. kurz **Konversion** - erforderlich.

Falsche bzw. unpassende Datentypen findet der Java-Compiler, da er eine strenge Typprüfung vornimmt. In einigen Fällen fügt der Compiler selbständig Programmcode ein, der für die Typumwandlung sorgt. Man spricht in diesem Fall von **impliziter Typumwandlung**. In den meisten Fällen ist jedoch eine **explizite Typumwandlung** erforderlich, also eine ausdrücklich mit Hilfe von Java Sprachmitteln vom Programmierer vorgenommene Typumwandlung. Das entsprechende Java-Sprachelement hierzu heißt **Cast**.

Typumwandlungen von und nach boolean gibt es nicht. Für die restlichen sieben elementaren Datentypen gibt es jeweils einen Cast. Das Sprachelement hierzu besteht aus dem Schlüsselwort des Ziel-Datentyps, in runde Klammern eingefügt. Die Casts lauten also

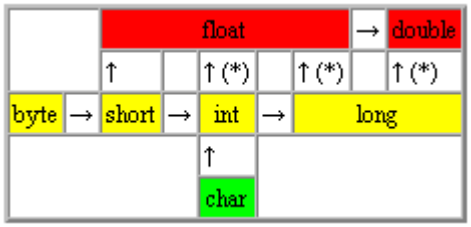
`(byte)` `(short)` `(int)` `(long)` `(float)` `(double)` und `(char)`.

Ein Beispiel für die Typumwandlung mit diesen Casts ist der folgende Programmcode aus dem Beispielprogramm bsp02013.Typumwandlung. Die Punkte . . . . zeigen an, dass an dieser Stelle Programmcode weggelassen wurde. Das Programm zeigt auch beispielhaft eine mögliche Verwendung von Modifizierern bei der Codierung von Methoden, die nicht außerhalb der Klasse sichtbar sein sollen. (Die Modifizierer private, protected und static können auch bei der Deklaration von Methoden eingesetzt werden, genaueres hierzu in

Kapitel 2.7).

```
package bsp02013;

import de.eikelberg.iotools.*;

public class Typumwandlung
{

// Deklaration der Variablen
  private static byte byteVar;
  private static short shortVar;
  private static int intVar;
  private static long longVar;
  private static float floatVar;
  private static double doubleVar;
  private static long lFloat,lDouble,Fehler;

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

  ....

  private static long typumwandlungVonGanzzahl()
  {
// Test Typumwandlung
    ScreenOut.kPrintln();
    ScreenOut.kPrintln(
      "Test Typumwandlung ausgehend von einer byte Ganzzahl");
    ScreenOut.kPrintln();

// einlesen einer Variablen mit Datentyp byte
    ScreenOut.kPrint("Geben Sie eine byte Zahl ein (-128 < ... < 127): ");
    byteVar = KeyboardIn.kReadByte();
    ScreenOut.kPrintln();

// implizite Umwandlung der eingelesenen byteVariablen
//  charVar = byteVar;   // ergibt Compilerfehler
    shortVar = byteVar;
    intVar = shortVar;
    longVar = intVar;
    floatVar = longVar;
    doubleVar = longVar;

// Ausgeben der konvertierten Daten
    ScreenOut.kPrintln
      ("Nach Konversion ergeben sich aus dem eingelesenen Wert:");
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("neuer Datentyp short  - Wert = : " + shortVar);
    ScreenOut.kPrintln("neuer Datentyp int    - Wert = : " + intVar);
    ScreenOut.kPrintln("neuer Datentyp long   - Wert = : " + longVar);
    ScreenOut.kPrintln("neuer Datentyp float  - Wert = : " + floatVar);
    ScreenOut.kPrintln("neuer Datentyp double - Wert = : " + doubleVar);
    ScreenOut.kPrintln();

    return longVar;
  }
}
```

Generell entsteht bei der Typumwandlung das Problem, dass intern für die verschiedenen Datentypen unterschiedlich viel Speicherplatz zur Verfügung steht. Damit ist es je nach Umwandlungsrichtung nicht immer möglich, den Informationsgehalt zu halten. Mit anderen Worten kann je nach Umwandlungsrichtung ein **Genauigkeitsverlust** entstehen. Alle automatischen Typumwandlungen zeigt die folgende Graphik. Ein Genauigkeitsverlust kann bei den Umwandlungen auftreten, die mit (*) gekennzeichnet sind.

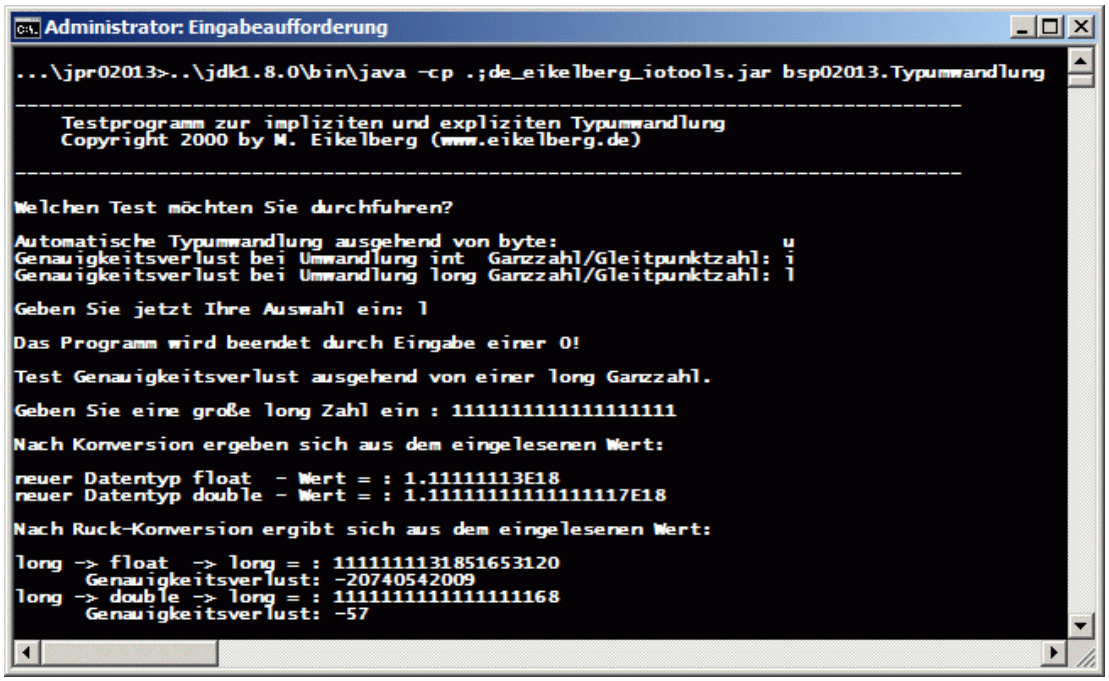


Achtung! Teilweise ist in der Literatur bzw. der Dokumentation davon die Rede, dass eine implizite Typumwandlung long → double stets ohne Genauigkeitsverlust durchgeführt wird. Dies ist nicht richtig zum Beispiel für die meisten ganzzahligen Zahlen mit 17 bis 19 Stellen. Es kann auch nicht richtig sein, da zwar beide Datentypen 8 Byte lang sind, beim Datentyp double einige Bits für den Exponenten verbraucht werden und damit nicht für die Ziffern 17 bis 19 zur Verfügung stehen.

Generell muß also bei allen Umwandlungen,

- die nicht den Pfeilen in der obigen Graphik entsprechen oder aber
- die zwar mit einem Pfeil in der Graphik eingezeichnet sind, jedoch mit (*) gekennzeichnet sind,

mit einem Genauigkeitsverlust bzw. Informationsverlust gerechnet werden. Das Ausmaß des Genauigkeitsverlustes zum Beispiel bei der Umwandlung zwischen Ganzzahlen vom Typ long und Gleitpunktzahlen kann mit Hilfe des Programms bsp02013.Typumwandlung, Auswahl 'l', getestet werden. Die Ausgabe des Programms in diesem Fall zeigt beispielhaft der folgende Screenshot:



2.3.2 Arithmetische Operatoren

2.3.2.1 Ausdrücke und Operatoren

Jeder Ausdruck in einem Java-Programm enthält mindestens einen **Operator**, der festlegt, wie die verschiedenen Variablen bzw. Werte in dem entsprechenden Ausdruck verknüpft werden sollen. Mathematische Ausdrücke enthalten dementsprechend die sogenannten **arithmetischen Operatoren**. Einen Überblick über die arithmetischen Operatoren bietet die folgende Tabelle. Dabei seien `a` und `b` **numerische Operanden**, also Variablen vom Typ byte, short, int, long, float oder double.

Findet der Java Compiler dabei zwei unterschiedliche Typen, so sorgt er automatisch für eine Konvertierung in jeden Datentyp, der sich gemäß der Pfeile in der Graphik in Kapitel 2.1.3.5 (implizite Typumwandlung) als gemeinsamer Datentyp ergibt, der beide Ausgangsdatentypen umfaßt. *Man beachte, daß dabei - wie in der Graphik auch vermerkt - auch eine nicht verlustfreie automatische Konvertierung in einen Gleitpunkt Datentyp erfolgen kann (vgl. Testprogramme* bsp02013.Typumwandlung *und* bsp02016.TypumwandlungBeiArithmetik *sowie die Ausführungen in Kapitel 2.3.1) !*

Operator	Bezeichnung	Wirkung	Anwendung	Bindungs-priorität	Assozia-tivität
`+`	unäres Plus	Beibehaltung des Vorzeichens: +a = a	`+a`	2	-
`-`	unäres Minus	Wechsel des Vorzeichens von a nach -a	`-a`	2	-
`+`	Addition	Addition zweier Zahlen	`a+b`	5	links
`-`	Subtraktion	Subtraktion zweier Zahlen	`a-b`	5	links
`*`	Multiplikation	Multiplikation zweier Zahlen	`a*b`	4	links
`/`	Division	Division zweier Zahlen	`a/b`	4	links
`%`	Modulo	Divisionsrest bei der Division einer Zahl a durch eine Zahl b	`a%b`	4	links
`++`	Präinkrement	Vergrößern des Wertes der Variablen um 1 **vor** Auswertung des umgebenden Ausdrucks	`++a`	1	-
`--`	Prädekrement	Verringern des Wertes der Variablen um 1 **vor** Auswertung des umgebenden Ausdrucks	`--a`	1	-
`++`	Postinkrement	Vergrößern des Wertes der Variablen um 1 **nach** Auswertung des umgebenden Ausdrucks	`a++`	1	-
`--`	Postdekrement	Verringern des Wertes der Variablen um 1 **nach** Auswertung des umgebenden Ausdrucks	`a--`	1	-

Die Bedeutung der Einträge zur Bindungspriorität wird in Kapitel 2.3.2.3 erklärt. Die Tabelleneinträge zur Assoziativität werden in Kapitel 2.4.6 erläutert. Bindungspriorität und Assoziativität sind für die logische Gruppierung und die Auswertungsreihenfolge von Operatoren von Bedeutung.

Die Inkrement und Dekrement-Operationen werden in 2.3.2.2 genauer erläutert. Die unbedingt zu beachtenden Besonderheiten bei der Division sind Thema von 2.3.2.4. Die Behandlung des Rechenfehlers bei der Gleitpunktarithmetik wird in 2.3.2.5 erklärt. In Kapitel 2.3.2.3 werden die in der Tabelle genannten Zahlen zur Bindungspriorität erläutert.

2.3.2.2 Inkrement und Dekrement

Die Inkrement und Dekrementoperation sind in der Regel für Programmieranfänger etwas gewöhnungsbedürftig und in ihrer Wirkung auch etwas verwirrend. Am einfachsten kann man die Wirkung dieser Operatoren wie folgt beschreiben:

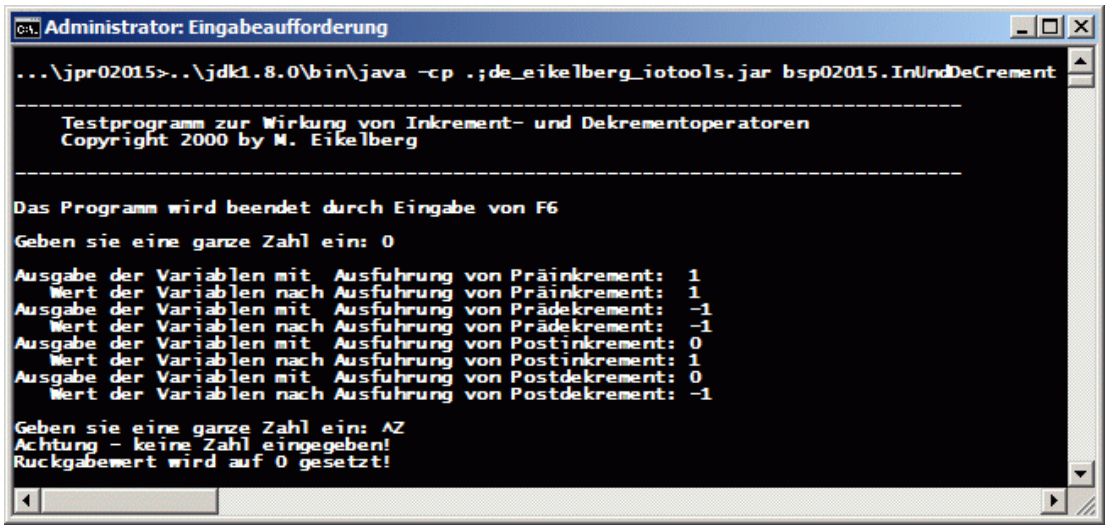
++a	Präinkrement	Ausführung von a = a+1 vor Auswertung des Ausdrucks
--a	Prädekrement	Ausführung von a = a-1 vor Auswertung des Ausdrucks
a++	Postinkrement	Ausführung von a = a+1 nach Auswertung des Ausdrucks
a--	Postdekrement	Ausführung von a = a-1 nach Auswertung des Ausdrucks

Dabei bewirkt die Zuweisung (vgl. Kapitel 2.2.4.3) `a = a + 1`, dass der auf dem Speicherplatz der Variablen abgelegte Wert gelesen, um 1 erhöht und wieder auf dem Speicherplatz abgelegt wird. Die Veränderung der Werte der Variablen ist damit dauerhaft und auch für die folgenden Teile des Programms wirksam!

Betrachte man die obige Liste, so stellt man fest, dass man die Nutzung von Inkrement und Dekrement immer dadurch vermeiden kann. Hierzu codiert man statt Inkrement bzw. Dekrement die in der obigen Liste genannte entsprechende Anweisung a=a+1 bzw. a=a-1 unmittelbar vor bzw. unmittelbar nach dem Ausdruck, in dem man die jeweilige Inkrement- oder Dekrementoperation verwenden soll. Ein

solches Vorgehen ist insbesondere für den Programmieranfänger zunächst auch durchaus empfehlenswert. Andererseits gibt es bei der Codierung von Steuerungselementen auch die Situation, dass die Verwendung von Inkrement oder Dekrement unmissverständlich und zweckmäßig ist (z. Bsp. bei Zählschleifen).

Die Wirkung von Inkrement und Dekrement kann anhand des Programms bsp2015.InUndDeCrement getestet werden. Der folgende Screenshot zeigt die Ausgabe des Programms bei einem Beispiel-Lauf:



2.3.2.3 Bindungspriorität

Sehr häufig sind in einem Ausdruck mehrere Operatoren enthalten. Fehlen vom Programmierer ausdrücklich gesetzte Klammern im Ausdruck, so stellt sich die Frage,

- wie in einem solchen Fall die Bestandteile der Ausdrücke logisch geordnet - der Mathematiker würde sagen "geklammert" - werden sollen und
- in welcher Reihenfolge diese Operatoren ausgewertet werden sollen.

Ein Beispiel für die Problematik ist der folgende Ausdruck:

```
++a*b+c*-d--
```

Dieser Ausdruck ist in einer Anweisung des Beispielprogramms bsp02017.BindungsPrioritaet enhalten. Mit Hilfe des Programms bsp02017.BindungsPrioritaet können Sie testen, welcher Wert sich als Ergebnis des obigen Ausdrucks ergibt. Die Berechnung dieses Ausdrucks wird weiter unten genauer untersucht.

Generell wird die Festlegung zur Gruppierung und Ausführungsreihenfolge von Operatoren anhand der Operator-Vorrangregeln vorgenommen. Die Operator-Vorrangregeln werden in Kapitel 2.4.6 genau erklärt. Sie basieren auf

- Werten zur Bindungspriorität der Operatoren,
- Werten zur Assoziativität der Operatoren und
- den Regeln zu Prä- und Postinkrement bzw. Prä- und Postdekrement.

In allen Tabellen zu Operatoren wird entsprechend zu jedem Operator ein Wert zu Bindungspriorität und Assoziativität genannt. Generell gilt, dass man sämtliche sich aus Bindungspriorität und Assoziativität ergebenden Gruppierungen der Operatoren durch explizites Setzen von runden Klammern überschreiben kann.

In diesem Kapitel befassen wir uns zunächst mit der **Bindungspriorität**, die übrigen Operator-Vorrangregeln werden in Kapitel 2.4.6 erklärt. Bis zur vollständigen Erklärung der Operator-Vorrangregeln sollte die Gruppierung der Operatoren zunächst explizit durch Codieren von runden Klammern im jeweiligen Ausdruck erfolgen.

Die in den Tabellen zu Operatoren eingetragenen Werte zur Bindungspriorität können wie folgt interpretiert werden: Je *niedriger* die als Bindungspriorität genannte Zahl ist, desto *enger* ist die logische Bindung des Operators. Statt des Begriffs Bindungspriorität werden in der Literatur in der Regel die Begriffe **Priorität** und **Bindungskraft** verwendet. Diese beiden Begriffe sind jedoch nach Ansicht des Autors missverständlich, weshalb in diesem Kurs der Begriff **Bindungspriorität** verwendet wird.

Die Bindungspriorität regelt also die **Gruppierung von Operatoren unterschiedlicher Bindungspriorität** (wohingegen die Assoziativität die Gruppierung von Operatoren gleicher Bindungspriorität festlegt): In einem Ausdruck werden zunächst alle Operatoren mit der Bindungspriorität 1 gruppiert, anschließend die Operatoren der Bindungspriorität 2 usw. Dies hat die gleiche Wirkung, als wenn man zunächst alle Ausdrücke der Bindungspriorität 1 mit Klammern umschließen und dies anschließend schrittweise fortsetzen würde mit allen nachfolgenden höheren Bindungsprioritäten.

Man beachte allerdings, dass beim Programmieren sehr schwer lesbare Ausdrücke entstehen können oder sogar schwerwiegende Irrtümer wirksam werden, wenn man zu komplizierte Ausdrücke schreibt, sich dabei zu sehr auf die Bindungsprioritäten verlässt und sich dabei irrt. Vielmehr gilt folgende Regel:

- In großen Ausdrücken,
- wenn man sich beim Programmieren in Bezug auf die Bindungsprioritäten nicht ganz sicher ist oder aber einfach
- zur Verbesserung der Klarheit des Programmcodes

ist es unabhängig von der Frage der Bindungsprioritäten daher oft eine gute Idee, die Reihenfolge der Ausführung von Operationen mit Hilfe von Klammern festzulegen bzw. zu beschreiben. Hierzu können im Programmcode innerhalb eines Ausdrucks Paare von runden Klammern

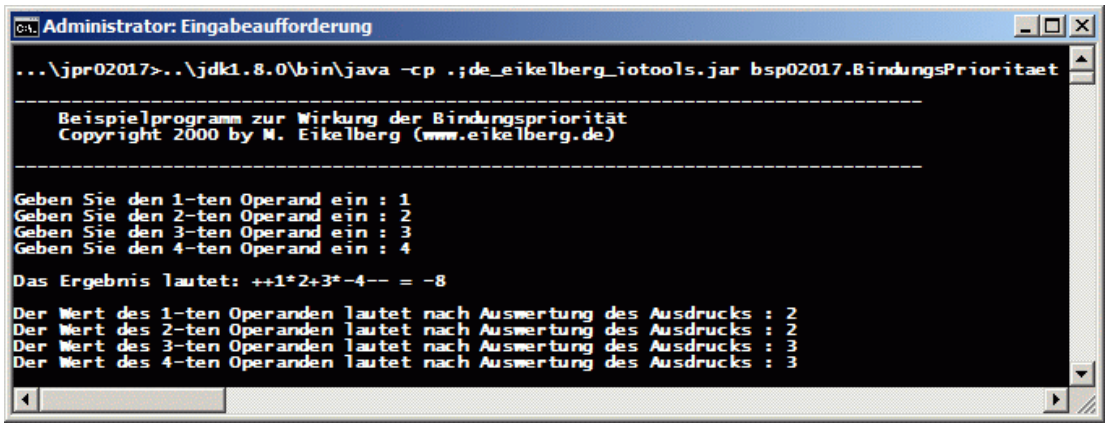
```
( ... )
```

eingefügt werden. Die so eingefügten Gruppierungen haben Vorrang vor den durch Bindungspriorität und Assoziativität festgelegten Gruppierungen.

Als Beispiel betrachten wir jetzt wieder den obigen Ausdruck

```
++a*b+c*-d--
```

Der Ausdruck wird im Programm bsp02017.BindungsPriorität ausgerechnet. Der folgende Screenshot zeigt einen Beispiellauf des Programms bsp02017.BindungsPriorität:



Diese Ergebnis kommt für Werte

a=1, b=2, c=3 und d=4

wie folgt zustande:

	++a	*****	**b**	**+**	**c**	*****	**-d--**
Vor Auswertung des Ausdrucks	a=1		b=2		c=3		d=4
Bindungspriorität 1							
Bindungspriorität 2							-4
Postdekrement (nach Bindungspriorität 2)							d=3
Bindungspriorität 3							
Präinkrement (vor Bindungspriorität 4)	a=2						
Bindungspriorität 4	4				-12		
Bindungspriorität 5	-8						
logische Gruppierung	((++a) * b) + (c * (-(d--)))						
Nach Auswertung des Ausdrucks	a=2		b=2		c=3		d=3

Man beachte, dass sich Prädekrement und Postdekrement auf die Variablen (Speicherplätze) beziehen und nicht auf die Werte der Operanden. Daher wirkt sich das Postdekrement nach Bindungspriorität 2 (d=3) nicht mehr auf den letzten Operanden (-4) des Ausdrucks aus, das heißt der nächste Auswertungsschritt ergibt c * -d-- = 3 * (-4) = -12.

Man beachte ferner, dass die Bindungspriorität nicht notwendig mit der Ausführungsreihenfolge der Operationen zusammenhängt, sondern lediglich die Gruppierung der Operatoren und Operanden regelt. Das Postdekrement wird nämlich trotz Bindungspriorität 1 erst nach der Auswertung des umgebenden Ausdrucks durchgeführt. Genauer gesagt wird eine Postinkrement- oder Postdekrementoperation genau nach Auswertung des kleinsten umgebenden Ausdrucks ausgeführt und nicht erst nach Auswertung der gesamten umgebenden Anweisung!

Alternativ zu dem Programmcode

```
x = ++a*b+c*-d--;
```

könnte man auch folgende Code-Zeilen schreiben, die die gleiche Ausführungszeit benötigen, jedoch wesentlich leichter zu lesen sind:

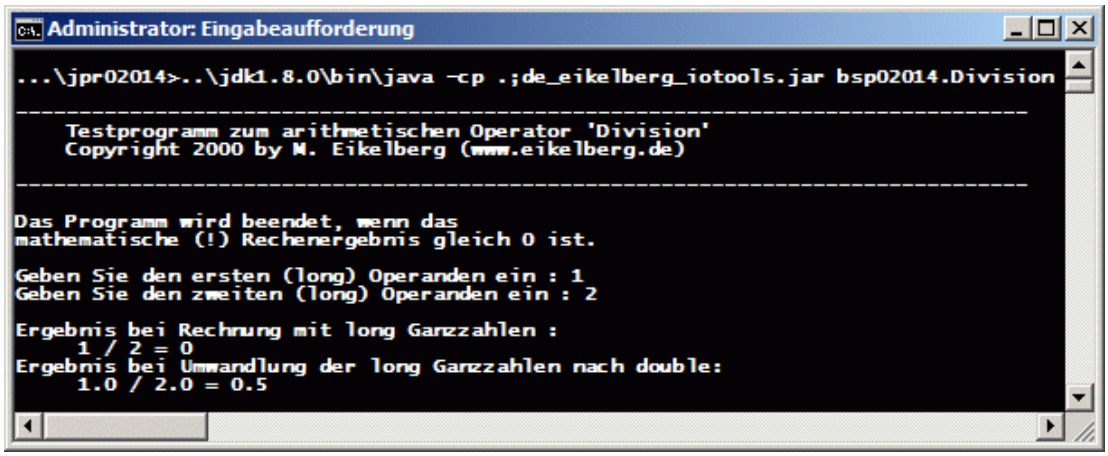
```
a = a + 1;
// Prioritäten bewirken: Punktrechnung geht vor Strichrechnung !
x = a*b-c*d;
d = d - 1;
```

Gewöhnen Sie sich auch im eigenen Interesse an, leicht verständlichen Programmcode zu schreiben und im Zweifel auch Klammern zu

setzen! Schon beim Codieren verringert sich dadurch die Zahl der Fehler erheblich, sie codieren schneller und sicherer. Auch die Zahl der beim Test und gar während der produktiven Anwendung Ihres Programms auftretenden Fehler ist dann wesentlich geringer. Ausserdem dürfte es auch Ihnen selbst unter Umständen nach einem halben Jahr schwerfallen, ihre einstmals 'geniale' Codierungsweise zu vestehen!

2.3.2.4 Besonderheiten der Divisionsoperation

Bei der Anwendung der Divisionsoperation wird der Programmieranfänger schnell mit einem Effekt konfrontiert, den es so auch in einigen anderen weit verbreiteten Programmiersprachen wie zum Beispiel FORTRAN gibt. Betrachten Sie hierzu den folgenden Screenshot eines Programmlaufs von bsp02014.Division:



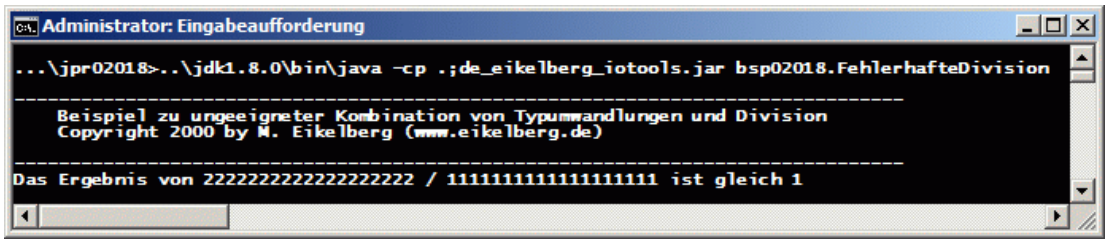
Diese Anzeige ist in einem Punkt unerwartet - das Programm rechnet 1 / 2 = 0 ! Auf der anderen Seite wird 1.0 / 2.0 = 0.5 korrekt berechnet. Des Rätsels Lösung liegt darin, *das der Datentyp des Ergebnisses der Division*

- dem Datentyp von Divisor und Dividend entspricht, wenn beide vom gleichen Datentyp sind bzw.
- gleich dem Datentyp ist, der sich bei verschiedenen Datentypen von Divisor und Dividend durch Typumwandlung als gemeinsamer umfassender Datentyp ergibt.

Insbesondere erhält man also " **Integer / Integer = Integer** "! Dabei werden sich gegebenenfalls mathematisch ergebende Nachkommastellen einfach abgeschnitten. Dies führt bei Division von ganzzahligen Operanden in der Regel zum mathematischen Rechenfehler. Die Beachtung dieses Phänomens bleibt einzig dem Programmierer überlassen - allerdings haben sich Generationen von Programmierern schnell und leicht an diese Besonderheit gewöhnt. Einen einfachen Ausweg aus der Situation bietet die explizite Typumwandlung nach float oder double. So ist entsprechend die zweite mathematisch korrekte Rechnung des Programms bsp02014.Division auch wie folgt programmiert:

```
d1 = (double)in1 / (double)in2;
```

wobei in1 und in2 Variablen für die beiden eingelesenen long Ganzzahlen sind und d1 eine double Variable ist für die mathematisch korrekt entstehende Gleitpunktzahl. Ein solches Vorgehen ist dringend zu empfehlen. Vor anderen Vorgehensweisen wird dagegen gewarnt. Insbesondere können explizite Typumwandlungen mit Genauigkeitsverlust vor der Division zu bösen Ergebnissen führen. Dies kann in einem solchen Fall zu folgendem Ergebnis führen:



Verursacht wurde dieses falsche Ergebnis durch folgenden Code im Programm bsp02018.FehlerhafteDivision:

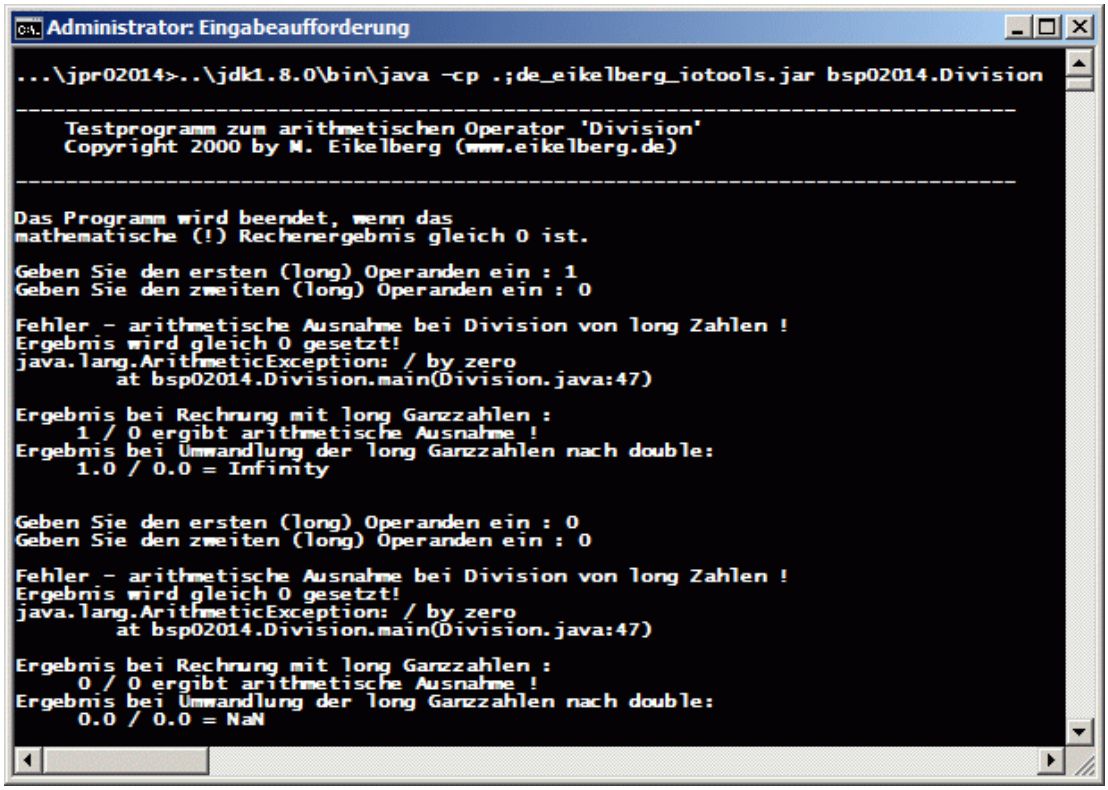
```
// Dividend und Divisor deklarieren
    long l1 = 2222222222222222222L;
    long l2 = 1111111111111111111L;

// Konvertierung von l2 nach double und wieder zurück nach
// long - führt zu geringfügigem Genauigkeitsverlust
    double d1 = (double)l2;
    long l3 = (long)d1;

// Berechnung der Division bei der Ausgabe
    ScreenOut.kPrintln(
      "Das Ergebnis von "+l1+" / "+l2+" ist gleich "+l1/l3);
```

Die Besonderheiten bei der Division durch Null sind anders als das Phänomen bei ganzzahliger Rechnung durchaus zu erwarten - es handelt sich dabei um die Behandlung der Aufgabenstellungen 1 / 0 und 0 / 0. Da Variable beliebige Zahlen enthalten können, muß bei der

Programmierung in der Regel auch mit diesen Fällen gerechnet werden. Dabei unterscheidet sich die Reaktion eines Java-Programms bei ganzzahliger Arithmetik von der Reaktion bei Gleitpunktarithmetik wie folgt:

Situation	ganzzahlige Arithmetik	Gleitpunktarithmetik
x / 0 mit x ≠ 0	Ausnahme "java.lang.ArithmeticException: / by zero"	Ergebnis "Infinity"
0 / 0	Ausnahme "java.lang.ArithmeticException: / by zero"	Ergebnis "NaN"

Wir schauen uns dieses Verhalten zunächst erst einmal anhand des Programms bsp02014.Division an:



"Infinity" und "Nan" (Not a Number) sind Werte, die eine Gleitpunktzahl gegebenenfalls annehmen kann und die - soweit sinnvoll - zum Beispiel auch bei der Ausgabe weiterverarbeitet werden können. Bei Gleitpunktarithmetik wird also die Verarbeitung des Programms fortgesetzt.

Bei ganzzahliger Arithmetik erfolgt im Gegensatz zur Gleitpunktarithmetik im Prinzip - wie in anderen Programmiersprachen auch - zunächst ein Programmabsturz. "java.lang.ArithmeticException: / by zero " - also arithmetische Ausnahme: Division durch Null - ist dabei die Meldung, die das Java-Programm beim Programmabsturz ausgibt. In Bezug auf die Programmierung in Java ist es allerdings nicht ganz sachgerecht, von einem Absturz zu reden, da ein solcher stets nicht mit einer sogenannten **Ausnahmebehandlung** vermieden wird. Man spricht daher auch nicht von einem Absturz, sondern lediglich von einer **Ausnahme**. Es ist nämlich anders als in anderen Programmiersprachen möglich, ein Java-Programm in den Zustand *vor* dem Absturz zurückzusetzen. Eine solche **Ausnahmebehandlung** ist auch im Programm bsp02014.Division programmiert, weshalb das Programm auch - wie oben zu beobachten - nicht abstürzt, sondern lediglich eine Fehlermeldung ausgibt.

Eine Ausnahmebehandlung in einem Java-Programm geschieht wie folgt:

- Zunächst veranlasst man im Programm, dass intern der Zustand des Java-Programms vor einem kritischen Schritt (auf dem Stack) gespeichert wird. Dies geschieht durch codieren des Schlüsselwortes **try**.
- Anschliessend "versucht" man den kritischen Schritt. Der oder die kritischen Schritte werden in einem Block codiert, der auf das Schlüsselwort try folgt.
- Treten Programmabsturz verursachende Situationen - in Java Terminologie **Exception** bzw. **Ausnahme** genannt - auf, so können diese Situationen abgefangen werden. Dies geschieht mit dem Schlüsselwort **catch** und unter Spezifikation der abzufangenden Ausnahme. Tritt eine Ausnahme auf, so wird der Fehler protokolliert, das Programm auf den Schritt vor `try` zurückgesetzt und es besteht die Möglichkeit einer zusätzlichen Fehlerbehandlung, der **Ausnahmebehandlung**. Die Ausnahmebehandlung wird in dem Block codiert, der auf die catch-Klausel (Klauseln sind Bestandteile von Anweisungen) folgt. Es ist möglich, zu verschiedenen Ausnahmen verschiedene catch-Klauseln mit zugehörigen Blöcken für die jeweilige Ausnahmesituation zu codieren.
- Gelingen die Schritte in dem Block nach `try`, so wird das Programm nach dem letzten Block zu direkt nachfolgenden catch-Klauseln fortgesetzt.

Das Java Sprachelement für die Ausnahmebehandlung sieht also wie folgt aus:

```
try
{
```

```
// In diesem Block werden die Anweisungen codiert, die
// einen Fehler hervorrufen könnten.
}
catch (Exception e)
{
/* Hier können die Anweisungen codiert werden, die als
   Fehlerbehandlung bei dem Auftreten einer Ausnahme vom Typ
   Exception gedacht sind.
   Zum Beispiel kann man im Konsolbildschirm anzeigen lassen,
   in welchem Objekt welcher Fehler auftreten ist. Hierzu verwendet
   man folgende Methode des Objektes e:
*/
  e.printStackTrace();
}
```

`Exception` ist eine Klasse, die Daten zu Ausnahmen aufnehmen kann und Methoden zur Analyse der Ausnahme bereitstellt. `e` ist ein Objekt der Klasse Exception, wobei als Name üblicherweise e gewählt wird, da das Objekt `e` sowieso nur in dem auf catch folgenden Block deklariert ist und damit der Name `e` jedesmal wieder verwendet werden kann. Statt der allgemeinen Klasse Exception können auch speziellere Klassen verwendet werden, in unserem Fall also die Klasse ArithmeticException. Dann gelten die Anweisungen des auf catch (ArithmeticException e) folgenden Blocks nur für den Fall einer arithmetischen Ausnahme. Weitere andersartige Ausnahmesituationen können durch zusätzlich angefügte catch - Klauseln mit nachfolgendem Block für die Behandlung der entsprechenden Fehlersituation abgefangen werden.

Im Programm bsp02014.Division sind alle diese Techniken verwendet worden. Die entsprechenden Code-Bestandteile des Programms lauten wie folgt (wieder wurden Teile des Programmcode weggelassen, die Auslassungen wurden durch Punkte .... gekennzeichnet):

```
....
// Variablendeklaration
    long in1,in2,l1;
    double d1;
    boolean lAusnahme,dAusnahme;
....
    do
    {

// Variablen für Ausnahmebehandlung zurücksetzen
      dAusnahme = false;
      lAusnahme = false;
....
// Rechenoperationen und Ausgabe bei Datentyp long
      try
      {
        l1 = in1 / in2;
      }
      catch (ArithmeticException e)
      {
        ScreenOut.kPrintln();
        ScreenOut.kPrintln
          ("Fehler - arithmetische Ausnahme bei Division von long Zahlen !");
        ScreenOut.kPrintln("Ergebnis wird gleich 0 gesetzt!");
        e.printStackTrace();
        lAusnahme = true;
        l1 = 0l;
      }

// Rechnung und Ausgabe mit vorheriger Konvertierung der Daten nach double
      try
      {
        d1 = (double)in1 / (double)in2;
      }
      catch (ArithmeticException e)
      {
        ScreenOut.kPrintln();
        ScreenOut.kPrintln
          ("Fehler - arithmetische Ausnahme bei Division von Gleitpunktzahlen !");
        ScreenOut.kPrintln("Ergebnis wird gleich 0 gesetzt!");
        e.printStackTrace();
        dAusnahme = true;
        d1 = 0.0d;
      }
....
// Ende der Schleife
```

```
    } while ( (d1 > 1e-100) || (d1 < -1e-100) );
....
```

Die letzte Code-Zeile dieses Beispiels zeigt gleichzeitig das letzte im Zusammenhang mit den arithmetischen Operationen zu beachtende Phänomen an. Diese Phänomen ist Thema des nächsten Unterkapitels:

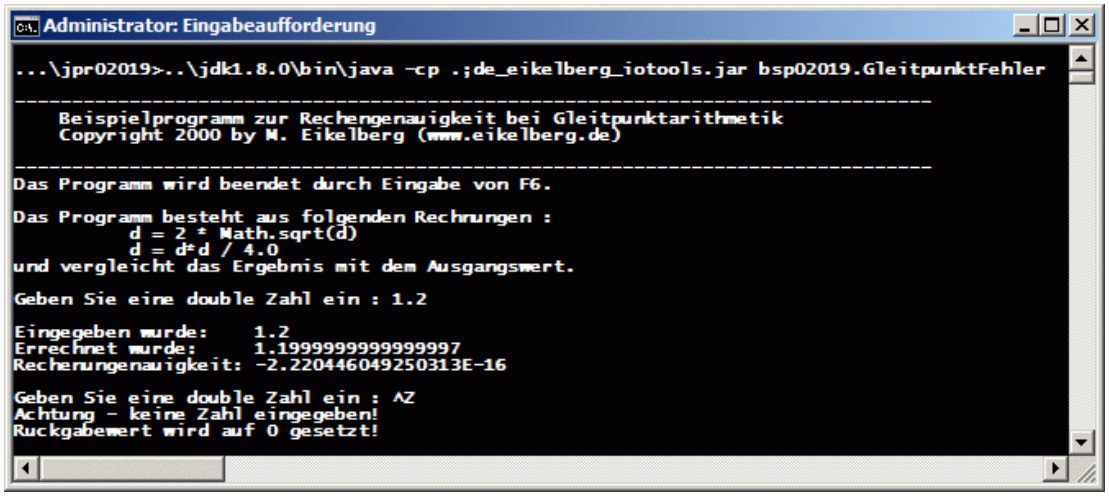
2.3.2.5 Rechengenauigkeit bei Gleitpunktarithmetik

Bei den heutigen Taschenrechnern ist das Problem der Rechengenauigkeit bei Gleitpunktarithmetik oft nicht mehr sichtbar, da sie mit mehr Stellen rechnen, als sie anschließend anzeigen. Dadurch wird der Rechenfehler nicht mehr sichtbar, der sich dadurch ergibt, dass bei Gleitpunktarithmetik oft die letzte Stelle der Zahl gerundet werden muß. Der alte Taschenrechner des Autors etwa aus dem Jahre 1978 errechnet zum Beispiel noch folgende Ergebnisse:

- Das Ergebnis von $2\sqrt{1.2}$ ist laut Taschenrechner gleich 2.1908902
- Das Ergebnis von $\frac{2.1908902^2}{4.0}$ ist laut Taschenrechner gleich 1.1999999

Damit sind Ausgangswert 1.2 und Endergebnis 1.1999999 leicht verschieden, obwohl mathematisch das Ergebnis wieder genau gleich 1.2 sein müßte. Dass heute eine solche Rechenungenauigkeit oft nicht mehr sichtbar wird bedeutet jedoch nicht, dass die entsprechende Problematik bei Gleitpunktarithmetik nicht mehr besteht. Vielmehr sind in diesen Fällen - oft programmtechnische - Maßnahmen getroffen worden, um die Rechenungenauigkeit zu verstecken.

Als Programmierer ist man in der Regel jedoch nach wie vor mit dem Problem der Rechenungenauigkeit bei Gleitpunktarithmetik konfrontiert. Wie leicht das Problem entstehen kann, zeigt das Beispielprogramm bsp02019.gleitpunktFehler. Es führt die gleiche Berechnung durch, wie sie bei obigem Taschenrechner-Beispiel vorgenommen wurde. Der folgende Screenshot zeigt die Ausgabe des Programms bei einem Beispiel-Lauf:



Besonders kritisch wird es, wenn im weiteren Programm eine Abfage codiert würde, ob der eingegebene Wert nach den entsprechenden Rechnungen wieder den Wert 1.2 ergibt. Eine solche Abfrage - wir werden später lernen, wie dies codiert werden könnte - würde sich dann der Wert 'false' ergeben, da streng logisch 1.2 und 1.1999999 natürlich nicht gleich sind. Auf diese Weise entstehen viele Laufzeitfehler in Programmen von Programmieranfängern! Die gleiche Symptomatik tritt bei einer Abfage auf, ob ein im Programm errechneter Gleitpunktwert *gleich Null* ist. Auch hier würde sich im Programm in der Regel nicht genau der Wert 0.0 ergeben, sondern eine betragsmäßig sehr kleine Zahl in der Größenordnung der akkumulierten Rechenfehler. Diese Zahl gibt das Programm bsp02019.gleitpunktFehler für die konkrete Rechnung als 'Rechenungenauigkeit' an.

Es besteht damit die Notwendigkeit einer Fehlerabschätzung. Hierzu gibt es geeignete mathematische Verfahren, zum Beispiel die Fehlerabschätzung mit Hilfe des totalen Differentials (vgl. Vorlesung Mathematik, Analysis Kapitel 4.1.5). Eine entsprechende mit Hilfe der Computeralgebrasoftware Maple V Rel 5.1 durchgeführte Abschätzung des Fehlers finden sie - nach HTML konvertiert - **hier**. Diese Berechnung ergibt ausgehend von einer achtstelligen Rechengenauigkeit einen betragsmäßig maximalen Fehler von 0.0000001. Dieser Rechenfehler entsteht bei der obigen Berechnung mit dem Taschenrechner auch tatsächlich. Das Java-Programm bsp02019.gleitpunktFehler arbeitet aufgrund der Verwendung des Datentyps double jedoch in jedem Rechenschritt mit einer weitaus größeren Genauigkeit und unterbietet damit auch insgesamt den abgeschätzten Fehler erheblich. Bezogen auf die Frage, wann eine Zahl als gleich Null gewertet werden soll, könnte jetzt im Programm eine Abfrage verwendet werden, ob der Betrag der als Null anzusehenden Zahl kleiner oder gleich 0.0000001 ist.

Eine solche Fehlerabschätzung ist natürlich aufwendig, ergibt aber einen guten Wert. In der Praxis reicht es alternativ allerdings oft aus, die Rechengenauigkeit im Programm hochzusetzen und einen grob nach oben abgeschätzten Wert als Rechenungenauigkeit anzusetzen.

2.3.2.6 Die Methoden der Klasse java.lang.Math

Betrachtet man den letzten Screenshot, so sieht man ein, dass das Programm bsp02019.gleitpunktFehler die Methode `Math.sqrt()` verwendet. Genauer gesagt handelt es sich um die (statische) Methode srqt() der Klasse Math des Paketes java.lang. Das Paket java.lang ist stets im Programm verfügbar, braucht also nicht importiert zu werden. Ferner sind alle Methoden der Klasse Math statisch , können also verwendet werden, ohne dass ein Objekt der Klasse Math instanziiert werden muss (zu statischen Methoden siehe Kapitel 2.3.3). Ein Aufruf statischer Methoden geschieht unter Voranstellung des *Klassennamens* statt des Namens eines Objektes. Die Methode sqrt() wird also im Programm für eine double Gleitpunktvariable dVar wie folgt aufgerufen:

```
  double dVar;
  ....
  Math.sqrt(dVar);
```

Die von der Klasse java.lang.Math bereitgestellten Variablen und Methoden sind in der folgenden Tabelle aufgelistet. Bei Methoden wird zwischen den runden Klammern der Variablentyp genannt, auf den die entsprechenden Methoden angewendet werden können.

Sprachelement	Bedeutung
Math.E	double Variable - Gleitpunktdarstellung der Eulerschen Zahl e
Math.PI	double Variable - Gleitpunktdarstellung der Zahl π
Math.abs(...)	Absolutbetrag für double, float, long und int Variablen
Math.round(float)	Runden einer float Gleitpunktzahl nach integer
Math.round(double)	Runden einer double Gleitpunktzahl nach long
Math.rint(double)	Runden einer double Gleitpunktzahl nach Ganzzahl (double Rückgabewert)
Math.max(...,...)	Maximum zweier Zahlen für zwei double / float / long oder int Zahlen
Math.min(...,...)	Minimum zweier Zahlen für zwei double / float / long oder int Zahlen
Math.floor(double)	Berechnen der nächstniedrigeren ganzen Zahl
Math.ceil(double)	Berechnen der nächsthöheren ganzen Zahl
Math.sqrt(double)	Quadratwurzel
Math.pow(double,double)	Potenz "a hoch b"
Math.exp(double)	e-Funktion
Math.log(double)	natürlicher Logarithmus (Basis e)
Math.log10(double)	Zehnerlogarithmus (Logarithmus zur Basis 10)
Math.sin(double)	Sinus
Math.cos(double)	Kosinus
Math.tan(double)	Tangens
Math.asin(double)	Arcus Sinus in Bogenmaß - ergibt Werte zwischen -0.5π und +0.5π
Math.acos(double)	Arcus Kosinus in Bogenmaß - ergibt Werte zwischen 0 und π
Math.atan(double)	Arcus Tangens in Bogenmaß - ergibt Werte zwischen -0.5π und +0.5π
Math.atan2(double,double)	berechnet aus kartesischen Koordinaten (x,y) den Winkel φ zwischen der Geraden durch (x,y) und (0,0) und der x-Achse, also die zweite Koordinate der Polarkoordianten (r,φ) des Punktes (x,y)
Math.random()	Berechnen einer Zufallszahl zwischen 0.0 und 1.0
Math.IEEEremainder(double,double)	Berechnen des Divisionsrestes gemäß IEEE 754 Standard

Der Divisionsrest zweier Zahlen x1 und x2 gemäß IEEE 754 ist die Zahl x1 - x2 × n, so dass gilt:

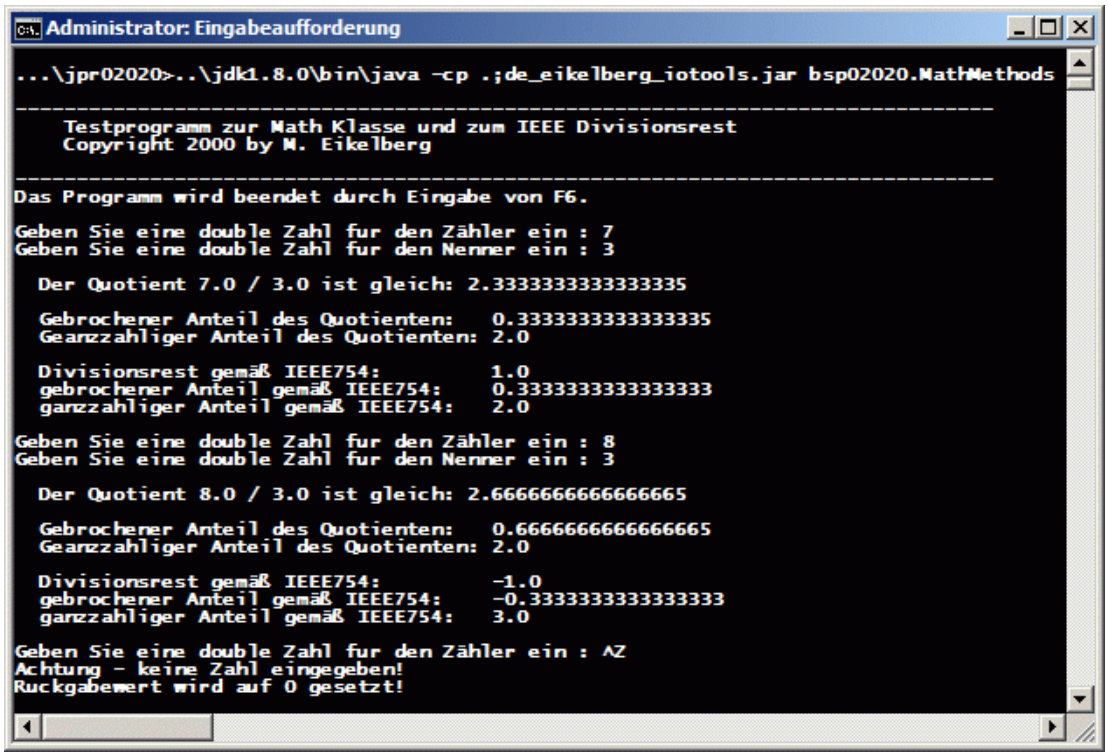
- Gibt es nur eine ganze Zahl, die dem Quotienten x1/x2 am nächsten liegt, so sei n diese Zahl.
- Liegen zwei Zahlen gleich nahe an x1/x2, so ist n diejenige der beiden Zahlen, die gerade ist.
- Wenn n gleich Null ist, so erhält n intern das Vorzeichen von x1.

Die Berechnung des Divisionsrestes gemäß IEEE 754 zu zwei eingegebenen Zahlen ist neben anderen Berechnungen Gegenstand des Beispielprogramms bsp02020.MathMethods. Alle im Programm bsp02020.MathMethods enthaltenen Aufrufe von Methoden der Klasse java.lang.Math sind in der folgenden Methode berechne() der Klasse MathMethods enthalten:

```
  public static void berechne(double d1,double d2)
  {
// Variablen für die Verarbeitung
    double bruch,ganz,res;

// Berechnung des Quotienten und Ausgabe - Fehlerbehandlung ist
// nicht notwendig wg. Gleitpunktzahlen
    bruch = d1 / d2;
    ScreenOut.kPrintln(
      "  Der Quotient "+d1+" / "+d2+" ist gleich: "+bruch);
```

```
    ScreenOut.kPrintln();
// Berechnung des gebrochenen Anteils:
    if (bruch >= 0)
    {
// bei positivem Quotienten ist der ganzzahlige Anteil kleiner
      res = (bruch - Math.floor(bruch));
    }
    else
    {
// bei negativem Quotienten ist der ganzzahlige Anteil größer !!
      res = (bruch - Math.ceil(bruch));
    }
    ScreenOut.kPrintln(
      "  Gebrochener Anteil des Quotienten:   "+res);
// Berechnung des ganzzahligen Anteils:
    ganz = bruch - res;
    ScreenOut.kPrintln(
      "  Geanzzahliger Anteil des Quotienten: "+ganz);
// Rechnungen gemäß IEEE 754
    ScreenOut.kPrintln();
// Berechnung des Divisionsrestes gemäß IEEE 754
    res = Math.IEEEremainder(d1,d2);
    ScreenOut.kPrintln(
      "  Divisionsrest gemäß IEEE754:         "+res);
// Berechnung des gebrochenen Anteils gemäß IEEE 754
    res = res / d2;
    ScreenOut.kPrintln(
      "  gebrochener Anteil gemäß IEEE754:    "+res);
// Berechnung der 'nächsten' ganzen Zahl gemäß IEEE 754 Divisionsrest
    ganz = bruch - res;
    ScreenOut.kPrintln(
      "  ganzzahliger Anteil gemäß IEEE754:   "+ganz);
  }
```

Der folgende Screenshot zeigt die Ausgabe des Programms bsp02020.MathMethods bei einem Beispiel-Lauf:



2.3.2.7 Rundung von Gleitpunktzahlen

Die in Kapitel 2.3.2.6 vorgestellte Methode Math.round(...) rundet nur auf ganze Zahlen. Oft wird jedoch eine Rundung auf eine bestimmte Anzahl von Stellen nach dem Komma benötigt. Diese Rundung kann als **Rundung auf n Nachkommastellen** oder als **Rundung auf n von Null verschiedene Stellen** erforderlich sein. Diese beiden Rundungsarten können wie folgt durchgeführt werden:

2.3.2.7.1 Rundung auf n Nachkommastellen

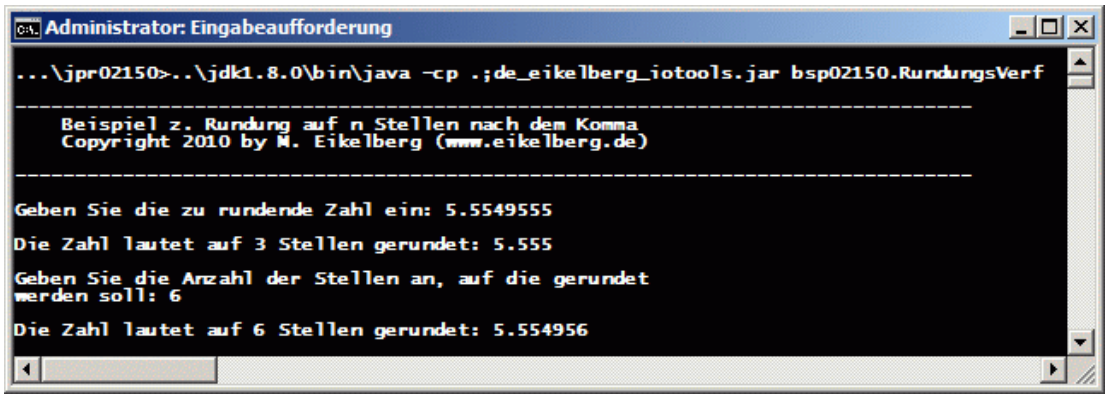
Die Rundung auf n Nachkommastellen kann mit folgenden Anweisungen erreicht werden (vgl jpr02150):

```
    double y = ...
    y = y * 1e3;
    y = Math.round(y);
    y = y / 1e3;
```

Mit diesen Anweisungen wird eine Rundung auf 3 Stellen nach dem Komma codiert, wobei die Stellenzahl fest im Programm codiert ist, auf die gerundet werden soll. Soll die Stellenzahl erst zur Laufzeit festgelegt werden, kann die Rundung wie folgt durchgeführt werden (vgl jpr2150):

```
// Einlesen der Gleitpunktzahl
    ScreenOut.kPrint("\nGeben Sie die zu rundende Zahl ein: ");
    double x = KeyboardIn.kReadDouble();

    ...

// Einlesen der Stellenzahl
    ScreenOut.kPrint(
        "\nGeben Sie die Anzahl der Stellen an, auf die gerundet\n"+
        "werden soll: ");
    int n = KeyboardIn.kReadInt();

// Rundung auf n Stellen
    y = x;
    double d = Math.pow(10, n);
    y = y * d;
    y = Math.round(y);
    y = y / d;

    ScreenOut.kPrintln("\nDie Zahl lautet auf "+n+" Stellen gerundet: "+y);
```

Beispiel jpr2150 führt entsprechend die Rundung auf die gewünschte Stellenzahl durch:



2.3.2.7.2 Rundung auf n von Null verschiedene Stellen

Bei einer Rundung auf n von Null verschiedene Stellen muss damit gerechnet werden, dass diese Stellen nicht direkt rechts vom Dezimalpunkt liegen oder dass eine oder mehrere dieser von Null verschiedenen Stellen links vom Dezimalpunkt liegen. Daher muss zunächst festgestellt werden, wie viele von Null verschiedene Stellen der Zahl links vom Dezimalpunkt liegen. Dies kann mit dem Zehnerlogarithmus erfolgen, der ja wie oben bereits erklärtmit der Anweisung `Math.log10(...)` berechnet werden kann. Bei der Berechnung muss aber berücksichtigt werden, dass der ganzzahlige Teil des Zehnerlogarithmus die Stellenzahl minus Eins ergibt (vgl jpr02151):

```
    double x = ...

    ...

    double vorKomma = Math.log10(x)+1;
    vorKomma = Math.floor(vorKomma);
    long vorKommaStellenZahl = Math.round(vorKomma);
```

Ausgehend von der so berechneten Anzahl von Vorkommastellen kann nun die für die Rundung erforderliche Zehnerpotenz ermittelt werden:

```
// Darstellung mit 6 von Null verschiedenen Stellen erzeugen
  private final static int STELLENZAHL=6;

  public static void main(String[] args)
  {
```

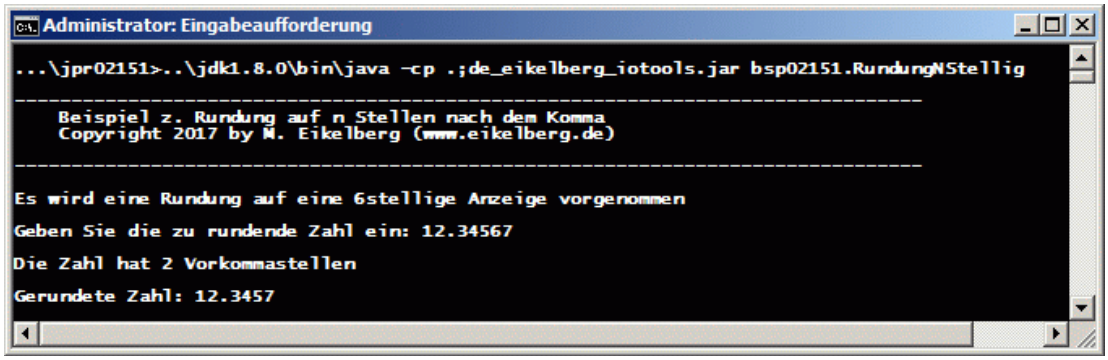
```
    ...

    double rundungsfaktor = Math.pow(10, STELLENZAHL-vorKommaStellenZahl);

    ...
  }
```

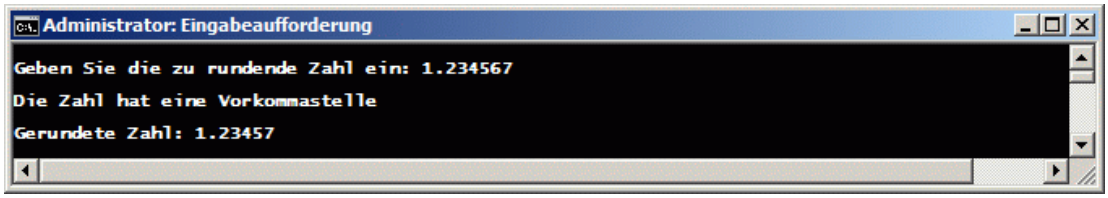
Dieser Rundungsfaktor kann nun bei der Rundung verwendet werden:

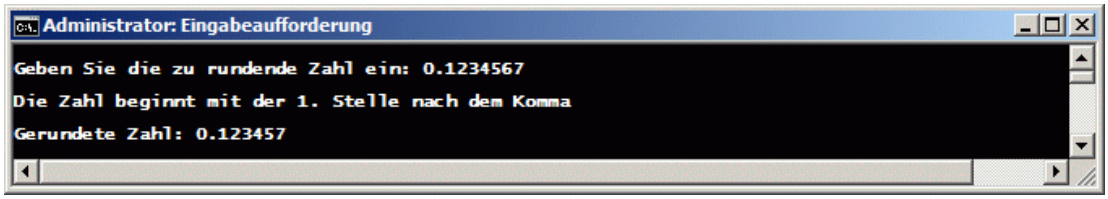
```
x = x * rundungsfaktor;
x = Math.round(x);
x = x / rundungsfaktor;     // hier kann wieder eine Rechenungenauigkeit auftreten... :-(
```

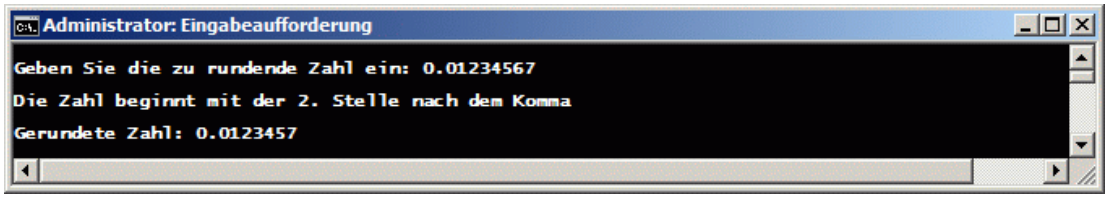
Wie im obigen Quellcode bereits als Kommentar eingetragen kann dabei im letzten Rechenschritt wieder eine Rechenungenauigkeit auftreten. In diesem Fall wird *nicht* erfolgreich gerundet. In der Variablen x ist schließtlich der hoffentlich erfolgreich gerundete Wert gespeichert.

Der gesamte obige Ablauf ist im Beispiel jpr02151 enthalten. Dieses Programm ergibt bei Tests die folgenden Rundungen:









2.3.3 Objektbasierte Programmierung

Verwendet man in einem Programm Objekte zu vorgegebenen Klassen, so spricht man von **objektbasierter Programmierung**. Eine solche Verwendung von vorgegebenen Klassen ist das Thema dieses Kapitels.

2.3.3.1 Begriffe Klasse und Objekt

Mit den Begriffen **Klasse** und **Objekt** wird die vom Programm fachlich zu verarbeitende Realität typisiert, damit sie anschließend im Programm abgebildet werden kann. Hierzu typisiert man wie folgt die in der Realität auftretenden Dinge:

Beispiel: Zwei Benutzer Tobi Test und Paula Probe sind ein Spezialfall bzw. eine konkrete Ausprägung einer Person.

Um also Informationen (Daten) sowie verarbeitende Verfahren (Algorithmen) der Realität in einem Computerprogramm abzubilden, ordnet man sie wie folgt themenorientiert in einem Computerprogramm an:

- Eine **Klasse** ist eine Quellcodedatei mit der Beschreibung der Daten und Algorithmen zu einem typisiert betrachteten Ding der Realität.
- Ein **Objekt** enthält die Daten einer konkreten Ausprägung dieses typisiert betrachteten Dings. Ein Objekt stellt eine konkrete Ausprägung der in der Klasse definierten Daten und Algorithmen dar. In der Informatik spricht man daher davon, dass ein Objekt ein Exemplar einer Klasse ist. Der entsprechende Merksatz lautet im Englischen:

an object is an instance of a class

Wörtlich in's Deutsche übersetzt lautet der Merksatz also

Ein Objekt ist ein **Exemplar** einer Klasse

Da das deutsche Wort Instanz neben der Bedeutung im juristischen Bereich auch die - heutzutage aber sehr ungebräuchliche - Bedeutung "Exemplar" haben kann, ist eine weitere gebräuchliche - allerdings nicht sehr sinnstiftende - Übersetzung des englischen Merksatzes wie folgt:

Ein Objekt ist eine **Instanz** einer Klasse

Beispiel:

- Eine Klasse `Person` definiert die Speicherplätze für Daten, die gemäß dem Entwurf des Computerprogramms - und soweit sie im Programm benötigt werden - gespeichert werden müssen. Zu diesen Informationen gehören zum Beispiel der Vorname, der Nachname und das Geburtsjahr einer Person. Ferner enthält die Klasse `Person` die Beschreibung der Funktionen (Algorithmen), über die eine Person im Computerprogramm verfügen soll, zum Beispiel eine Funktion für das Erzeugen eines Textes, mit dem sich die Person anderen Personen vorstellt.
- Ein Objekt `paula` der Klasse `Person` enhält die konkreten Daten einer Person Paula Probe. Im Objekt `paula` werden also der konkrete Vorname, der konkrete Nachname und das konkrete Geburtsjahr von Paula Probe gespeichert.

Im Beispielprojekt jpr02301 befindet sich entsprechend die folgende Klasse Person, in der zunächst Variablen für die Eigenschaften Vorname, Nachname und Geburtsjahr deklariert werden:

```
package bsp02301;

public class Person
{
  String vorName;
  String nachName;
  int geburtsJahr;
}
```

2.3.3.1 Instanzvariablen

Die in der Klasse bsp02301.Person deklarierten Variablen zur Speicherung der Daten einer Person heißen **Instanzvariablen**, da sie bei der Objektbildung für jedes Objekt (also für jede Instanz) einmal erzeugt werden. Eine der Grundregeln der objektorientierten Programmierung besagt, dass solche Instanzvariablen für andere Programmierer versteckt (man sagt **gekapselt** angelegt werden sollten. Hierzu stehen folgende Schlüsselwörter (sogenannte **Modifizierer** zur Verfügung:

Attribut	Wirkung
private	Die Variable bzw. die Methode ist nur in der *Klasse* verwendbar, in der sie deklariert ist.
protected	Die Variable bzw. die Methode ist nur in dem *Paket* verwendbar, in der sie deklariert ist.
public	Die Variable bzw. die Methode ist im gesamten Java-Programm verwendbar.

Wird - wie in Beispiel jrp02301 - kein Modifizierer angegeben, so spricht man von der Eigenschaft **default** bzw. **package private**. Die folgende Tabelle zeigt die genaue Wirkung der Modifizierer bzw. die Wirkung der Eigenschaft package private:

Modifizierer	Zugriff aus der eigenen Klasse	Zugriff aus dem eigenen Paket	Zugriff aus einer abgeleiteten Klasse	Zugriff aus einer beliebigen Klasse
`public`	Ja	Ja	Ja	Ja
`protected`	Ja	Ja	Ja	Nein
default (kein Modifizierer)	Ja	Ja	Nein	Nein
`private`	Ja	Nein	Nein	Nein

Im Beispiel jpr02302 wurden wie folgt die Variablen der Klasse Person als private gekennzeichnet, um sie gemäß den Regeln der objektorientierten Programmierung zu kapseln - sie werden mit der Markierung als `private` vor dem Zugriff aus anderen Klassen zu geschützt:

```
package bsp02302;

public class Person
{
  private String vorName;
  private String nachName;
  private int geburtsJahr;

  ...
}
```

Allerdings besteht nun das Problem, dass auf diese Variablen aus anderen Klassen nicht mehr zugegriffen werden kann. Daher erzeugt man nun die sogenannten **get-Methoden** bzw. **set-Methoden**, auch **Getter** und **Setter** genannt:

- Eine **get-Methode** gibt den Inhalt einer Instanzvariablen zurück. Sie ist regelmäßig wie folgt codiert:

```
private Datentyp variable;

public Datentyp getVariable()
{
  return variable;
}
```

- Eine **set-Methode** nimmt Daten an und speichert sie in einer Instanzvariablen. Sie ist regelmäßig wie folgt codiert:

```
private Datentyp variable;

public void setVariable(Datentyp variable)
{
  this.variable = variable;
}
```

- Im Quellcode der set-Methode tauchen zwei gleichnamige Variablen mit dem Namen `variable` auf. Einmal tritt diese Variable als Parameter der Methode `setInstanzvariable(Datentyp variable)` auf. Diese Variable ist gültig in der gesamten set-Methode, außerhalb der Methode jedoch nicht definiert. Das zweite Mal tritt die Variable als Instanzvariable **variable** auf. Diese Variable ist in der gleichen Klasse vorhanden, wird aber in der Methode gemäß der Regel "lokal vor global" von dem gleichnamigen Parameter **variable** überdeckt. In der set-Methode kann allerdings auf die Instanzvariable **variable** zuzugegriffen werden durch Voransetzen des "Zeigers" this. Dieser Zeiger kann auch dann verwendet werden, wenn die Instanzvariable nicht überdeckt wird.

In Beispiel jpr02302 lauten die get- und set-Methoden also wie folgt:

```
package bsp02302;

public class Person
{
  private String vorName;
  private String nachName;
  private int geburtsJahr;

  public String getVorName()
  {
    return this.vorName;
  }
  public void setVorName(String vorName)
  {
```

```
    this.vorName = vorName;
  }

  public String getNachName()
  {
    return this.nachName;
  }
  public void setNachName(String nachName)
  {
    this.nachName = nachName;
  }

  public int getGeburtsJahr()
  {
    return this.geburtsJahr;
  }
  public void setGeburtsJahr(int geburtsJahr)
  {
    this.geburtsJahr = geburtsJahr;
  }

  ...
}
```

2.3.3.2 Instanzmethoden

Algorithmen, die jedes Ding der Realität separat ausführen kann, können gemeinsam mit den Daten des Dings in der Klasse programmiert werden. Man spricht dann von **Instanzmethoden**. Diese Instanzmethoden können mit den gleichen Modifiziern gekennzeichnet werden wie Instanzvariablen. Beispiel jpr02302 enthält folgende Instanzmethode:

```
package bsp02302;

public class Person
{
  private String vorName;
  private String nachName;
  private int geburtsJahr;

  ...

  public String stelleDichVor()
  {
    String vorst;
    vorst = "Ich heisse "+vorName+" "+nachName+" und bin im Jahr "+geburtsJahr+" geboren.";
    return vorst;
  }
}
```

2.3.3.3 Instanziierung von Objekten, Konstruktoren

Für die konkrete Speicherung von Daten müssen anhand der Deklaration von Variablen und Methoden nun Objekte gebildet werden. Man spricht von **Instanziierung**. Die Instanziierung eines Objektes erfolgt in zwei Schritten,

- der Deklaration der Objektvariablen sowie
- der eigentlichen Instanziierung des Objektes mit Zuweisung der Objektadresse in die Objektvariable.

Die Syntax für diesen Vorgang ist im Allgemeinen im einfachsten Fall wie folgt:

```
KlassenName objektName;              // (1)
objektName = new KlassenName();      // (2)
```

Im Schritt (1) wird eine (Objekt-)Variable vom Datentyp `Klassenname` mit dem Namen `objektName` angelegt. In Schritt (2) wird anschließend mit der Anweisung `new KlassenName()` das Objekt angelegt. Dabei zeigt der new-Operator an, dass an der nachfolgenden Stelle ein Objekt angelegt wird. Die Methode `KlassenName()` hat den gleichen Namen wie die Klasse und heisst **(parameterloser) Konstruktor**. Ein parameterloser Konstruktor kann vom Programmierern in der Klasse codiert werden (Beispiel jpr02303). Ansonsten wird der parameterlose Konstruktor vom Compiler automatisch angelegt (Beispiel jpr02302). Das Ergebnis des Konstruktoraufrufs ist eine Speicheradresse, die mit der Zuweisung in der Anweisung (2) in der Objektvariablen `objektName` gespeichert wird.

Beispiel jpr02302 enthält neben der Klasse bsp02302.Person auch eine Klasse bsp02302.PersonenVerarbeitung, die eine main-Methode mit der Verarbeitung des Programms enthält. In dieser Verarbeitung wird wie folgt

- ein Objekt `tobi` der Klasse `Person` angelegt,
- dieses Objekt unter Verwendung von set-Methoden mit Daten gefüllt und zuletzt
- eine verarbeitende Methode des Objektes aufgerufen:

Dabei wird folgende Regel angewendet:

> **Der Zugriff auf Instanzmethoden bzw. Instanzvariablen erfolgt stets über den Objektnamen des Objekts, für das die Verarbeitung aufgerufen wird bzw. in dem die Daten gespeichert sind.**

Dabei schreibt man zunächst den Objektnamen, nachfolgend dann einen Punkt sowie anschließend den Namen der Instanzmethode bzw. der Instanzvariable. Entsprechend steht vor dem Methodenaufruf `stelleDichVor()` der Name des Objektes (`tobi`), für das die Methode aufgerufen wird:

```
package bsp02302;

public class PersonenVerarbeitung
{
  public static void main(String[] args)
  {
    ...

    Person tobi;
    tobi = new Person();

    tobi.setVorName("Tobi");
    tobi.setNachName("Test");
    tobi.setGeburtsJahr(2000);

    String hallo= tobi.stelleDichVor();
    System.out.println("\n"+hallo+"\n");
  }
}
```

Ein Programmlauf von Beispiel jrp02302 führt entsprechend zu folgender Konsolausgabe:

Im Beispiel jpr02302 wird das Objekt `tobi` mit Hilfe von set-Methoden mit Daten gefüllt. Dies ist aber auch über einen sogenannten `parameterbehafteten Konstruktor` möglich. Ein parameterbehafteter Konstruktor ist ein Konstruktor mit Parametern, über die Informationen vom Konstruktor angenommen werden. Diese Informationen werden in der Regel in der Verarbeitung des Konstruktors in Instanzvariablen kopiert. Dies geschieht auch im parameterbehafteten Konstruktor der Klasse bsp02303.Person:

```
public class Person
{
  ...

  public Person(String vorName, String nachName, int geburtsJahr)  // parameterbehafteter
Konstruktor
  {
// Kopieren der Daten aus den Parametern in (gleichnamige) Instanzvariable
    this.vorName = vorName;
    this.nachName = nachName;
    this.geburtsJahr = geburtsJahr;
    System.out.println("Objekt der Klasse Person wird mit parameterbehafteten Konstruktor
angelegt...");
  }

  private String vorName;
  private String nachName;
  private int geburtsJahr;

  ...
}
```

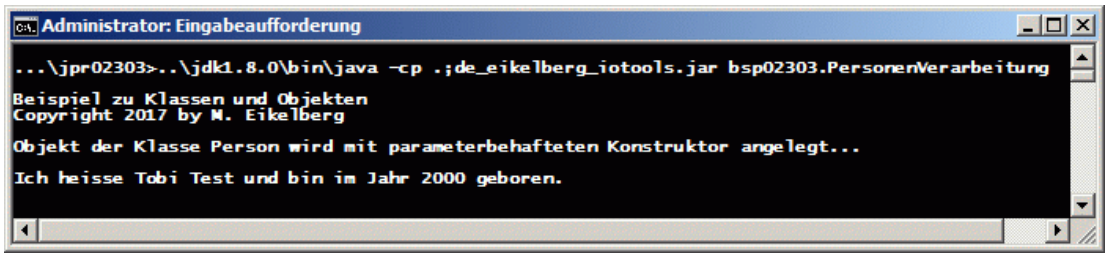
Der Quellcode im parameterbehafteten Konstruktor

`public Person(String vorName, String nachName, int geburtsJahr)` besteht zunächst aus drei Anweisungen, die die beim Aufruf des Konstruktors angegebenen Daten - wie bei einer set-Methode - in die Instanzvariablen des Objektes kopieren. Ferner enthält der Konstruktor noch eine verarbeitende Anweisung mit einer Konsolmeldung.

In main-Methode der Klasse bsp02303.PersonenVerarbeitung wird nun der parameterbehaftete Konstruktor wie folgt verwendet:

```
package bsp02303;

public class PersonenVerarbeitung
{
  public static void main(String[] args)
  {
    ...

    Person tobi;
    tobi = new Person("Tobi","Test",2000);

    String hallo= tobi.stelleDichVor();
    System.out.println("\n"+hallo+"\n");
  }
}
```

Beim Lauf des Programms in Beispiel jpr02303 erscheint nun anders als beim Programmlauf zum Projekt jpr02302 zusätzlich die Meldung, die vom parameterbehafteten Konstruktor in die Konsolbox geschrieben wird:



2.3.3.4 Klassenvariablen und Klassenmethoden

Klassenvariablen bzw. **Klassenmethoden** werden auch statische Variablen bzw. statische Methoden genannt, entsprechend bezeichnet man Instanvariablen bzw. Instanzmethoden auch als nicht statische Variablen bzw. nicht statische Methoden. Die Eingenschaft **statisch** erhalten diese Variablen und Methoden mit Hilfe des folgenden Modifizierers:

Attribut	Wirkung
static	Die Variable bzw. Methode wird nicht für jedes Objekt neu angelegt, sondern nur einmal angelegt und dann in allen Objekten der Klasse verwendet. Die Variable bzw. die Methode kann auch dann verwendet werden, wenn keine Instanz der Klasse erzeugt wurde, in der die Variable bzw. die Methode deklariert wurde. Der Zugriff auf statische Variablen und Methoden sollte stets über den Klassennamen erfolgen. Nicht statische Variablen und Methoden werden für jedes Objekt neu erzeugt, auf sie wird über den Objektnamen zugegriffen.

Es ist programmtechnisch möglich, über Objektnamen auf statische Variablen oder Methoden zuzugreifen. Das gilt aber als schlechter Programmierstil denn es suggeriert, dass die entsprechenden Variablen bzw. Methoden einem Objekt "gehören" würden. Das ist aber nicht der Fall, vielmehr sind diese Variablen und Methoden *für alle Objekte insgesamt nur einmal vorhanden.* Daher gilt folgende Regel:

> **Der Zugriff auf Klassenmethoden sowie Klassenvariablen sollte stets über den Namen der Klasse erfolgen, in der die Methode bzw. die Variable codiert ist.**

Abgesehen davon, dass eine statische Variable nur einmal für alle Objekte angelegt wird, wird mit dem Modifizierer static aber auch die **Lebensdauer** verändert - also die Frage, in welchen Bereichen des Programms die Variablen und Methoden verwendbar sind:

- Verwendet ein Programmierer static - deutsch statische - Variablen und Methoden, so legt der Compiler diese Variablen und Methoden automatisch beim Programmstart an. Damit stehen statische Methoden und Variablen während des gesamten Programmablaufs zur Verfügung.
- Nicht statische Variablen und Methoden stehen dagegen nur zur Verfügung, wenn ein Objekt der umgebenden Klasse instanziiert wurde. Nicht statische Variablen und Methoden 'leben' als nur von der Instanziierung dieses Objektes bis zur Destruktion des Objektes. Die Destruktion - also die Entfernung des Objektes - erfolgt in der Regel automatisch sobald die virtuelle Maschine erkennt, dass auf das Objekt nicht mehr zugegriffen wird.

Die Klasse bsp02304.Person enthält eine statische Variable sowie eine statische Methode:

```
package bsp02304;
```

```
public class Person
{
  public Person()
  {
    this.bestimmePersonennummer();
    System.out.println("Person Nummer "+this.personenNummer+" wurde im DV-System angelegt.");
  }

  public Person(String vorName, String nachName, int geburtsJahr)
  {
    this.vorName = vorName;
    this.nachName = nachName;
    this.geburtsJahr = geburtsJahr;
    this.bestimmePersonennummer();
    System.out.println(
      "Person "+this.vorName+" "+this.nachName+" wurde "+
      "mit Personennummer "+this.personenNummer+" im DV-System angelegt."
    );
  }

  private String vorName;
  private String nachName;
  private int geburtsJahr;

  private int personenNummer = 0;          // jede Person hat eine eigene Personennummer
  private static int personenZaehler = 0;  // der Zähler ist für alle Personen insgesamt
                                           // nur einmal vorhanden
  private void bestimmePersonennummer()
  {
    Person.personenZaehler++;
    this.personenNummer = Person.personenZaehler;
  }

  ...

  public static String getPersonenZaehlerInfo()
  {
    String ret;
    ret = "Anzahl der im DV-System angelegten Personen ist: "+Person.personenZaehler;
    return ret;
  }
}
```

Im obigen Quelltext befindet sich zunächst die statische Variable `personenZaehler`. Diese Variable ist für alle Objekte insgesamt nur einmal vorhanden und kann von allen Objekten verwendet werden. Sie enthält im Beispiel einen Zähler, der die Anzahl im DV-System angelegten Personen enthält. Über die Anweisung

```
this.personenNummer = Person.personenZaehler;
```

wird der jeweils aktuelle Zählerwert in die Instanzvariable `personenNummer` kopiert. Diese Variable ist für jedes Objekt einmal vorhanden - da Instanzvariable - und enthält somit für jedes Objekt dann die jeweils individuelle Personen-Nummer.

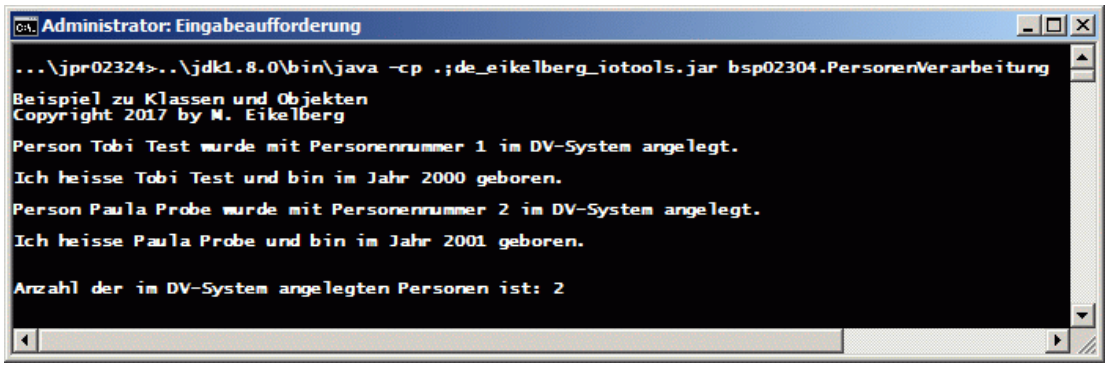
Ferner enthält die Klasse bsp02304.Person die statische Methode `getPersonenZaehlerInfo()`, auf die in anderen Klassen wie folgt zugegriffen werden kann: Für einen Zugriff auf eine statische Methode oder Variable codiert entsprechend der obigen Regel man zunächst den Klassennamen, nachfolgend dann einen Punkt sowie anschließend den Namen der Klassenmethode bzw. der Klassenvariablen. Das entsprechende Vorgehen zeigt die Verarbeitung in der main-Methode der Klasse bsp02304.PersonenVerarbeitung:

```
package bsp02304;

public class PersonenVerarbeitung
{
  public static void main(String[] args)
  {
    ...

    Person tobi;
    tobi = new Person("Tobi","Test",2000);

    String hallo= tobi.stelleDichVor();
    System.out.println("\n"+hallo+"\n");

    Person paula;
```

```
    paula = new Person("Paula","Probe",2001);

    hallo= paula.stelleDichVor();
    System.out.println("\n"+hallo+"\n");

    String info = Person.getPersonenZaehlerInfo();
    System.out.println("\n"+info+"\n");
  }
}
```

Bei einem Programmlauf wird daraufhin angezeigt,

- dass Tobi Test die Personennummer 1 hat,
- dass Paula Probe die Personennummer 2 hat und
- dass am Ende des Programmlaufs 2 Personen im Programm gespeichert sind:



2.3.3.5 Finale Variablen

Klassen- und Instanzvariablen können mit dem Modifizierer `final` als schreibegeschützt markiert werden. In diesem Fall muss diesen Variablen direkt bei der Deklaration ein Wert zugewiesen werden. Im weiteren Programmlauf kann dann dieser Wert nicht mehr verändert werden:

Attribut	Wirkung
final	Gilt nur für Variablen. Einer Variablen vom Typ final kann nur einmal bei der Deklaration ein Wert zugewiesen werden. Dieser Wert kann anschließend nicht mehr verändert werden. Finale Variablen nennt man auch **Konstanten**. Für die Variablennamen von Konstanten wählen Java-Programmierer stets Variablennamen, die vollständig aus Großbuchstaben bestehen.

Eine Prinzip der Objektorientierten Programmierung ist - wie bereits oben erwähnt - das Prinzip der Kapselung. Dieses Prinzip besagt, dass Klassen- und Instanzvariablen nicht außerhalb einer Klasse sichtbar sein (also `private` gesetzt werden) sollten. Konstanten sind aber von dieser Regelung ausgenommen, da die Werte der Konstanten nicht veränderbar sind.

Konstanten werden verwendet, um für Berechnungen benötigte feste Werte im Programm zu definieren, es gibt also zum Beispiel in der Klasse `Math` die Konstanten `Math.PI` für einen Näherungswert für die Zahl π bzw. `Math.E` für einen Näherungswert zur Eulerschen Zahl e.

Konstanten können aber auch vom Programmierer codiert werden, um Einstellungen für ein Programm festzuglegen. So ist in der Klasse bsp02232.Vorzeichen eine Konstante `EPS` enthalten mit der festgelegt wird, dass alle Werte kleiner oder gleich `EPS` vom Programm als gleich Null betrachtet werden sollen:

```
package bsp02232;

...

public class Vorzeichen
{
  private static final double EPS = 1e-12;

  public static void main(String[] args)
  {
    ...

    double d;

    ...

    if (d > EPS)
```

```
    {
      ScreenOut.kPrintln("Die Zahl ist größer Null.");
    }
    else
    {
      ...
    }
  }
}
```

2.3.3.6 Beispiel - System.out.println()

In Kapitel 1 wurde die Java Standardmethode System.out.println() zur Ausgabe in der Konsolbox verwendet. Diese Methode hat einige Nachteile und wurde in den bisherigen Beispielen in Kapitel 2 durch die Methoden de.eikelberg.iotools.kPrint() bzw. de.eikelberg.iotools.kPrintln() ersetzt (vgl. Kapitel 2.3.5). Wenn also die Verwendung von System.out.println() in den nächsten Kapitel in der Regel nicht mehr erfolgen soll, so soll hier dennoch die Struktur dieses Methodenaufrufs erklärt werden.

Der Ausdruck `System.out.println()` setzt sich aus folgenden Bestandteilen zusammen:

1. `System` ist eine Klasse des Paketes `java.lang`. Das Paket `java.lang` wird automatisch vom Compiler verfügbar gemacht, das heisst es ist zur Nutzung dieses Paketes keine import-Anweisung erforderlich.
2. `out` ist eine statische und finale public Variable der Klasse `System`. Damit ist `out` unabhängig von Instanzen der Klasse `System` in Programmen verfügbar.
3. `out` ist seinerseits Instanz der Klasse `PrintStream`. Man sieht hier, dass in allgemeiner Terminologie Objekte auch als Variablen aufgefasst werden. Damit kann es auch statische Objekte geben.
4. `println()` ist eine Instanzmethode der Klasse PrintStream und damit eine Methode des Objektes `out`. Da `out` bereits statisch ist, steht `println()` bereits nach der automatischen Instanziierung des statischen Objektes `out` zur Verfügung.

Insgesamt gilt also: println() ist eine Methode des Objektes out, das seinerseits statische Variable der Klasse System ist. out wird als statische Objekt-Variable automatisch instanziiert, System braucht nicht instanziiert zu werden, da lediglich ein Zugriff auf eine statische Variable von System erfolgt.

Wie man sieht, ist also der Aufruf System.out.println() intern ein relativ verzwicktes Konstrukt, was den Aufruf selbst aber nicht verkompliziert. Sollten Sie die Erklärung der Struktur von System.out.println() nicht vollständig verstanden haben, so ist dies für das weitere Vorgehen nicht tragisch. Derart anspruchsvolle Konstrukte der objektorientierten Programmierung werden zumindest bis Ende von Kapitel 2 nicht weiter auftreten. Einfachere Beispiele zum Aufruf statischer und nicht statischer Methoden finden Sie im Zusammenhang mit der Erklärung der Methoden der Klassen des Paketes de.eikelberg.iotools (vgl. Kapitel 2.3.5).

Interessierte Leser finden im Paket bsp02007 die beiden Beispielprogramme bsp02007.Analyze1 und bsp02007.Analyze2. In bsp02007.Analyze1 wird eine Zerlegung des Ausdrucks System.out.println() vorgenommen und dabei das statische Objekt out verwendet. Alternativ dazu erfolgt in bsp02007.Analyze2 statt der Nutzung des statischen Objektes out der Klasse PrintStream die Instanziierung eines neuen Objektes der Klasse PrintStream.

2.3.4 Das Paket de.eikelberg.iotools

Vorbemerkung: Die Klassen des Paketes de.eikelberg.iotools korrigieren (soweit möglich) das in Kapitel 2.2.3 beschriebene Problem der unterschiedlichen Codierung von Schriftzeichen in Windows-Betriebssystemen gemäß Code Page 1252 (Windows Fenster) bzw. Code Page 850 (Konsolbox). Dieses Problem betrifft wie in Kapitel 2.2.3 erklärt insbesondere die Darstellung der deutschen Umlaute 'ä','ö','ü','Ä','Ö','Ü' sowie des Zeichens 'ß'. Bei Anwendung der Klassen des Paketes de.eikelberg.iotools werden die genannten Sonderzeichen korrekt verarbeitet mit Ausnahme des Zeichens 'ü' in der Konsolbox. Dieses kann ab Java-Version 1.2 prinzipiell nicht mehr von der Java virtuellen Maschine in der Konsolbox ausgegeben werden. Daher geben die Klassen des Paketes de.eikelberg.iotools statt des Zeichens 'ü' in der Konsolbox das Zeichen 'u' aus.

2.3.4.1 Problemstellung

Die Programmausgabe und insbesondere das Einlesen von Daten in ein Java-Konsolprogramm sind sowohl bei der Verwendung von Bildschirm und Tastatur als auch bei der Verwendung von Dateien für einen Programmieranfänger alles andere als einfach und auch für einen geübten Programmierer mit wesentlichem Aufwand verbunden. Dies liegt daran, daß bei der Verwendung der Methoden des Standard Sprachumfanges von Java bei der Ein- und Ausgabe im Prinzip überwiegend nur einzelne Schriftzeichen (Datentyp int bzw. char) bearbeitet werden und es wenig vorbereitete Methoden auf höherem Niveau gibt.

Zudem berücksichtigen die Methoden von Java in keiner Weise die in Kapitel 2.2.3 beschriebene Problematik unterschiedlicher Codierungen. Vielmehr setzten diese Methoden voraus, dass das zugrundeliegende Betriebssystem Unicode codiert oder aber zumindest nur eine einzige Codierung verwendet. Aus diesen sowie aus den in Kapitel 2.2.3 beschriebenen Gründen funktionieren die Java Standardmethoden für die Ein- und Ausgabe von Daten zum Beispiel unter Windows nicht zufriedenstellend sondern ergeben oft falsche Schriftzeichen.

Um also einerseits die Ein- und Ausgabe unter Java zu vereinfachen sowie andererseits zur Lösung der besonderen Problematik unter Windows ist vom Autor das bereits erwähnt Paket **de.eikelberg.iotools** entwickelt worden. Rechtlich leider notwendig sind diesbezüglich an dieser Stelle folgende Bemerkungen:

2.3.4.2 Juristische Bemerkungen

- Die Software des Paketes de.eikelberg.iotools ist Freeware für private Zwecke und für den Einsatz an öffentlich-rechtlichen Schulen und Hochschulen. Eine kommerzielle Nutzung des Paketes bedarf des schriftlichen Einverständnisses des Autors und kann kostenpflichtig sein.
- Der Autor übernimmt bei einer Verwendung der Software als Freeware keinerlei Gewährleistung für Fehler jeglicher Art, die aus der Nutzung des Paketes entstehen. Jegliche diesbezuegliche Haftung wird ausdrücklich ausgeschlossen. Treten bei der Anwendung des Paketes Fehler auf, so bittet der Autor um einen qualifizierten Fehlerbericht möglichst durch einen (Hochschul-)Lehrer.
- Die Weitergabe des Paketes ist nicht erlaubt. Jeder Nutzer ist verpflichtet, sich die Software vom **FTP-Intranet-Server des Institutes für Mathematik und Informatik** der Fachhochschule Bochum oder von den Seiten des Autors auf dem Web-Server der Fachhochschule Bochum (erreichbar über **www.eikelberg.de**) herunterzuladen oder als Bestandteil der DVD des Autors zur Programmierung in Java zu erwerben.

2.3.4.3 Inhalt des Paketes

Derzeit (Release 3.01, September 2006) besteht das Paket de.eikelberg.iotools aus 12 sichtbaren Klassen und stellt 117 sichtbare Methoden bereit. Eher für systemnahe Programmierung gedacht sind folgende Klassen:

- **de.eikelberg.iotools.CodePageAdapter** - zentrale Methoden für die Umsetzung von Codierungen
- **de.eikelberg.iotools.CodePage850Adapter** - Umsetzung der Codierung Unicode ↔ Code Page 850 (Konsolbox von Windows)
- **de.eikelberg.iotools.CodePage850Reader** - grundlegende Methoden zur Eingabe mit Code Page 850 Verschlüsselung
- **de.eikelberg.iotools.CodePage850Writer** - grundlegende Methoden zur Ausgabe mit Code Page 850 Entschlüsselung
- **de.eikelberg.iotools.CodePage1252Adapter**- Umsetzung der Codierung Unicode ↔ Code Page 1252 (Windows GUI-Anwendungen und Windows Dateien)
- **de.eikelberg.iotools.CodePage1252Reader** - grundlegende Methoden zur Eingabe mit Code Page 1252 Verschlüsselung
- **de.eikelberg.iotools.CodePage1252Writer** - grundlegende Methoden zur Ausgabe mit Code Page 1252 Entschlüsselung

Die obigen Klassen sind für den Programmieranfänger nicht interessant und werden für die Ein- und Ausgabe auch nicht direkt sichtbar verwendet. Allerdings bilden diese Klassen die Basis und Voraussetzung für die übrigen Klassen des Paketes. Diese haben folgende Namen und Aufgaben:

- **de.eikelberg.iotools.KeyboardIn** - Tastatureingabe in der Konsolbox
- **de.eikelberg.iotools.ScreenOut** - Bildschirmausgabe in der Konsolbox
- **de.eikelberg.iotools.HoldScreen** - Ausgabe einer Programmende-Meldung und Anhalten der Bildschirmausgabe in der Konsolbox
- **de.eikelberg.iotools.DatasetIn** - Einlesen aus einer (sequentiellen) Eingabedatei
- **de.eikelberg.iotools.DatasetOut** - Ausgabe in eine (sequentielle) Datei

Eine detaillierte Erklärung der Klassen und Methoden finden Sie in der mit Hilfe des Dokumentationstools `javadoc` erstellten **Dokumentation zum Paket de.eikelberg.iotools**. Im Rahmen dieses Kurses erfolgt die Erklärung der einzelnen Methoden jeweils im Zusammenhang mit den Datentypen, auf die die Methoden angewendet werden können. Die Ein- und Ausgabe von elementaren Datentypen erfolgt also im anschließenden Kapitel 2.3.5. Die Ein- und Ausgabe von Referenzdatentypen erfolgt in Kapitel 2.6.4 im Anschluß an die Behandlung dieser Datentypen. Testprogramme zu den Methoden in de.eikelberg.iotools finden Sie im Paket bsp02006.

2.3.5 Ein- und Ausgabe elementarer Datentypen mit den Methoden des Paketes de.eikelberg.iotools

Die folgenden Tabellen geben einen Überblick über die Methoden des Paketes de.eikelberg.iotools. Eine detailliertere Erklärung finden Sie in der **Dokumentation zum Paket de.eikelberg.iotools**. Beachten Sie bitte ferner, dass alle Klassen des Paketes *gekapselt* sind, ein direkter Zugriff auf die Variablen der Klassen also nicht möglich ist. Daher werden in den Beschreibungen auch nur Methoden aufgeführt.

Zu den Tabellen dieses Kapitels beachten Sie bitte folgende Punkte:

- Mit Tastatureingabe und Bildschirmausgabe ist stets eine Ein- und Ausgabe über die Konsolbox gemeint.
- Sehr viele Methoden geben über eine interne Variable der Objekte eine Rückmeldung über Gelingen oder Fehler. Der Wert der Variablen - auch **Status**-Wert genannt - kann über eine entsprechende Methode abgefragt werden.
- In den Tabellen gibt es Methoden, die mit gleichem Namen und gleicher Funktion in mehr als nur einer Klasse vorhanden sind.
- Es gibt weitere Methoden, die in Kapitel 2.6.4 erklärt werden.

Alle Beispiele dieses Kapitels sind Methoden der Beispielprogramme bsp02021.UserIO und bsp02022.DatasetIO.

2.3.5.1 Grundlegende Methoden des Paketes de.eikelberg.iotools für die Ein- und Ausgabe in der Konsolbox

Beachten Sie bitte, dass alle genannten Methoden für die Tastatureingabe und die Bildschirmausgabe statisch sind!

Grundlegend für die Ein- und Ausgabe in der Konsolbox sind folgende Methoden:

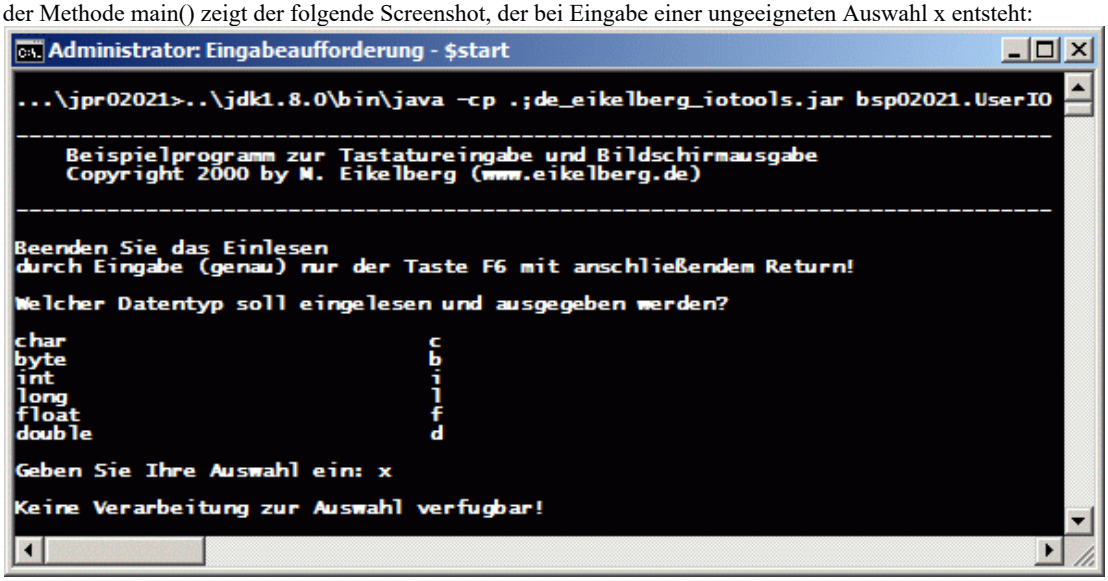
Methode(n)	Klasse	Funktion
Befehle für die Tastatureingabe		
kStartReadChar()	KeyboardIn	Ausgabe einer Meldung, wie die Tastatureingabe beendet werden kann
Befehle für die Bildschirmausgabe		
kStartScreenOut()	ScreenOut	Schreiben einer Startmeldung zum Programm
kPrint("...")	ScreenOut	Schreiben des Textes zwischen den doppelten Hochkommata
kPrintln()	ScreenOut	Wechsel in die nächste Zeile der Ausgabe
kPrintln("...")	ScreenOut	Schreiben des Textes zwischen den doppelten Hochkommata mit anschließendem Zeilenwechsel
stop()	HoldScreen	Ausgabe einer Programmende-Meldung, die mit Return bestätigt werden muss.

Ein Beispiel für die Anwendung der Methoden kStartScreenOut(), kStartReadChar(), stop() sowie der beiden Methoden kPrintln() finden Sie in der folgenden Methode main des Programms bsp02021.UserIO:

```
  public static void main(String[] args)
  {
// Variable für Status-Code nach Tastatureingabe
    int status = 0;

// Startmeldung des Programms
    ScreenOut.kStartScreenOut(
      "Beispielprogramm zur Tastatureingabe und Bildschirmausgabe",
      "M. Eikelberg (www.eikelberg.de)","2000");

// Information zur Beendigung des Programms ausgeben
    KeyboardIn.kStartReadChar();

// Menü ausgeben und Auswahl feststellen
    char auswahl=' ';
    auswahl = menu();

// Steuerung und Aufruf der Methoden
// Wiederholung der Verarbeitung solange status = 0 bleibt
    do
    {
// Verarbeitung in Abhängigkeit von der festgestellten Auswahl
      switch (auswahl)
      {
// Die Methoden teste*() geben den Status-Wert zurück und
// berichten so, ob eine ordnungsgemäße Verarbeitung erfolgte
        case 'c': status = testeChar(); break;
```

```
        case 'b': status = testeByte(); break;
        case 'i': status = testeInt(); break;
        case 'l': status = testeLong(); break;
        case 'f': status = testeFloat(); break;
        case 'd': status = testeDouble(); break;
        default:
          ScreenOut.kPrintln();
          ScreenOut.kPrintln("Keine Verarbeitung zur Auswahl verfügbar!");
          status = 4;
      }
    } while (status == 0);

// Programm-Ende Meldung ausgeben und Bildschirmausgabe anhalten
    HoldScreen.stop();
  }
```

Die Wirkung der Methode main() zeigt der folgende Screenshot, der bei Eingabe einer ungeeigneten Auswahl x entsteht:



2.3.5.2 Ein- und Ausgabe von elementaren Datentypen in der Konsolbox

Beachten Sie bitte wieder, dass alle genannten Methoden für die Tastatureingabe und die Bildschirmausgabe statisch sind!

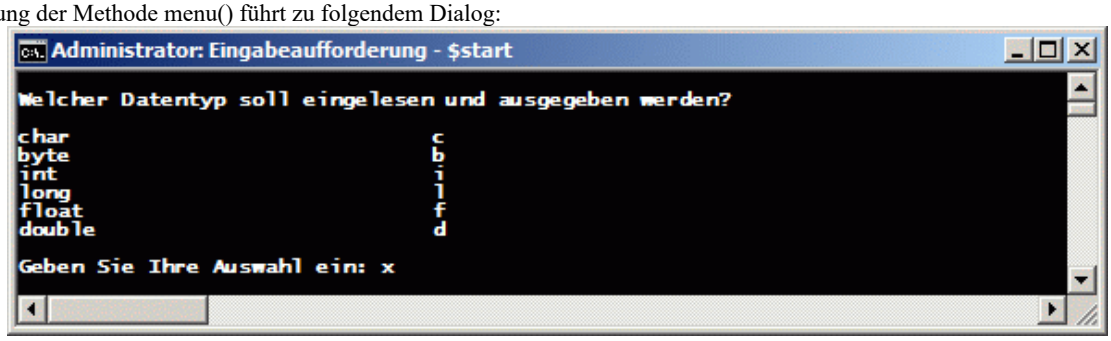
Für Verarbeitung elementarer Datentypen bei der Tastatureingabe und der Bildschirmausgabe über die Konsolbox können Sie folgende Befehle verwenden:

Methode(n)	Klasse	Funktion
Befehle für die Tastatureingabe		
kRead()	KeyboardIn	Einlesen eines Zeichens und Codierung als Dezimalzahl - Funktionalität wie bei System.in.read()
kReadChar()	KeyboardIn	Einlesen eines character Zeichens
kReadByte()	KeyboardIn	Einlesen einer byte Zahl
kReadInt()	KeyboardIn	Einlesen einer int Zahl
kReadLong()	KeyboardIn	Einlesen einer long Zahl
kReadFloat()	KeyboardIn	Einlesen einer float Zahl
kReadDouble()	KeyboardIn	Einlesen einer double Zahl
kReadLine()	KeyboardIn	Einlesen der Tastatureingabe bis 'Return' bzw. 'Enter'
kSkip()	KeyboardIn	Überlesen von Zeichen bei der Tastatureingabe
kGetStatus()	KeyboardIn	Abfrage der Status-Variablen - Fehlerinformation zum letzten Tastatureingabe
Befehle für die Bildschirmausgabe		
kPrint(...)	ScreenOut	Schreiben des Inhaltes von boolean, char, byte, int, long, double oder float Variablen
kPrintln(...)	ScreenOut	Schreiben des Inhaltes von boolean, char, byte, int, long, double oder float Variablen mit anschließendem Zeilenwechsel

Ein Beispiel für die Anwendung der Methoden kReadChar(), kReadLine(), kGetStatus() sowie der Methoden kPrint("..."), kPrintln() und kPrint("...") aus 2.3.5.1 ist die folgende Methode menu() des Programms bsp02021.UserIO:

```
  private static char menu()
  {
// Variable für Status-Code nach Tastatureingabe
    int status;
// Variable für Auswahl des Menüpunktes
    char auswahl = ' ';
// Menü auf Bildschirm ausgeben
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("Welcher Datentyp soll eingelesen und ausgegeben werden?");
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("char                                c");
    ScreenOut.kPrintln("byte                                b");
    ScreenOut.kPrintln("int                                 i");
    ScreenOut.kPrintln("long                                l");
    ScreenOut.kPrintln("float                               f");
    ScreenOut.kPrintln("double                              d");
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Geben Sie Ihre Auswahl ein: ");
// ersten eingegebenen Buchstaben einlesen
    auswahl = KeyboardIn.kReadChar();
// Status abfragen
    status = KeyboardIn.kGetStatus();
    if (status == 0)
// Rest der Eingabe incl RETURN einlesen und nicht auswerten
      KeyboardIn.kReadLine();
    else
// Fehler bei der Eingabe führt zur Rückgabe eines Leerzeichens
      auswahl = ' ';
// Auswahl an rufende Methode zurückgeben
    return auswahl;
  }
```

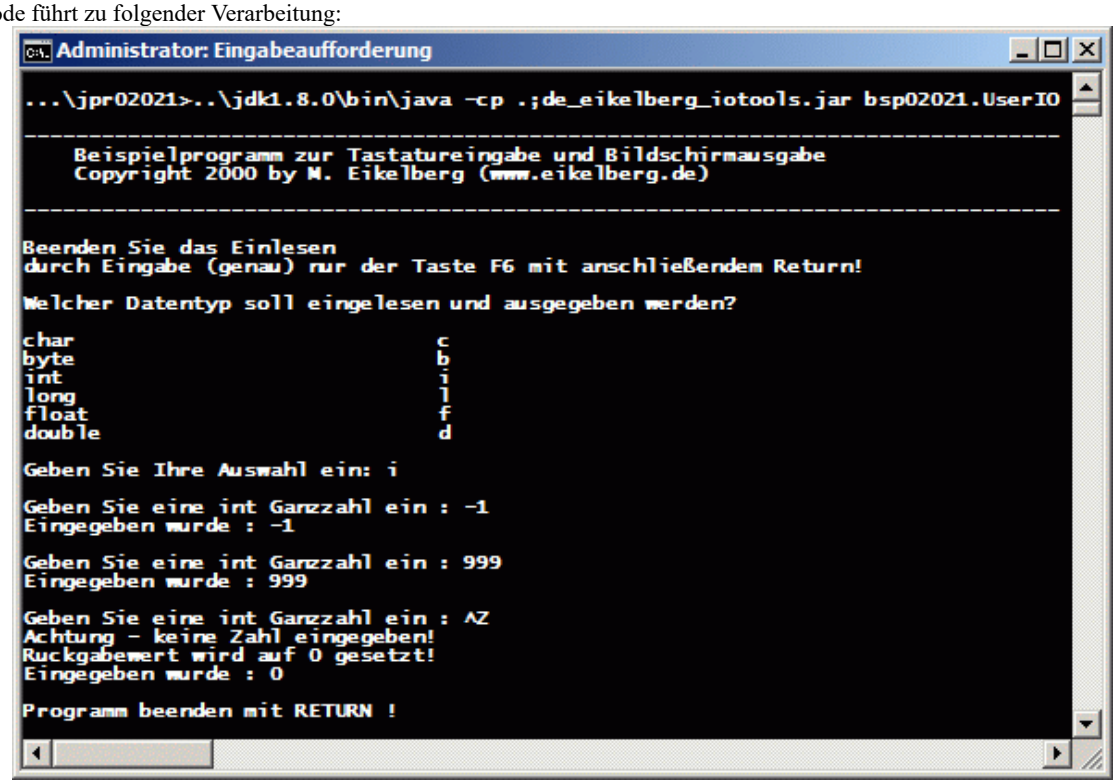
Die Ausführung der Methode menu() führt zu folgendem Dialog:



Die Methoden kReadChar(), kReadByte(), kReadInt(), kReadLong(), kReadFloat(), kReadDouble(), kSkip() sowie die Varianten der Methoden kPrint(...) und kPrintln(...) werden in den Methoden testeChar(), testeByte(), testeInt(), testeLong(), testeFloat() und testeDouble() verwendet. Die Methode testeChar() hat wie generell die Verarbeitung von Zeichen einige Besonderheiten. Die übrigen Methoden sind alle analog aufgebaut - für sie sei die Methode testeInt() als Beispiel hier angeführt:

```
// Methode für int Ganzzahlen
  private static int testeInt()
  {
// int Variable und Status-Variable deklarieren
    int x;
    int status;
// Einlesen einer integer Zahl
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Geben Sie eine int Ganzzahl ein : ");
    x = KeyboardIn.kReadInt();
// Status feststellen
    status = KeyboardIn.kGetStatus();
// Zahl wieder ausgeben
    ScreenOut.kPrint("Eingegeben wurde : ");
    ScreenOut.kPrintln(x);
// Status an die rufende Methode zurückgeben
    return status;
  }
```

Diese Methode führt zu folgender Verarbeitung:



An diesem Screenshot sieht man die Wirkung der Betätigung der Taste F6: Das Betriebssystem zeigt das Zeichen ^Z (control Z). Insbesondere kann die Methode KeyboardIn.kReadInt() keine Zahl einlesen, weshalb sie die Fehlermeldungen

```
Achtung - keine Zahl eingegeben!
Rückgabewert wird auf 0 gesetzt!
```

ausgibt. Damit eine sinnvolle weitere Verarbeitung erfolgen kann, gibt KeyboardIn.kReadInt() an Stelle einer eingelesenen Zahl eine Null zurück.

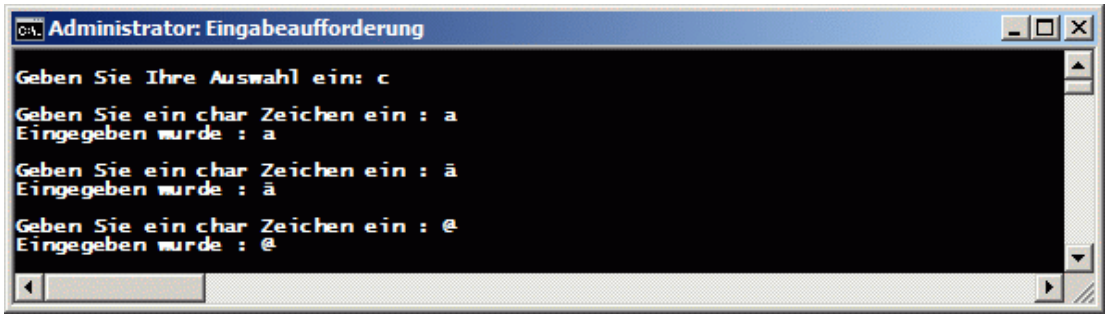
Die folgende Methode testeChar() ist wegen der Besonderheiten bei der Bearbeitung von character Zeichen anders aufgebaut als die Methoden zur Verarbeitung von Zahlen:

```
// Methode für char Zeichen
  private static int testeChar()
  {
// char Variable und Status-Variable deklarieren
    char x;
    int status;
// Einlesen eines char Zeichens
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Geben Sie ein char Zeichen ein : ");
    x = KeyboardIn.kReadChar();
// Status feststellen
    status = KeyboardIn.kGetStatus();
/* Wenn nicht F6 eingegeben wurde und die Eingabe geeignet war, ist
   status gleich 0. Es befinden sich in dem Tastaturpuffer als Folge
   der Betätigung von RETURN oder ENTER dann noch die Zeichen
   \r und \n - return und newline . Diese beiden
   Zeichen müssen in diesem Fall noch überlesen werden
*/
    if (status == 0)
    {
      KeyboardIn.kSkip(2);
// Bei status gleich 0 wird auch die Zahl wieder ausgeben
      ScreenOut.kPrint("Eingegeben wurde : ");
      ScreenOut.kPrintln(x);
    }
// Status an die rufende Methode zurückgeben
    return status;
  }
```

Bei der Verarbeitung von Zeichen entstehen Komplikationen dadurch, dass die Betätigung der ENTER-Taste bzw. der RETURN-Taste zum Einlesen der Folge der beiden Zeichen **Return** und **Newline** führt. Return wird als Zeichen mit der Codierung 13 und Newline als Zeichen

mit der Codierung 10 interpretiert. Diese beiden Zeichen werden vom Programm im Tastaturpuffer vorgefunden und müssen daher auch verarbeitet werden. Wurde dagegen F6 eingegeben, so ist der Tastaturpuffer leer und zudem der weitere Kontakt mit der Tastatur unterbunden. In diesem Fall darf also auch kein weiteres Einlesen erfolgen. Eine geeignete Verarbeitung erfolgt am leichtesten unter Verwendung der von der Methode kGetStatus() zurückgegebenen Status-Variablen. Nach Betätigung von F6 ist die Status-Variable gleich 4, im Fehlerfall gleich 8. Damit ist ein Einlesen von Return und Newline nur im Fall Status gleich Null sinnvoll. Die Situation, dass leglich die RETURN-Taste betätigt wird, ist allerdings durch diesen Programmcode nicht abgefangen!

Die Methode testeChar() führt zu folgender Verarbeitung:



Alle als Beispiele zu 2.3.5.1 und 2.3.5.2 genannten Methoden sind Bestandteil des Programms bsp02021.UserIO. Testen Sie dieses Programm ausführlich und beobachten Sie die Reaktion bei falscher Eingabe, zu kleinen oder zu großen Zahlen und Eingabe von Sonderzeichen!

2.3.5.3 Grundlegende Methoden des Paketes de.eikelberg.iotools für das Einlesen aus Dateien und die Ausgabe in Dateien

Grundlegend für das Einlesen aus einer Datei und die Ausgabe in eine Datei sind die folgenden Befehle:

Beachten Sie bitte, dass alle genannten Methoden für die Dateiverarbeitung nicht statisch sind! Vielmehr muß mit Hilfe eines Konstruktors für jede Datei ein Objekt angelegt werden, auf das beim Lese- oder Schreibvorgang Bezug genommen werden muss.

Methode(n)	Klasse	Funktion
Befehle für das Lesen aus einer Datei		
DatasetIn()	DatasetIn	Konstruktor / lesenden Zugriff auf Datei beginnen (Datei "öffnen")
dClose()	DatasetIn	lesenden Zugriff auf Datei beenden (Datei "schließen")
Befehle für das Schreiben in eine Datei		
DatasetOut()	DatasetOut	Konstruktor / schreibenden Zugriff auf Datei beginnen (Datei "öffnen")
dPrint("...")	DatasetOut	Schreiben des Textes zwischen den doppelten Hochkommata
dPrintln()	DatasetOut	Wechsel in die nächste Zeile der Ausgabe
dPrintln("...")	DatasetOut	Schreiben des Testes zwischen den doppelten Hochkommata mit anschließendem Zeilenwechsel
dClose()	DatasetOut	schreibenden Zugriff auf Datei beenden (Datei "schließen")

Für die Dateiverarbeitung fundamental ist zunächst die Bedingung, dass **für jede Datei genau ein Objekt** der Klasse DatasetIn oder DatasetOut instantiiert werden muss. Dabei ist - wie bei jeder Programmiersprache - ein Zugriff auf eine (sequentielle) Datei mit Standardmethoden ein gleichzeitiger bzw. abwechselnder lesender *und* schreibende Zugriff *nicht* möglich. Vielmehr gilt:

- Für einen lesenden Zugriff instantiiert man - beispielsweise zu einer Datei eingabe.txt - wie folgt genau ein Objekt der Klasse DatasetIn:

  ```
  DatasetIn dsIn1;
  dsIn1 = new DatasetIn("daten\\eingabe.txt");
  ```

  Mit diesen Befehlen wird der lesende Zugriff auf die Datei begonnen, man sagt auch, dass die Datei **(für input) geöffnet** wird.
- Für einen schreibenden Zugriff instantiiert man - beispielsweise zu einer Datei ausgabe.txt - wie folgt genau ein Objekt der Klasse DatasetOut:

  ```
  DatasetOut dsOut1;
  dsOut1 = new DatasetOut("daten\\ausgabe.txt");
  ```

  Mit diesen Befehlen wird der schreibende Zugriff auf die Datei begonnen, man sagt auch, dass die Datei **(für output) geöffnet** wird.

Oft stehen die Deklaration und die Instantiierung eines Objektes nicht direkt beieinander, da die Instantiierung an der jeweils von der

Verarbeitung her sinnvollen Stelle codiert werden muss, die Deklaration aber ausserhalb der Methoden der entsprechenden Klasse erfolgen muss, da in mehreren Methoden der Klasse auf das Objekt zugegriffen wird. Daher deklariert man die Objekte zu den Dateien in der Regel als private static Variable der Klasse. Der entsprechende Programmcode lautet dann wie der folgende Code im Beispielprogramm bsp02022.DatasetIO:

```
package bsp02022;

import de.eikelberg.iotools.*;

public class DatasetIO
{

/* Achtung: dsIn1 und dsOut1 müssen als private static Variable
   der gesamten Klasse deklariert werden, da auf dsIn1 und dsOut1
   in verschiedenen Methoden Bezug genommen wird!
*/
  private static DatasetIn dsIn1;
  private static DatasetOut dsOut1;

  public static void main(String[] args)
  {

  ...

    status = openDatasets("daten\\cEingabe.txt","daten\\cAusgabe.txt");
  ...

// Dateien schließen
    dsIn1.dClose();
    dsOut1.dClose();

  ....

  }

  private static long openDatasets(String einDS, String ausDS)
  {
// Statusvariable
    long status;

/* Eingabedatei öffnen durch Instantiieren eines Objektes der
   Klasse DatasetIn
   ...
*/
    dsIn1 = new DatasetIn(einDS);

// Status abfragen
    status = dsIn1.dGetStatus();

    if (status == 0)
    {
/* Ausgabedatei öffnen durch Instantiieren eines Objektes der
   Klasse DatasetOut
   ....
*/
      dsOut1 = new DatasetOut(ausDS);

// Status abfragen
      status = dsIn1.dGetStatus();
    }

// Status-Wert an das rufende Programm zurückgeben
    return status;
  }

...

}
```

Das obige Beispiel zeigt die Verwendung der **Konstruktoren DatasetIn()** und **DatasetOut()** sowie der Methoden dGetStatus. Man beachte, dass diese Methode sowohl in der Klasse DatasetIn als auch in der Klasse DatasetOut vorhanden ist. Ganz im Sinne objektorientierter Programmierung liefert dann im obigen Beispiel

- dsIn1.dGetStatus() den Status-Code zum letzten Einlesevorgang und
- dsOut1.dGetStatus() den Status-Code zum letzten Schreibvorgang.

Aufgrund der Verwendung von Objekten gibt es zu jeder Datei ein Objekt und in jedem Objekt die Statusvariable zur entsprechenden Datei. Daher sind die Statusvariablen völlig unabhängig voneinander und überschreiben sich auch nicht! Man sieht hier, dass das Objekt gemäß der Zielsetzung objektorientierter Programmierung zu einer Datei den Zugriff, die Verarbeitung sowie Informationen - also alles was mit der Datei zusammenhängt - thematisch gruppiert.

Ein wichtiges weiteres Element in obigem Programmcode bildet der Aufruf der Methoden close(), die den Zugriff auf die jeweiligen Dateien beenden. Wieder gibt es zu jeder Klasse eine Methode **close()**, die entsprechend auch für jedes instantiierte Objekt aufgerufen werden muss. Das Fehlen dieses Aufrufs führt oft nicht sofort zu einem Problem und wird auch vom Compiler nicht bemerkt! Zur Laufzeit können dann allerdings Fehler auftreten. insbesondere wenn der Aufruf von close() zu einer Ausgabedatei vergessen wird, kann im Allgemeinen ein Teil der zuletzt zu schreibenden Daten fehlen (da der Buffer zur Datei nicht von jedem Schreib-Befehl anschließend auch 'geflusht' - also entleert - wird).

2.3.5.4 Ein- und Ausgabe von elementaren Datentypen bei Dateiverarbeitung

Für Verarbeitung elementarer Datentypen können Sie folgende Befehle verwenden:

Beachten Sie bitte wieder, dass alle genannten Methoden für die Dateiverarbeitung nicht statisch sind! Beachten Sie die Notwendigkeit des Instantiierens eines Objekts zu jeder Datei.

Methode(n)	Klasse	Funktion
Befehle für das Lesen aus einer Datei		
dRead()	DatasetIn	Einlesen eines Zeichens und Codierung als Dezimalzahl
dReadChar()	DatasetIn	Einlesen eines character Zeichens
dReadByte()	DatasetIn	Einlesen einer byte Zahl
dReadInt()	DatasetIn	Einlesen einer int Zahl
dReadLong()	DatasetIn	Einlesen einer long Zahl
dReadFloat()	DatasetIn	Einlesen einer float Zahl
dReadDouble()	DatasetIn	Einlesen einer double Zahl
dReadLine()	DatasetIn	Einlesen einer ganzen Zeile der Eingabedatei
dSkipChar()	DatasetIn	Überlesen von Zeichen bei der Eingabe
dNewLine()	DatasetIn	Überlesen des Restes der aktuellen Zeile und Wechsel in die nächste Zeile der Datei
dGetStatus()	DatasetIn	Abfrage der Status-Variablen - Fehlerinformation zum letzten Einlesevorgang
Befehle für das Schreiben in eine Datei		
dPrint(...)	DatasetOut	Schreiben des Inhaltes von boolean, char, byte, int, long, double oder float Variablen
dPrintln(...)	DatasetOut	Schreiben des Inhaltes von boolean, char, byte, int, long, double oder float Variablen mit anschließendem Zeilenwechsel
dGetStatus()	DatasetOut	Abfrage der Status-Variablen - Fehlerinformation zum letzten Schreibvorgang

Wie bei der Erklärung der Schreib- und Lesemethoden zu Tastatur und Bildschirm soll auch für die Dateiverarbeitung eine Methode zur Verarbeitung von Zahlen und die Methode zur Verarbeitung von Zeichen als Beispiel angeführt werden, da die Verarbeitung von char Zeichen wegen der Verarbeitung von Zeilenwechseln eine Besonderheit aufweist.

Beispiel für die Verarbeitung von Zahlen ist die Methode testeInt() des Programms bsp02022.DatasetIO:

```
// Methode für int Ganzzahlen
  private static int testeInt()
  {
// int Variable und Status-Variable deklarieren
    int x;
    int status;
// Einlesen einer integer Zahl
    x = dsIn1.dReadInt();
// Status feststellen unter Bezug auf das Objekt dsIn1
    status = dsIn1.dGetStatus();
// Bei Erfolg Ausgabe
    if (status == 0)
    {
```

```
// Zahl wieder ausgeben - jede Zahl in einer neuen Zeile
      dsOut1.dPrintln(x);
// Status feststellen - jetzt unter Bezug auf das Objekt dsOut1 !!
      status = dsOut1.dGetStatus();
    }
// Status an die rufende Methode zurückgeben
    return status;
  }
```

In der Methode testeInt() wird

- mit dsIn1.dReadInt() eine integer Zahl gelesen und anschließend
- mit dsInt1.dGetStatus() der hierdurch entstandene Statuscode aus dem Objekt dsIn1 gelesen.

Die Methode dReadInt() überliest Leerzeichen und Zeilenwechsel in der Eingabedatei bis zum nächsten Zeichen. Es liest die darauffolgenden Zeichen bis zum nächsten Leerzeichen oder Zeilenwechsel und wandelt die so gefundene Zeichenkette in eine int Zahl um sofern dies möglich ist. Andernfalls wird die Zahl 0 als gelesen zurückgegeben und ein von der jeweiligen Fehlersituation abhängiger von Null verschiedener Statuscode gesetzt.

Ist beim Einlesen kein Fehler aufgetreten (status gleich 0), so wird

- mit dsOut1.dPrintln(x) die gelesene Zahl mit anschließendem Zeilenwechsel in die Ausgabedatei geschrieben und
- mit dsOut1.dGetStatus(); aus dem Objekt zur dsOut1 zur Ausgabedatei der Statuscode gelesen.

Aufgrund der anderen Verarbeitung des Zeilenwechsels unterscheidet sich die Methode testeChar() von der Methode testeInt(). Die Methode dReadInt() überliest wie oben bereits erklärt Leerzeichen und Zeilenwechsel. Die in testeChar() aufgerufene Methode dReadChar() liest jedes Zeichen einzeln aus der Datei und liefert damit auch Leerzeichen sowie bei Zeilenwechsel die beiden Zeichen Return (\r - Zeichen 13) und Newline (\n - Zeichen 10) zurück. Daher ist testeChar() so codiert, dass alle Zeichen eins-zu-eins aus der Eingabedatei gelesen und in die Ausgabedatei geschrieben werden:

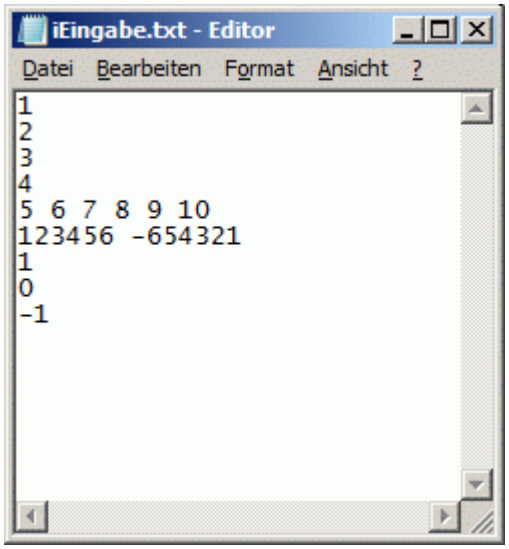
```
// Methode für char Zeichen
  private static int testeChar()
  {
// char Variable und Status-Variable deklarieren
    char x;
    int status;
// Einlesen eines char Zeichens
    x = dsIn1.dReadChar();
// Status feststellen unter Bezug auf das Objekt dsIn1
    status = dsIn1.dGetStatus();
// Bei Erfolg Ausgabe
    if (status == 0)
    {
/* Zahl wieder ausgeben - jedes Zeichen hintereinander
   Zeilenwechsel entstehen aus den Zeichen Return und Newline
   in der Eingabedatei
*/
      dsOut1.dPrint(x);
// Status feststellen - jetzt unter Bezug auf das Objekt dsOut1 !!
      status = dsOut1.dGetStatus();
    }
// Status an die rufende Methode zurückgeben
    return status;
  }
```

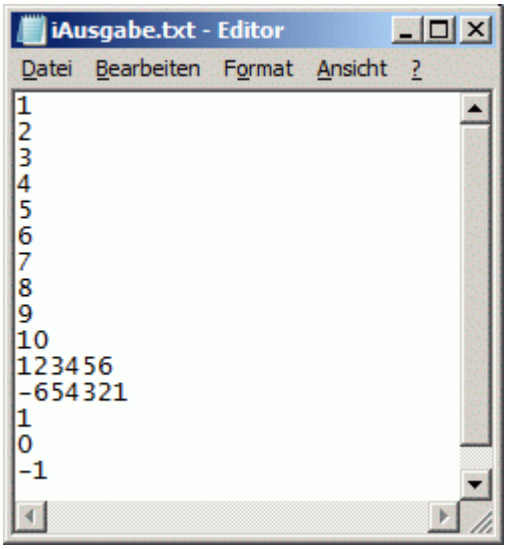
Vergleicht man die Methoden testeInt() und testeChar(), so ist der wesentliche Unterschied, dass

- testeInt() jeweils mit anschließendem Zeilenwechsel in die Ausgabedatei schreibt und dass
- testeChar() niemals mit anschließendem Zeilenwechsel in die Ausgabedatei schreibt.

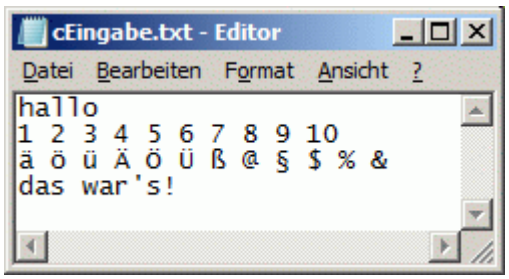
Die Zeilenwechsel in der Ausgabedatei iAusgabe.txt zum Test mit der Methode testeInt() entstehen daher alle aus der Verarbeitung des Programms. Die Zeilenwechsel in der Ausgabedatei cAusgabe.txt zum Test mit der Methode testeChar() entstehen dagegen genau nicht aus der Verarbeitung sondern sämtlich aus den Zeilenwechseln der zugehörigen Eingabedatei cEingabe.txt.

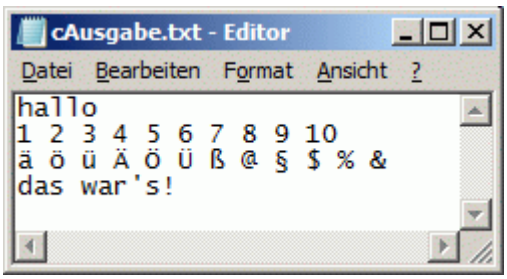
Zur Verdeutlichung stellen wird die Ein- und Ausgabedateien zu den beiden Tests im Folgenden zusammen. Bei dem Test mit der Methode testeInt() wird aus der Datei daten\iEingabe.txt gelesen und in die Datei daten\iAusgabe.txt geschrieben:





Man sieht, dass aufgrund der Verarbeitung in der Ausgabedatei jede Zahl in einer eigenen Zeile steht. Bei dem Test mit der Methode testeChar() wird aus der Datei daten\cEingabe.txt gelesen und in die Datei daten\cAusgabe.txt geschrieben:





Man sieht, dass aufgrund der Verarbeitung bei der Verwendung von kReadChar() die Zeilenwechsel in der Ausgabedatei genau den Zeilenwechseln der Eingabedatei entsprechen.

2.3.5.5 Verarbeitung deutscher Sonderzeichen

> **Hinweis:** In der in Eclipse integrierten Konsolbox erfolgt eine Ein- und Ausgabe von Schriftzeichen unter Verwendung von Code Page 1252 (!). Dies entspricht nicht dem Verhalten des Programms bei Start in einer Konsolbox. Bei einem Start eines Java-Programms über Eclipse erfolgt somit die Verarbeitung der Ein- und Ausgabe deutscher Sonderzeichen in der Konsolbox fehlerhaft codiert!

Mit dem Beispiel jpr02103 kann die Verarbeitung deutscher Sonderzeichen durch die Klassen des Paketes de.eikelberg.iotools getestet werden. Das Beispielprogramm bsp02103.SZeichenTest führt folgende drei Tests durch:

1. Einlesen in Konsolbox und Ausgabe in Windows-Fenster und Datei
2. Einlesen aus Windows-Fenster und Ausgabe in Konsolbox und Datei
3. Einlesen aus einer Datei und Ausgabe in Windows-Fenster und Konsolbox

Für den Test der Ein- und Ausgabe in einem Windows-Fenster werden dabei die Klassen JFrame und JOptionPane des Paketes javax.swing verwendet (vgl. Kapitel 3.8.2);

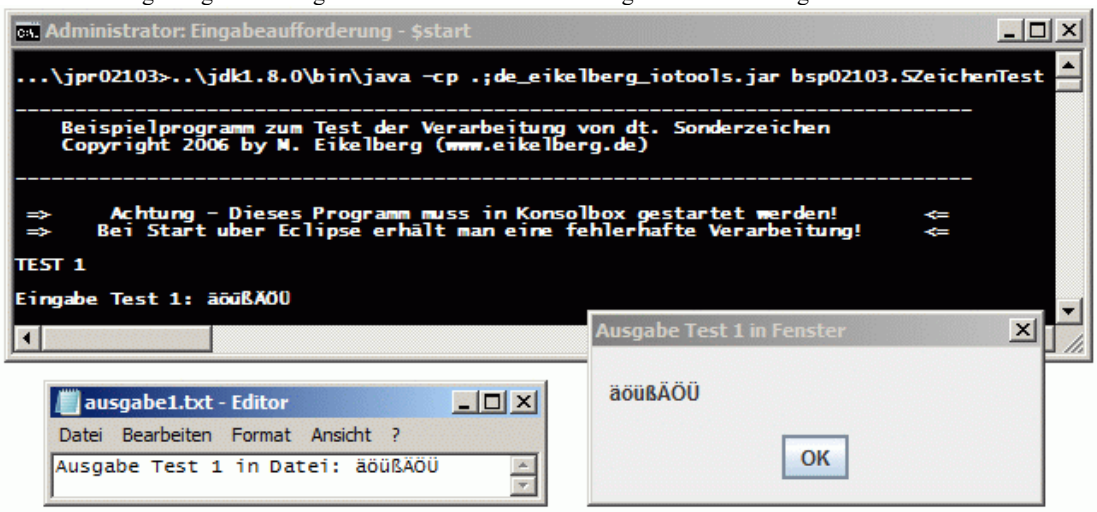
Test 1: Einlesen in Konsolbox und Ausgabe in Windows-Fenster und Datei

Der Programmcode für die Durchführung des ersten Tests lautet:

```
// 1. Test: Einlesen in Konsolbox und Ausgabe in Windows-Fenster und Datei
    ScreenOut.kPrintln("\nTEST 1");
// 1.1 Einlesen von der Tastatur
    ScreenOut.kPrint("\nEingabe Test 1: ");
    String test = KeyboardIn.kReadLine();
// 1.2 Ausgabe in eine Datei
    DatasetOut Aus1 = new DatasetOut("daten\\ausgabe1.txt");
    Aus1.dPrint("Ausgabe Test 1 in Datei: "+test);
    Aus1.dClose();
// 1.3 Ausgabe in ein Windows-Fenster
    JFrame jf = new JFrame();
```

```
    JOptionPane.showMessageDialog
    (
       jf,test,
       "Ausgabe Test 1 in Fenster" ,JOptionPane.PLAIN_MESSAGE
    );
```

Der Ablauf dieser Anweisungen ergibt bei Eingabe des Textes "äöüßÄÖÜ" folgende Verarbeitung:



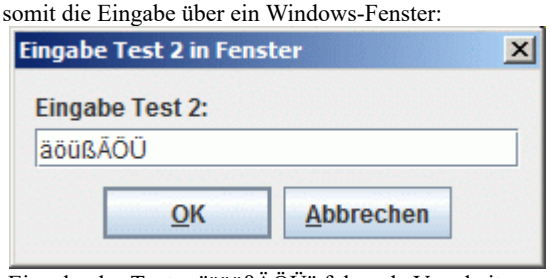
Die Sonderzeichen werden somit korrekt in der Konsole eingelesen und in ein Windows-Fenster sowie in eine Datei ausgegeben.

Test 2: Einlesen aus Windows-Fenster und Ausgabe in Konsolbox und Datei

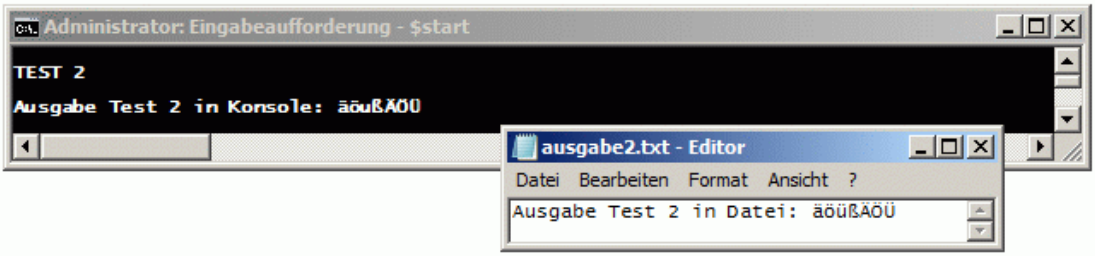
Der Programmcode für die Durchführung des zweiten Tests lautet:

```
// 2. Test: Einlesen aus Windows-Fenster und Ausgabe in Konsolbox und Datei
    ScreenOut.kPrintln("\nTEST 2");
// 2.1 Einlesen aus Fenster
    test = JOptionPane.showInputDialog
    (
       jf,"Eingabe Test 2:",
       "Eingabe Test 2 in Fenster",JOptionPane.PLAIN_MESSAGE
    );
// 2.2 Ausgabe in eine Datei
    DatasetOut Aus2 = new DatasetOut("daten\\ausgabe2.txt");
    Aus2.dPrint("Ausgabe Test 2 in Datei: "+test);
    Aus2.dClose();
// 2.3 Ausgabe in die Konsolbox
    ScreenOut.kPrintln("\nAusgabe Test 2 in Konsole: "+test);
```

Bei der Durchführung dieses Tests erfolgt somit die Eingabe über ein Windows-Fenster:



Der Ablauf dieser Anweisungen ergibt bei Eingabe des Textes "äöüßÄÖÜ" folgende Verarbeitung:



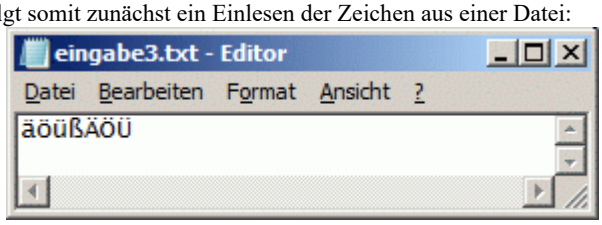
Man sieht, dass in der Konsole statt eines kleinen 'ü' ein kleines 'u' ausgegeben wird. Diese Verarbeitung wurde so implementiert, weil es ab Java 1.2 grundsätzlich unmöglich ist, ein kleines 'ü' von der JVM in der Konsole auszugeben. Ansonsten werden die Sonderzeichen sowohl in der Konsole als auch in die Datei korrekt ausgegeben.

Test 3: Einlesen aus einer Datei und Ausgabe in Windows-Fenster und Konsolbox

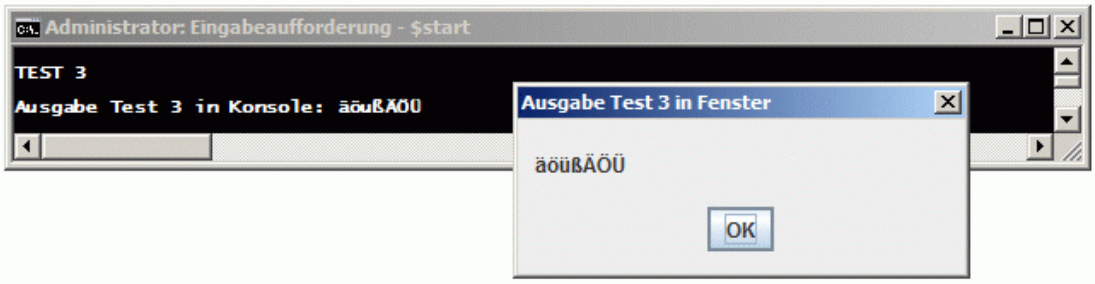
Der Programmcode für die Durchführung des dritten Tests lautet:

```
// 3. Test: Einlesen aus Datei und Ausgabe in Windows-Fenster und Konsolbox
    ScreenOut.kPrintln("\nTEST 3");
// 3.1 Einlesen aus Datei
    DatasetIn Ein3 = new DatasetIn("daten\\eingabe3.txt");
    test = Ein3.dReadLine();
    Ein3.dClose();
// 3.2 Ausgabe in die Konsolbox
    ScreenOut.kPrintln("\nAusgabe Test 3 in Konsole: "+test);
// 3.3 Ausgabe in ein Windows-Fenster
    JOptionPane.showMessageDialog
    (
       jf,test,
       "Ausgabe Test 3 in Fenster" ,JOptionPane.PLAIN_MESSAGE
    );
```

Bei der Durchführung dieses Tests erfolgt somit zunächst ein Einlesen der Zeichen aus einer Datei:



Wieder wird in der Konsolbox statt eines kleinen 'ü' ein kleines 'u' ausgegeben. Ansonsten erfolgt sowohl in der Konsolbox als auch im Windows-Fenster eine korrekte Anzeige.



2.4 Operatoren

2.4.1 Typen von Operatoren

In Kapitel 2.2.4.2 wurde der **Ausdruck** als kleinste ausführbare Einheit eines Java-Programms eingeführt. Ausdrücke dienen zur Berechnung von numerischen Werten, logischen Ausdrücken sowie zur Veränderung des Wertes von Variablen. Ein Ausdruck besteht aus mindestens einem Operanden und mindestens einer Variablen. Oft bezeichnet man auch Java Quellcode rechts neben einem Zuweisungsoperator (s.u.) als Ausdruck.

Ein erster Operatoren-Typ ist bereits in Kapitel 2.3.2 erklärt worden. Ein **Operator** ist die symbolhafte Darstellung eines Vorgangs, der mit den zugeordneten Variablen durchgeführt werden soll. Die zugeordneten Variablen heißen in diesem Fall **Operanden**. Die Zahl der Operanden nennt man **Stelligkeit** des Operators, Operatoren mit einem Operand heißen **einstellig**, Operatoren mit zwei Operanden heißen **zweistellig** und Operatoren mit drei Operanden heißen **dreistellig**.

Erste Beispiele kennen wir bereits aus Kapitel 2.3.2 - die arithmetischen Operatoren. So ist zum Beispiel

```
x1 + x2
```

ein Ausdruck, der aus dem zweistelligen Operator + und den beiden zugehörigen Operanden `x1` und `x2` besteht.

Ingesamt kennt Java folgende Typen von Operatoren:

- **arithmetische** Operatoren
- **relationale** Operatoren (**Vergleichsoperatoren**)
- **logische** Operatoren
- **Zuweisungsoperatoren**
- **bitweise** Operatoren
- Operatoren mit spezieller Funktionalität

Die arithmetischen Operatoren wurden bereits in **Kapitel 2.3.2** erklärt. Dort finden Sie auch eine Übersichtstabelle zu den arithmetischen Operatoren. Relationale Operatoren sind Thema von Kapitel 2.4.2. Die logischen Operatoren werden in Kapitel 2.4.3 behandelt. Kapitel 2.4.4 thematisiert die Zuweisungsoperatoren. Gegenstand von Kapitel 2.4.5 sind die bitweisen Operatoren und die Operatoren mit spezieller Funktionalität. Die Reihenfolge der Auswertung der Operatoren in Ausdrücken ist Thema von Kapitel 2.4.6.

2.4.2 Relationale Operatoren

Sehr häufig ist es in Programmen erforderlich, numerische Werte miteinander zu vergleichen. Hierzu gibt es die folgenden sechs **relationalen Operatoren**, die als Rückgabewert die logischen Werte `true` und `false` ergeben:

Operator	Bezeichnung	Wirkung	Anwendung	Bindungspriorität	Assoziativität
`==`	Gleich	Vergleich zweier Operatoren auf Gleichheit	`a == b`	8	links
`!=`	Ungleich	Vergleich zweier Operatoren auf Ungleichheit	`a != b`	8	links
`<`	Kleiner	Vergleich zweier Operatoren - kleiner	`a < b`	7	links
`<=`	Kleiner gleich	Vergleich zweier Operatoren - kleiner gleich	`a <= b`	7	links
`>`	Größer	Vergleich zweier Operatoren - größer	`a > b`	7	links
`>=`	Größer gleich	Vergleich zweier Operatoren - größer gleich	`a >= b`	7	links

Die Bedeutung der Einträge zur Bindungspriorität und Assoziativität werden in Kapitel 2.4.6 erklärt. Diese Einträge sind für die logische Gruppierung und die Auswertungsreihenfolge von Operatoren von Bedeutung.

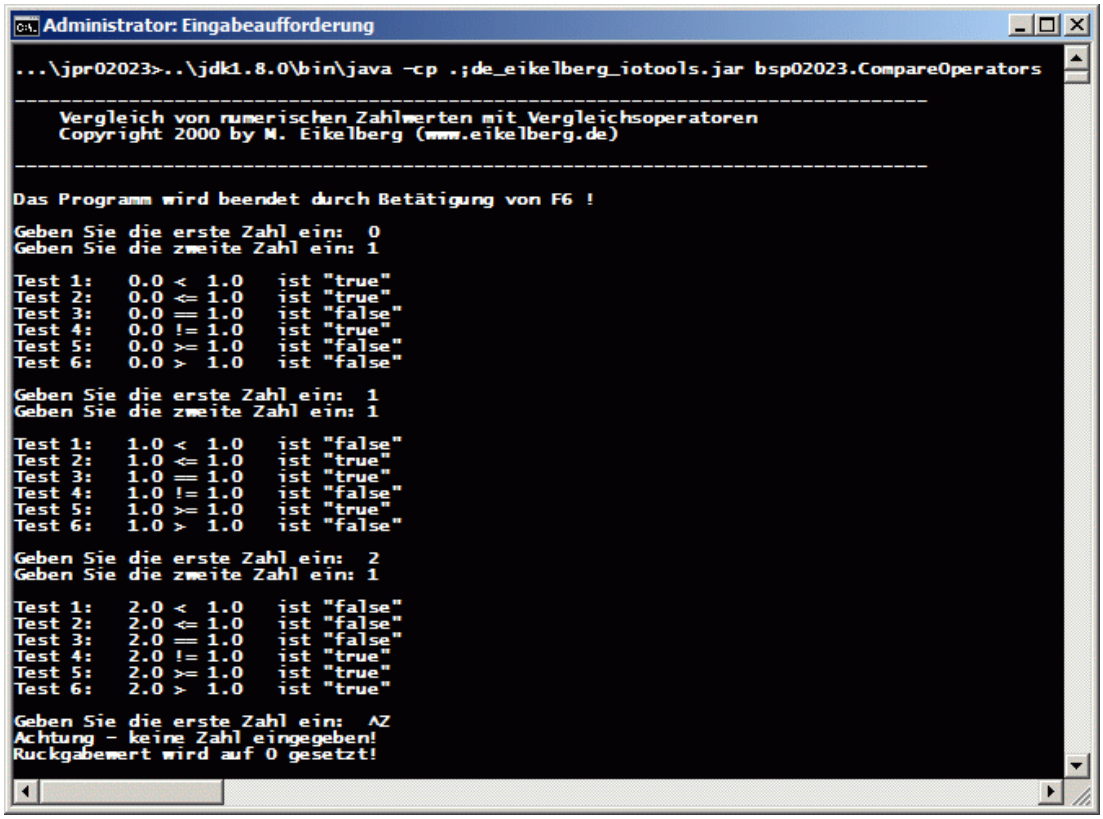
Bei der Verwendung der relationalen Operatoren ist es nicht notwendig, dass beide Operanden vom gleichen numerischen Datentyp sind.

Die Nutzung von relationalen Operationen ist im Prinzip verhältnismäßig einfach, wie das folgende Beispielprogramm bsp02023.CompareOperators zeigt:

```
package bsp02023;

import de.eikelberg.iotools.*;

public class CompareOperators
{

  public static void main(String[] args)
  {
// Startmeldung
    ScreenOut.kStartScreenOut(
      "Vergleich von numerischen Zahlwerten mit Vergleichsoperatoren",
      "M. Eikelberg (www.eikelberg.de)","2000");

// Beenden des Programms bei Einlesen einer Null
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("Das Programm wird beendet durch Betätigung von F6 !");

// Deklaration der beiden zu vergleichenden Zahlen
    float x1,x2;

// Status-Variable für die Eingabe
    int status = 0;

// Schleife mit Verarbeitung
    do
    {
// Einlesen der ersten Zahl
      ScreenOut.kPrintln();
      ScreenOut.kPrint("Geben Sie die erste Zahl ein:  ");
      x1 = KeyboardIn.kReadFloat();
      status = KeyboardIn.kGetStatus();
// Verarbeitung, wenn status gleich 0 ist
      if (status == 0)
      {
// Einlesen der zweiten Zahl
        ScreenOut.kPrint("Geben Sie die zweite Zahl ein: ");
        x2 = KeyboardIn.kReadFloat();
        status = KeyboardIn.kGetStatus();
// Vergleich durchführen, wenn status wieder gleich 0 ist
        if (status == 0)
        {
// Vergleich der eingegebenen Zahlen
```

```
          boolean b1,b2,b3,b4,b5,b6;
          b1 = x1 <  x2;    // Test 1
          b2 = x1 <= x2;    // Test 2
          b3 = x1 == x2;    // Test 3
          b4 = x1 != x2;    // Test 4
          b5 = x1 >= x2;    // Test 5
          b6 = x1 >  x2;    // Test 6
// Ausgabe
          ScreenOut.kPrintln();
          ScreenOut.kPrintln
            ("Test 1:   "+x1+" <  "+x2+"   ist "+'"'+b1+'"');
          ScreenOut.kPrintln
            ("Test 2:   "+x1+" <= "+x2+"   ist "+'"'+b2+'"');
          ScreenOut.kPrintln
            ("Test 3:   "+x1+" == "+x2+"   ist "+'"'+b3+'"');
          ScreenOut.kPrintln
            ("Test 4:   "+x1+" != "+x2+"   ist "+'"'+b4+'"');
          ScreenOut.kPrintln
            ("Test 5:   "+x1+" >= "+x2+"   ist "+'"'+b5+'"');
          ScreenOut.kPrintln
            ("Test 6:   "+x1+" >  "+x2+"   ist "+'"'+b6+'"');
// Blöcke schliessen
        }
      }
// Schleife beenden
    } while (status == 0);
  }
}
```

Einen Programmlauf von bsp02023.CompareOperators zeigt der folgende Screenshot:



Man beachte allerdings, dass bei der Verwendung von Gleitpunktzahlen die bereits in Kapitel 2.3.2.5 angesprochenen Besonderheiten der Gleitpunktarithmetik in Bezug auf die Rechengenauigkeit beachtet werden müssen:

Bei der Verwendung von Gleitkommazahlen kann es aufgrund des Genauigkeitsverlustes zu unerwartet auftretender Ungleichheit von Zahlen oder allgemeiner zu unerwarteten Ergebnissen bei einem Vergleich kommen. Durch den Genauigkeitsverlust kann nämlich bei einem Vergleich ein anderes Ergebnis entstehen als es mathematisch und logisch zu erwarten wäre. Insbesondere bei der Steuerung von Schleifen (vgl. Kap. 2.5.3) kann es hier zu unerwartetem Verhalten zur Laufzeit kommen.

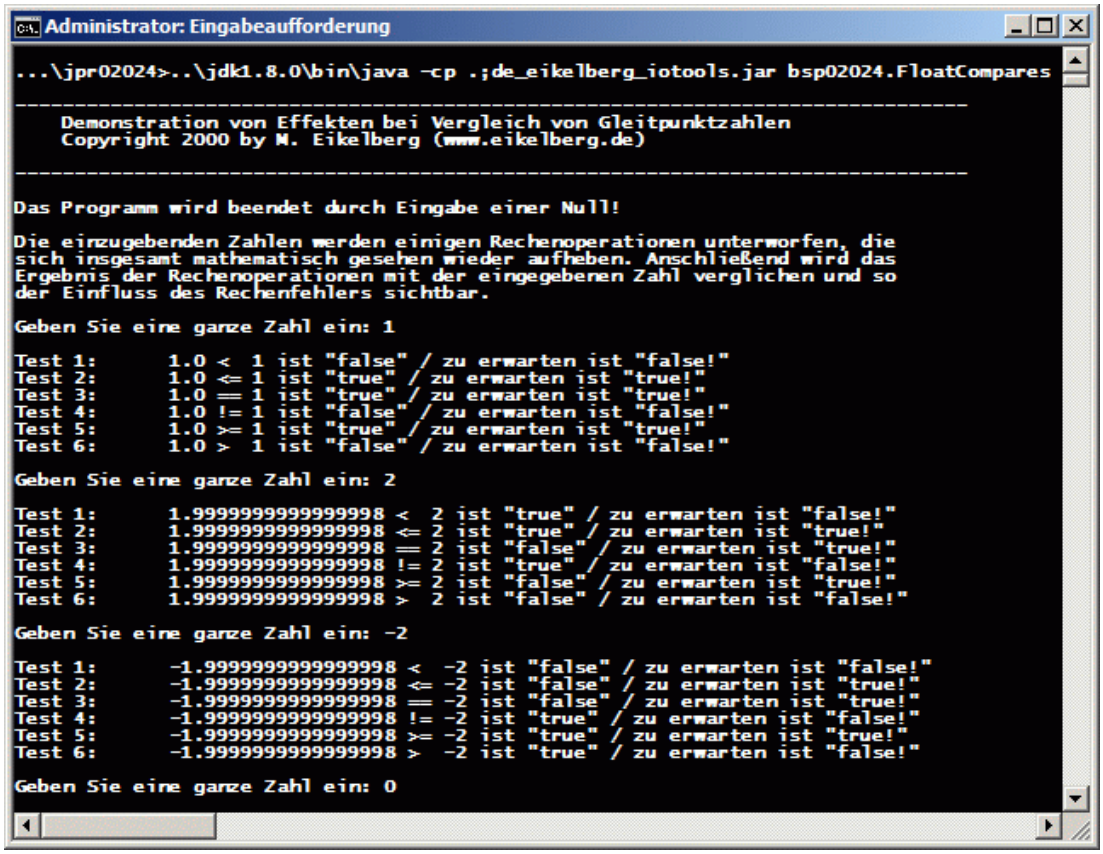
Ein Beispiel für dieses Phänomen ist das Beispielprogramm bsp02024.FloatCompares. Dieses Programm liest eine ganze Zahl ein und speichert sie auf einer Ganzzahlvariablen mit Namen `faktor` ab. Anschließend wird folgende Verarbeitung vorgenommen, die - mathematisch gesehen - insgesamt keine Auswirkung auf die eingegebenen Zahl haben sollte:

```
// Anwendung von Rechenoperationen auf die Zahl faktor
        double f;
        f = (Math.PI/4) / faktor;          // Math.Pi ist die Zahl pi=3.14....
        f = 2*Math.sin(f)*Math.cos(f);     // Sinus von 2*f berechnen:
                                           //   sin(2*f)=2*sin(f)*cos(f)
        f = Math.asin(f)/2;                // Arcus Sinus berechnen
                                           //   und durch 2 teilen
        f = (Math.PI/4) / f;               // 1. Rechenschritt rückgängig machen
```

Nach der Verarbeitung wird das Rechenergebnis in der Variablen `f` mit der ursprünglich eingegebenen Zahl in der Variablen `faktor` wie folgt verglichen:

```
// Vergleich von eingegebener Zahl und Rechenergebnis
        boolean b1,b2,b3,b4,b5,b6;
        b1 = f <  faktor;    // Test 1
        b2 = f <= faktor;    // Test 2
        b3 = f == faktor;    // Test 3
        b4 = f != faktor;    // Test 4
        b5 = f >= faktor;    // Test 5
        b6 = f >  faktor;    // Test 6
```

Zu jedem dieser Tests gibt das Programm aus, welcher logische Wert ohne Genauigkeitsverlust zu erwarten gewesen wäre, was also das mathematisch und logisch zu erwartende Ergebnis ist. Lässt man das Programm bsp02024.FloatCompares laufen, so erhält man folgende Bildschirmausgabe:



Man sieht, dass es sowohl ein richtiges Ergebnis als auch Abweichungen nach oben und unten geben kann. Wenn Abweichungen aufgetreten sind, gibt es zu jedem Test *jeweils mindestens ein erwartetes und ein unerwartetes Ergebnis* gibt.

2.4.3 Logische Operatoren

Logische Operatoren werden auf ein oder zwei logische Operanden angewendet. Insgesamt gibt es die folgenden sechs logischen Operatoren:

Operator	Bezeichnung	Wirkung	Anwendung	Bindungspriorität	Assoziativität
`!`	logisches NICHT - Negation/Komplement (NOT)	Umkehrung des logischen Wertes, Verneinung	`!a`	2	rechts
`&`	logisches UND (AND)	ergibt wahr, wenn beide Operanden wahr sind	`a & b`	9	links
`&&`	logisches UND (AND)	ergibt wahr, wenn beide Operanden wahr sind (Short-Circuit)	`a && b`	12	links
`|`	logisches nicht exklusives ODER (OR)	ergibt wahr, wenn mindestens einer der beiden Operanden wahr ist	`a | b`	11	links
`||`	logisches nicht exklusives ODER (OR)	ergibt wahr, wenn mindestens einer der beiden Operanden wahr ist (Short-Circuit)	`a || b`	13	links
`^`	logisches exklusives ODER (XOR)	ergibt wahr, wenn genau ein Operand wahr und genau ein Operand falsch ist	`a ^ b`	10	links

Die Verwendung logischer Operatoren ist relativ einfach, allerdings sollte man sich vor der Codierung - zum Beispiel mit Hilfe von Struktogrammen - genaue Gedanken über die zu implementierende (also die in Programmcode umzusetzende) Logik machen. Die Implementierung einer falschen Logik führt in der Regel zu Laufzeitfehlern, deren Suche unter Umständen sehr aufwendig sein kann.

Ein Beispiel für die Verwendung der logischen Operatoren ohne Short-Circuit-Evaluation ist das Programm bsp02025.BooleanOps:

```
Administrator: Eingabeaufforderung
...\jpr02025>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02025.BooleanOps
--------------------------------------------------------------------------------
    Beispielprogramm zu logischen Operatoren
    Copyright 2000 by W. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Das Programm wird beendet durch eine falsche Eingabe
oder durch Eingabe von (genau nur) F6!

Das Programm berechnet folgende Ausdrucke:
  Test 1:   b11 = !b1;
  Test 2:   b12 = b1 | b2;
  Test 3:   b13 = b1 & b2;
  Test 4:   b14 = b1 ^ b2;

Geben Sie einen Wahrheitswert fur b1 ein [w fur true, f fur false] : w
Geben Sie einen Wahrheitswert fur b2 ein [w fur true, f fur false] : w

Ergebnisse:
  Test 1:   false = !true
  Test 2:   true = true | true
  Test 3:   true = true & true
  Test 4:   false = true ^ true

Geben Sie einen Wahrheitswert fur b1 ein [w fur true, f fur false] : f
Geben Sie einen Wahrheitswert fur b2 ein [w fur true, f fur false] : w

Ergebnisse:
  Test 1:   true = !false
  Test 2:   true = false | true
  Test 3:   false = false & true
  Test 4:   true = false ^ true

Geben Sie einen Wahrheitswert fur b1 ein [w fur true, f fur false] : x
```

Das Beispiel zeigt, dass der Ausdruck `a^b` genau dann "wahr" ist, wenn genau ein Operand "wahr" und genau ein Operand "falsch" ist. Sind beide Operanden wahr, so ist das Ergebnis von `a^b` dagegen gleich "falsch"!

Wir betrachten jetzt die beiden logischen Operatoren mit Short-Circuit-Evaluation. Vom Ergebnis her gibt es zwischen den Operatoren ohne und mit Short-Circuit-Evaluation, also zwischen `&` und `&&` bzw. `|` und `||` keine Unterschiede. Verwendet man die beiden Operatoren `&&` und `||`, so erfolgt bei der Auswertung eine sogenannte **Short-Circuit-Evaluation**: Short-Circuit-Evaluation bedeutet, dass der zweite Operand des Gesamtausdrucks nur dann ausgewertet wird, wenn dies für die Berechung des Gesamtergebnisses erforderlich ist. Bei `a&&b` wird b also nur dann ausgewertet, wenn a wahr ist. Entsprechend wird bei `a||b` der Ausdruck b nur dann ausgewertet, wenn a falsch ist. Dies kann Auswirkungen auf beteiligte Variablen haben wie das Beispielprogramm bsp02026.ShortCircuit zeigt:

```
Administrator: Eingabeaufforderung
...\jpr02026>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02026.ShortCircuit
--------------------------------------------------------------------------------
    Beispielprogramm zur Short-Circuit-Evaluation bei log. Operatoren
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------

  Das Programm berechnet folgende Ausdrucke:
  Test 1:   b1 = true  |  (x1 == x2++);
  Test 2:   b1 = true  || (x1 == x2++);
  Test 3:   b1 = false &  (x1 == x2++);
  Test 4:   b1 = false && (x1 == x2++);

Beenden Sie das Einlesen
durch Eingabe (genau) nur der Taste F6 mit anschließendem Return!

Geben Sie eine Zahl zwischen 0 und 9 fur x1 ein : 2
Geben Sie eine Zahl zwischen 0 und 9 fur x2 ein : 1

  Ergebnisse:  x1 | x2 | b1
  --------------+----+-------
  Test 1:      2  | 2  | true
  Test 2:      2  | 1  | true
  Test 3:      2  | 2  | false
  Test 4:      2  | 1  | false

Geben Sie eine Zahl zwischen 0 und 9 fur x1 ein : AZ
Achtung - keine Zahl eingegeben!
Ruckgabewert wird auf 0 gesetzt!
```

Bei der Verwendung der Operatoren mit Short-Circuit-Evaluation wird in diesem Programmlauf jeweils der als zweiter Operand eingesetzte Ausdruck nicht ausgewertet, da das Ergebnis der Operation bereits durch den Wert des ersten Operanden bestimmt wird. Dementsprechend wird auch die Increment-Operation im Ausdruck zum zweiten Operanden in diesem Fall nicht ausgeführt. Zur Vermeidung derartiger Situationen und der daraus gegebenenfalls entstehenden Laufzeitfehler aufgrund von Irrtümern gelten folgende Empfehlungen:

- Setzen Sie die Operatoren mit Short-Circuit-Evaluation stets mit äußerster Sorgfalt ein!
- Verwenden Sie Operatoren mit Short-Circuit-Evaluation nur dann, wenn sie deutliche Vorteile für das Programm mit sich bringen!

Die Klammern um die Gleichheitsoperatoren im Programm bsp02026.ShortCircuit sind im Übrigen eigentlich überflüssig, da der Gleichheitsoperator die Bindungspriorität 8 hat. Damit hat er eine höhere Bindungspriorität als die logischen Operatoren in den entsprechenden Ausdrücken. Lediglich der NICHT-Operator hat mit einer Bindungspriorität von 2 im vorliegenden Ausdruck eine höhere Bindungspriorität als der Gleichheitsoperator. Es ist jedoch eine gute Angewohnheit und verbessert die Lesbarkeit und Sicherheit von Programmen, wenn man wie in diesem Beispiel auch nicht unbedingt notwendige Klammern setzt.

2.4.4 Zuweisungsoperatoren

Zuweisungsoperatoren werden zur Veränderung des Wertes von Variablen eingesetzt. Der wichtigste Zuweisungsoperator ist die **Zuweisung**. Alle anderen Zuweisungsoperatoren sind eine Kombination der Zuweisung mit einem weiteren Operator. Die folgende Tabelle zeigt die Zuweisungsoperatoren bis auf die drei Operatoren, die sich als Kombination mit den bitweisen Operatoren (vgl. Kap. 2.4.5) ergeben:

Operator	Bezeichnung	Wirkung	Anwendung	Bindungspriorität	Assoziativität
`=`	Zuweisung	Berechnung des Ausdrucks rechts vom Operator und Zuweisung des errechneten Wertes zur Variablen links vom Operator	`a = b`	15	rechts
`+=`	Additionszuweisung	Kurzschreibweise für `a = a+b`	`a += b`	15	rechts
`-=`	Subtraktionszuweisung	Kurzschreibweise für `a = a-b`	`a -= b`	15	rechts
`*=`	Multiplikationszuweisung	Kurzschreibweise für `a = a*b`	`a *= b`	15	rechts
`/=`	Divisionszuweisung	Kurzschreibweise für `a = a/b`	`a /= b`	15	rechts
`%=`	Modulozuweisung - Berechnung des Divisionsrestes bei Division einer Zahl a durch eine Zahl b	Kurzschreibweise für `a = a%b`	`a %= b`	15	rechts
`&=`	UND-Zuweisung	Kurzschreibweise für `a = a&b`	`a &= b`	15	rechts
`|=`	ODER-Zuweisung	Kurzschreibweise für `a = a|b`	`a |= b`	15	rechts
`^=`	EXKLUSIV-ODER-Zuweisung	Kurzschreibweise für `a = a^b`	`a ^= b`	15	rechts

Die Zuweisung ("=") wurde bereits in Kapitel 2.2.4.3 erklärt. Alle anderen Zuweisungsoperatoren ergeben sich als Kombination der Zuweisung mit einem weiteren bereits erklärten Operator. Zusätzlich zu den Zuweisungsoperatoren in der Tabelle gibt es drei weitere Zuweisungsoperatoren, die sich als Kombination der Zuweisung mit den bitweisen Operatoren ergeben und die in Kapitel 2.4.5 erklärt werden.

Bei allen Zuweisungsoperatoren gilt die Besonderheit, dass sie eine **doppelte Wirkung** haben:

1. Das Ergebnis des Operators ist der Rückgabewert der Operation. In dieser Form werden die Zuweisungsoperatoren allerdings oft *nicht* verwendet.
2. Anders als bei den bisher erklärten Operatoren verändern Zuweisungsoperatoren auch einen *Operanden*, nämlich den linken Operand! Zuweisungsoperatoren werden in der Regel unter Ausnutzung dieser Eigenschaft verwendet.

Hierzu folgendes Beispiel:

```
a += b
```

hat die gleiche Wirkung wie `a = a+b` und weist der Variablen `a` die Summe der Werte der Variablen `a` und `b` zu. Zugleich ist die Summe jedoch auch *Rückgabewert* der Operation, das heisst es ist möglich, folgendes zu codieren:

```
c = a += b
```

In diesem Fall wird der Wert von `a+b` als Rückgabewert der Operation `a+=b` der Variablen `c` zugewiesen. Konsequenterweise kann man dann auch folgende Anweisung codieren:

```
c = a = a + b
```

Als Resultat haben dann alle drei Variablen den Wert von `b`!

Wir testen diese Zuweisungen im Programm bsp02027.DoubleEffect und erhalten folgenden Bildschirmausdruck:

```
Administrator: Eingabeaufforderung

...\jpr02027>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02027.DoubleEffect

--------------------------------------------------------------------------------
    Beispielprogramm zur doppelten Wirkung von Zuweisungsoperatoren
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)

--------------------------------------------------------------------------------

Beenden Sie das Einlesen
durch Eingabe (genau) nur der Taste F6 mit anschließendem Return!

Geben Sie eine einstellige ganze Zahl als Wert fur die Variable a ein : 2
Geben Sie eine einstellige ganze Zahl als Wert fur die Variable b ein : 1

Werte der Variablen nach Ausfuhrung der Operation:
  Operation      | c | a | b
  --------------+---+---+---
  a += b;        | - | 3 | 1
  c = a += b;    | 3 | 3 | 1
  c = a *= b;    | 2 | 2 | 1
  c = a = b;     | 1 | 1 | 1

Geben Sie eine einstellige ganze Zahl als Wert fur die Variable a ein : ^Z
Achtung - keine Zahl eingegeben!
Ruckgabewert wird auf 0 gesetzt!
```

Wie man auch am Bildschirmausdruck sieht, erfolgt die **Auswertung einer Folge von Zuweisungsoperatoren** im Prinzip stets *von rechts nach links*.

2.4.5 Weitere Operatoren

2.4.5.1 Überblick über die weiteren Operatoren

Neben den in Kapitel 2.3.2 und 2.4.1 bis 2.4.4 erklärten Operatoren gibt es noch folgende weitere Operatoren:

- bitweise Operatoren
- Schiebezuweisungen
- Fragezeichenoperator
- Type-Cast-Operatoren

Die **Type-Cast-Operator**en, auch kurz **Cast**s genannt, sind bereits in Kapitel 2.3.1.5 erklärt worden. Der Fragezeichenoperator wird in Kapitel 2.4.5.3 erklärt. Es folgen in Kapitel 2.4.5.2 zunächst die bitweisen Operatoren und Schiebezuweisungen.

2.4.5.2 Bitweise Operatoren und Schiebezuweisungen

Ausgangspunkt für das Verständnis der **bitweisen Operatoren** und **Schiebezuweisungen** ist die Tatsache, dass numerische Datentypen während des Ablaufs eines Java-Programms im Hauptspeicher als Bitfolge verschlüsselt werden. Die bitweisen Operatoren und Schiebezuweisungen greifen direkt auf diese Bitverschlüsselung zu, werten diese aus und ändern sie gegebenenfalls ab. Daher erfordert die Verwendung der Operatoren der folgenden Tabelle die genaue Kenntnis der Bitverschlüsselung von Zahlen und insbesondere auch die Kenntnis der Regeln zur Verschlüsselung negativer Zahlen. Andererseits werden bitweise Operatoren und Schiebezuweisungen nur selten wirklich benötigt.

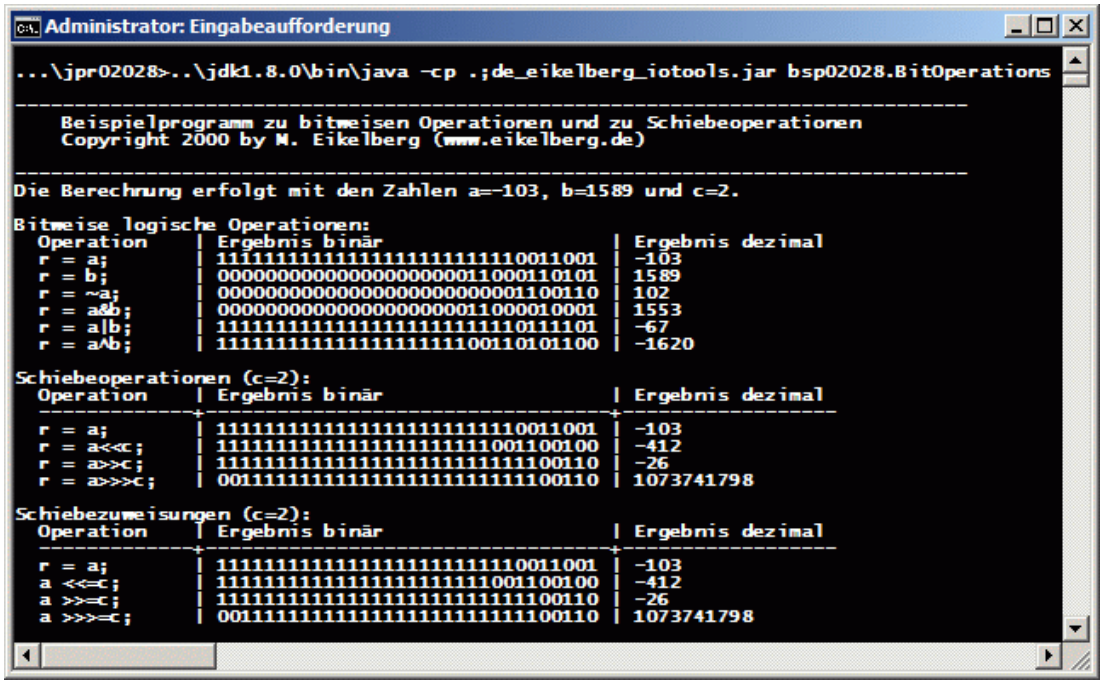
Operator	Bezeichnung	Wirkung	Anwendung	Bindungs-priorität	Assoziativität
~	Einerkomplement	Invertieren (Umkehren) aller Bits der Verschlüsselung von a	`~a`	2	rechts
&	bitweises UND	bitweise UND-Verknüpfung aller positionsgleicher Bits der Verschlüsselung zweier Zahlen a und b	`a&b`	9	links
|	bitweises nicht exklusives ODER	bitweise ODER-Verknüpfung aller positionsgleicher Bits der Verschlüsselung zweier Zahlen a und b	`a|b`	10	links
^	bitweises exklusives ODER	bitweise exklusive ODER-Verknüpfung aller positionsgleicher Bits der Verschlüsselung zweier Zahlen a und b	`a^b`	11	links
<<	Linksschieben	Verschieben aller Bits der Verschlüsselung einer Zahl a um b Positionen nach links mit Auffüllen der rechten Positionen durch Nullen	`a<<b`	6	links
>>	Rechtsschieben mit (Beibehaltung des) Vorzeichens	Verschieben aller Bits der Verschlüsselung einer Zahl a um b Positionen nach rechts. Am linken Ende des Speicherplatzes wird mit dem Bit-Wert aufgefüllt, den das linke äußere Bit vor der Schiebeoperation hatte. Dadurch bleibt das Vorzeichen der Zahl erhalten.	`a>>b`	6	links
>>>	Rechtsschieben ohne (Beibehaltung eines negativen) Vorzeichen(s)	Verschieben aller Bits der Verschlüsselung einer Zahl a um b Positionen nach rechts mit Auffüllen der linken Positionen durch Nullen. Hierdurch verlieren negative Zahlen ihr Vorzeichen.	`a>>>b`	6	links
<<=	Linksschiebezuweisung	Kurzschreibweise für `a = a<<b`	`a<<=b`	15	links
>>=	Rechtsschiebezuweisung mit (Beibehaltung des) Vorzeichen	Kurzschreibweise für `a = a>>b`	`a>>=b`	15	links
>>>=	Rechtsschiebezuweisung ohne (Beibehaltung des) Vorzeichen	Kurzschreibweise für `a = a>>>b`	`a>>>=b`	15	links

Als Beispiel für die Wirkung der bitweisen Operatoren wählen wir die Zahl -103. Als int Ganzzahl - also in 4 Byte Länge - gespeichert erhält

man die Darstellung dieser Zahl wie folgt:

Dualzahl zu +103	1100111_B
Binärcodierung in 4 Byte Länge	00000000 00000000 00000000 01100111
Invertierung (Vertauschen der Bitzustände)	11111111 11111111 11111111 10011000
Addition von 1_B ergibt Binärcodierung von -103	11111111 11111111 11111111 10011001

Wir wählen a=-103 und b=1589=11000110101_B sowie c=2, wobei c bei Schiebungen verwendet werden soll. Wendet man die Operatoren der obigen Tabelle auf diese Zahlen an, so erhält man folgende Ergebnisse:



Der Bildschirmausdruck zeigt, dass die bitweisen Operationen nicht zur unverständlichen Codierung der Ganzzahlarithmetik gedacht sind sondern vielmehr zur Programmierung hardwarenaher Aufgabenstellungen. *Man sieht ferner, dass ein unbeabsichtigtes Codieren von* `a&b` *für zwei ganze Zahlen* `a` *und* `b` *zu lauffähigem Code führt, dessen Ergebnis allerdings unerwartet ist.* Im obigen Beispiel erhält man zum Beispiel 1553=-103&1589. Auch für die anderen logischen Operatoren ist solch ein Missgeschick möglich.

2.4.5.3 Der Bedingungsoperator

Der Bedingungsoperator ist eigentlich ein Sprachelement, das nur sehr selten verwendet werden braucht. De facto ist es in den meisten Fällen möglich, statt des Bedingungsoperators eine if-Anweisung (vgl. Kapitel 2.5.1) zu codieren. Dementsprechend wird auch empfohlen, den Bedingungsoperator möglichst nicht einzusetzten. Ausserdem gibt es keine Darstellung des Bedingungsoperators durch ein Struktogramm-Element, so dass er bei konsequenter stukturierter Programmentwicklung auch **nicht erforderlich** ist. Der Hauptgrund für das Vorhandensein des Bedingungsoperators ist wohl die Existenz eines gleichartigen Operators in der Sprachdefinition der Programmiersprachen C bzw. C++. Der Bedingungsoperator ist der einzige dreistellige Operator in Java.

Operator	Bezeichnung	Wirkung	Anwendung	Bindungspriorität	Assoziativität
`? :`	Bedingungsoperator bzw. Fragezeichenoperator	Ist der logische Ausdruck a wahr, so wird der Ausdruck b ausgewertet und zurückgegeben. Ist a nicht wahr, so wird der Ausdruck c ausgewertet und zurückgegeben.	`a?b:c`	-	rechts

Man beachte, dass der Bedingungsoperator *von rechts nach links* gruppiert, das heisst

```
a ? b : c ? d : e
```

ist gleichbedeutend mit

```
a ? b : (c ? d : e)
```

In der Begleitliteratur zu Java gibt es keine Aussagen zur Bindungspriorität des Bedingungsoperators, wohl aber ausführliche und komplizierte Erläuterungen zum Rückgabewert des Bedingungsoperators, wenn in `a?b:c` die Ausdrücke b und c von unterschiedlichem Datentyp sind. Insgesamt ergeben sich daraus - auch unter Beachtung der Regeln strukturierter Programmierung - folgende Regeln zum Bedingungsoperator:

- Man sollte den Bedingungsoperator möglichst nicht verwenden.

- Wenn man den Bedingungsoperator verwendet, sollte man die logische Gruppierung durch Klammern festlegen.
- Bei Verwendung des Bedingungsoperators `a?b:c` sollten die Ergebnisse der Ausdrücke b und c vom gleichen Datentyp sein.

Das Programm bsp02029.CondExprOperator ist ein Beispielprogramm zum Fragezeichenoperator. Dieses Programm liest eine von der Tastatur aus eingegebene Zeile von Schriftzeichen ein und zählt die Leerzeichen sowie die vom Leerzeichen verschiedenen Schriftzeichen jeweils getrennt. Zu jedem Zeichen wird vom Programm ausgegeben, wie sich der zugehörige Zähler verändert hat. Die eigentliche Verarbeitung des Programms besteht aus folgendem Programmcode:

```
...

// Variablen
    final char LEERZEICHEN = ' ';   // Konstante für das Leerzeichen
    int leerZeichenZahl = 0;        // Zähler für das Leerzeichen
    int anderesZeichenZahl = 0;     // Zähler für andere Zeichzen
    int veraendert = 0;             // Variable für die Veränderung
    char zeichen;                   // Variable für das jeweils einzulesende Zeichen

// Verarbeitung
    do
    {
// Einlesen der Zeichen
      zeichen = KeyboardIn.kReadChar();
// Verarbeitung bis Zeilenende/Return (Zeichenfolge \r\n)
      if (zeichen != '\r')
      {
        veraendert =
          zeichen==LEERZEICHEN ?
            ++leerZeichenZahl :
            ++anderesZeichenZahl;
      ScreenOut.kPrint("Gelesenes Zeichen: "+zeichen+" / ");
      ScreenOut.kPrintln("Veränderung Zähler: "+ (veraendert-1) +" -> "+veraendert);
      }
// Schleife bis Zeilenende/Return (Zeichenfolge \r\n)
    } while (zeichen != '\r');

...
```

Das Beispielprogramm nutzt aus, dass die Betätigung der Tasten Return oder Enter zur Erzeugung der beiden nicht druckbaren Schriftzeichen mit Namen Return (Zeichen dezimal 10 bzw. '\r') und Newline (Zeichen dezimal 13 bzw. '\n') führt.

Die Anwendung des Bedingungsoperators geschieht in folgendem Codebestandteil:

```
        veraendert =
          zeichen==leerZeichen ?
            ++leerZeichenZahl :
            ++anderesZeichenZahl;
```

Dieser Code führt zu folgender Verarbeitung: Zunächst wird geprüft, ob das eingelesene Zeichen - abgespeichert auf der Variablen zeichen - gleich dem Leerzeichen - abgespeichert auf der final Variablen leerZeichen - ist. Liegt Gleichheit vor, so wird die Operation `++leerZeichenZahl` vorgenommen und der neue Wert anschließend auf der Variablen veraendert gespeichert. Bei Ungleichheit wird die Operation `++anderesZeichenZahl` durchgeführt und dieser neue Wert auf der Variablen veraendert gespeichert. Man beachte, dass eine Codierung `leerZeichenZahl++` zu einer Erhöhung des Wertes von leerZeichenZahl *nach* Verarbeitung der gesamten Zeile zur Folge gehabt hätte, so dass der Variablen veraendert noch der alte niedrigere Wert zugewiesen worden wäre.

Das Programm bsp02029.CondExprOperator hat folgende Wirkung:

```
Administrator: Eingabeaufforderung

...\jpr02029>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02029.CondExprOperator

--------------------------------------------------------------------------------
    Beispielprogramm zum Bedingungsoperator
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)

--------------------------------------------------------------------------------
Lesen einer Zeile, getrenntes Zählen der Leerzeichen sowie der von Leer-
zeichen verschiedenen Zeichen und Protokollierung der Veränderung der Zähler.


Geben Sie eine Zeile mit Schriftzeichen ein:
12 3#5  abc
Gelesenes Zeichen: 1 / Veränderung Zähler: 0 -> 1
Gelesenes Zeichen: 2 / Veränderung Zähler: 1 -> 2
Gelesenes Zeichen:   / Veränderung Zähler: 0 -> 1
Gelesenes Zeichen: 3 / Veränderung Zähler: 2 -> 3
Gelesenes Zeichen: # / Veränderung Zähler: 3 -> 4
Gelesenes Zeichen: 5 / Veränderung Zähler: 4 -> 5
Gelesenes Zeichen:   / Veränderung Zähler: 1 -> 2
Gelesenes Zeichen:   / Veränderung Zähler: 2 -> 3
Gelesenes Zeichen: a / Veränderung Zähler: 5 -> 6
Gelesenes Zeichen: b / Veränderung Zähler: 6 -> 7
Gelesenes Zeichen: c / Veränderung Zähler: 7 -> 8

Zahl der Leerzeichen:     3
Zahl der anderen Zeichen: 8
```

2.4.6 Operator-Vorrangregeln

In den Tabellen zu den verschiedenen in Java definierten Operatoren werden zu jedem Operator Werte zur Bindungspriorität und Assoziativität genannt. Die **Bindungspriorität** wurde bereits in Kapitel 2.3.2.3 erklärt. Sie regelt die logische Gruppierung der Operanden - also der Variablen und Konstanten - um die Operatoren, legt also fest, wie bei nicht explizit codierten Klammern vom Compiler Klammern in einen Ausdruck eingefügt werden.

Bei einer Häufung von Operatoren gleicher Bindungskraft kann allerdings die Gruppierung der Operatoren und Operanden nicht mehr durch die Bindungspriorität geregelt werden. Eine entsprechende Regelung erfolgt hier durch die **Assoziativität**. Insgesamt gilt daher folgende Regel:

Bindungspriorität und **Assoziativität** bestimmen die logische Gruppierung der Operatoren in einem Ausdruck, sofern die Gruppierung nicht durch Klammern vom Programmierer vorgegeben wird. Man beachte, dass wie bereits in Kapitel 2.3.2.3 erklärt durch Bindungspriorität und Assoziativität lediglich die Gruppierung der Operatoren und Operanden eindeutig bestimmt wird, nicht jedoch die Ausführungsreihenfolge bei der Auswertung des Ausdrucks. Die **Auswertungsreihenfolge** - also die zeitliche Abfolge der Auswertung der einzelnen Operationen in einem Ausdruck - ergibt sich aus den Regeln zur Anwendung der Inkrement- und Dekrementoperationen sowie aus der logischen Gruppierung gemäß Bindungspriorität und Assoziativität.

Die Regeln zur Bindungspriorität und Assziativität zusammen bilden die sogenannten **Operator-Vorrangregeln**. Anhand der Operator-Vorrangregeln wird also die Gruppierung von Operatoren und Operanden festgelegt. Insgesamt gibt es folgende Regeln:

1. **Bindungsregel:** Sind keine Klammern vorhanden bzw. innerhalb von geklammerten Bereichen gilt *zwischen Operatoren unterschiedlicher Bindungspriorität* die logische Gruppierung aufgrund der Bindungspriorität der Operatoren: Zunächst werden alle Ausdrücke der *höchsten* Bindungspriorität logisch gruppiert. Begonnen wird also mit Bindungspriorität 1. Anschließend erfolgt die Gruppierung aller Ausdrücke der nächsten Bindungspriorität 2 usw.
2. **Assoziativitätsregel:** Die Gruppierung von *Operatoren gleicher Bindungspriorität* erfolgt je Operator gemäß den Eintragungen zur Assoziativität in den Tabellen zu den Operatoren. Dabei wird zur Assoziativität immer die Stelle genannt, bei der die Gruppierung beginnt. Ein Eintrag "links" bedeutet also die Gruppierung der Operatoren von links nach rechts. Eine Eintragung zur Assoziativität ist nicht für alle Operatoren sinnvoll, da einige Operatoren nicht mehrfach auf gleicher Ebene auftreten können.
3. Logische Gruppierungen, die mit Hilfe von **runden Klammern** vorgegeben werden, überschreiben die durch Bindungsregel und Assoziativitätsregel vorgegebenen Reihenfolgen. Innerhalb von geklammerten Bereichen gilt wieder (isoliert betrachtet) die durch Bindungsregel und Assoziativitätsregel vorgegebene Reihenfolge.

Als Beispiel für die Anwendung der Bindungsregel kann die Tatsache gelten, dass auch bei Java-Ausdrücken die Regel Punktrechnung (Bindungspriorität 4) vor Strichrechnung (Bindungspriorität 5) gilt.

Zum Begriff der Bindungspriorität beachte man folgende Bemerkung: In der Literatur werden statt des Begriffs Bindungspriorität die Begriffe **Priorität** und **Bindungskraft** verwendet. Sowohl der Begriff Bindungskraft als auch der äquivalente Begriff Priorität sind ungünstig. Der Begriff Bindungskraft passt zu dem Sachverhalt, dass es bei der Bindungsregel um die logische Gruppierung von Operatoren in einem Ausdruck geht. Ungünstig ist, dass der Wert 1 den höchsten und der Wert 2 einen *geringeren* Wert bedeutet. Bei dem Begriff Priorität passt dagegen, dass Ausdrücke mit Wert 1 zuerst gruppiert werden, jedoch ist die (Bindungs-)Priorität nicht gleich der Auswertungsreihenfolge, wie man am Postinkrement- und Postdekrementoperator sieht (vgl. Ausdruck 4 im Beispiel weiter unten). Beide Gründe führen dazu, dass in diesem Kurs der Begriff Bindungspriorität verwendet wird.

Man beachte, das durch die obigen Regeln 1 bis 3 noch keine Aussagen über die Reihenfolge der Auswertung von Teilausdrücken gemacht wird. Hier gelten folgende Regeln:

4. Die Auswertungsreihenfolge zu Operatoren mit Bindungspriorität 1 ergibt sich wie folgt: Präinkrement- und Prädekrementoperatoren werden unmittelbar vor der Auswertung des umgebenden Ausdrucks ausgeführt. Postinkrement- und Postdekrementoperatoren werden unmittelbar nach Auswertung des umgebenden Ausdrucks ausgeführt.
5. Abgesehen von der Bindungspriorität 1 bestimmen Bindungspriorität und Assoziativität auch die Auswertungsreihenfolge - nach Auswertung der jeweiligen Präinkrement- und Prädekrementoperatoren werden zunächst alle Operatoren der Bindungspriorität 2 ausgewertet, anschließend die Operatoren der Bindungspriorität 3 usw. Die Auswertung von Operationen gleicher Bindungspriorität erfolgt je Operator gemäß der in der Tabelle genannten Assoziativität: Eine Assoziativität "links" ergibt eine Auswertung von links nach rechts, eine Assoziativität "rechts" ergibt eine Reihenfolge von rechts nach links.

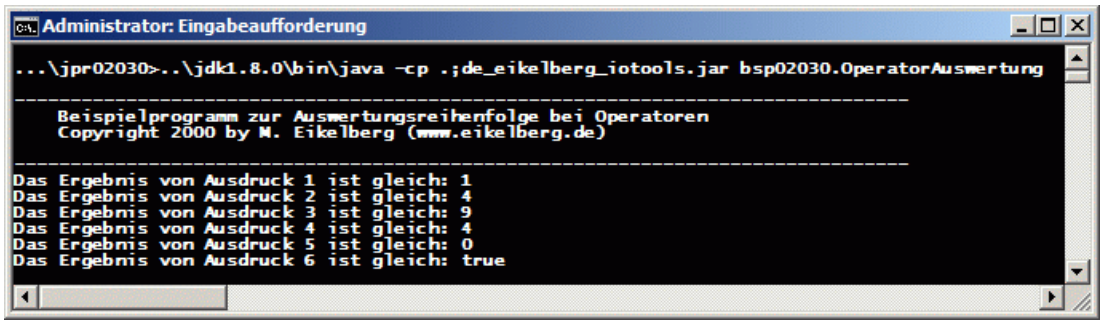
Insgesamt ergeben alle 5 Regeln stets eine eindeutige logische Gruppierung sowie eine eindeutige Auswertungsreihenfolge. Allerdings ist diese Reihenfolge oft nicht direkt ersichtlich und es empfiehlt sich daher, zur Sicherheit Gruppierungen mit Hilfe von logischen Klammern im Quellcode vorzunehmen.

Einfache Beispiele enthält der folgende Ausschnitt des Beispielprogramms bsp02030.OperatorAuswertung

```
// Variablendeklaration
    int i1=8, i2=4, i3=2, i4, i5, i6, i7, i8;
    boolean b1;
// Ausdruck 1 - Assoziativität der Division ist links
//              Auswertung von links nach rechts
    i4 = i1 / i2 / i3;
// Ausdruck 2 - Setzen von Klammern
    i5 = i1 / (i2 / i3);
// Ausdruck 3 - verschiedene Prioritäten und Auswertungsreihenfolge
```

```
    i6 = ++i3 * 6 / --i3;
// Ausdruck 4 - verschiedene Prioritäten und Auswertungsreihenfolge
    i7 = i3++ * 6 / i3--;
// Ausdruck 5 - Punktrechnung (Priorität 4) geht vor Strichrechnung (Priorität 5)
    i8 = i3 / i2 - i2 / i1;
// Ausdruck 6 - Klammerung
    b1 = i1 > i2 & i3 > 0;
```

Die Ergebnisse dieser 6 Ausdrücke zeigt der folgende Bildschirmausdruck:



In Ausdruck 1 führt die Assoziativität der Division zu einer Auswertungsreihenfolge von links nach rechts, also zu einer Auswertung äquivalent zu folgender Klammerung:

```
i4 = (i1 / i2) / i3;
```

Man erhält also zunächst 8/4=2 und daraus mit 2/2=1 das Gesamtergebnis 1.

In Ausdruck 2 führt das Setzen von Klammern zu einer gegenüber Ausdruck 1 veränderten Reihenfolge - zunächst ergibt der Ausdruck in der Klammer 4/2=2, anschließend erhält man aus 8/2=4 das Gesamtergebnis 4.

In Ausdruck 3 sind jeweils zwei Operationen der Bindungspriorität 1 und 4 gemischt. Die Operatoren der Bindungspriorität 4 sind der Multiplikations- und Divisionsoperator, die gemäß der Assoziativität von links nach rechts abgearbeitet werden. Präinkrement und Prädekrement werden unmittelbar vor der Auswertung des umgebenden Ausdrucks ausgewertet. Es wird also zunächst i3 um eins erhöht und man erhält eine 3. Die anschließende Multiplikation mit 6 ergibt 18. *Erst jetzt wird der Postdekrementoperator ausgeführt, i3 also wieder zu 2!* Die Division von 18 durch 2 ergibt dann das Endergebnis 9.

Ausdruck 4 entspricht Ausdruck 3 bis auf die Verwendung von Postdekrement und Postimkrement. Die Assoziativität von Multiplikation und Division ist links, es wird also zunächst der Inhalt von i3 mit 6 multipliziert und man erhält 12. Anschließend wird die Postinkrementoperation ausgeführt und der Inhalt von i3 um eins auf 3 erhöht. Der nächste Schritt besteht in der Ausführung der Division und man erhält 12/3=4. Der Wert 4 ist zugleich der Wert des gesamten Ausdrucks. Zum Abschluss wird der Inhalt von i3 wieder um eins auf 2 vermindert, was aber keine Auswirkung auf den Wert des gesamten Ausdrucks hat (wohl aber gegebenenfalls auf die weitere Verarbeitung im Programm).

Ausdruck 5 illustriert die Gültigkeit der Regel "Punktrechnung geht vor Strichrechnung" in der Sprachdefinition von Java. Die Bindungspriorität der Subtraktion ist mit einem Wert von 5 *niedriger* als die Bindungspriorität 4 von Multiplikation und Division.

Ausdruck 6 besteht aus einer Kombination zweier relationaler Operatoren mit einem logischen Operator. Die Bindungspriorität der relationalen Operatoren ist mit einem Wert von 7 *höher* als die Bindungspriorität 9 des logischen Operators, so dass der logische Operator zuletzt ausgeführt wird.

2.5 Steuerungselemente

2.5.1 Überblick

Die allerwenigsten Programme bestehen aus einem festen Ablauf von Anweisungen. Vielmehr müssen in der Regel im Programm situationsabhängige Entscheidungen getroffen werden, Auswahlmöglichkeiten angeboten und gleichartige Abläufe mehrfach wiederholt durchgeführt werden. Daher gibt es in allen Programmiersprachen Sprachelemente, die als **Steuerungselemente** bezeichnet werden.

Die Logik von Steuerungselementen ist bereits im Kapitel über Struktogramme (Kap. 1.5.2) angesprochen worden. Dort wurden die **selektierenden** sowie die **iterierenden Strukturblöcke** erklärt. Einen Überblick über diese Strukturblöcke, die zugehörigen Java Sprachelemente sowie den Aufbau dieses Kapitels bietet die folgende Tabelle:

Typ des Strukturblocks	Strukturblock	Java Sprachelement	Kapitel-Nummer
Selektierender Strukturblock	bedingte Anweisung	**if**-Anweisung	2.5.2
	Auswahlanweisung	**switch**-Anweisung	2.5.3
Iterierender Strukturblock	bedingte Ausführung mit Wiederholung	**while**-Schleife	2.5.4
	bedingte Wiederholung	**do**-Schleife	2.5.5
	Zählschleife	**for**-Schleife	2.5.6

Bei der Beschreibung der Syntax der Sprachelemente ist es sinnvoll, diese Sprachelemente als Mischung einzelner Java Sprachelemente sowie größerer Codebestandteile wie Ausdrücke und Anweisungen darzustellen. Dabei wird Java Programmcode wie bisher in `dicktengleicher Schrift` (nicht proportionale Schrift - alle Buchstaben haben die gleiche Breite) dargestellt, größere Codebestandteile jedoch *`kursiv`* . So sind zum Beispiel in der Anweisung

```
if (logAusdruck) Anweisung1 else Anweisung2;
```

die Bestandteile

```
if ( ) else ;
```

direkte Bestandteile des Java Programmcodes. Dagegen stehen

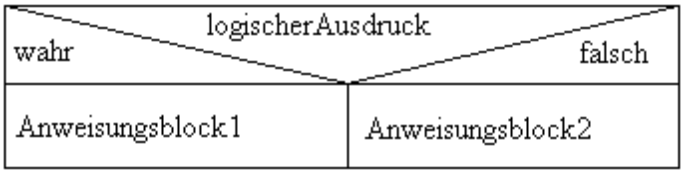
```
logAusdruck Anweisung1 Anweisung2
```

für einen aus Java Programmcode gebildeten logischen Ausdruck beziehungsweise zwei Anweisungen oder sogar Anweisungsblöcke.

2.5.2 if-Anweisungen

2.5.2.1 Das Sprachelement zur if-Anweisung

if-Anweisungen werden zur Umsetzung der Logik einer **bedingten Anweisung** eingesetzt. Bei dem Entwurf eines Programms wird diese bedingte Anweisung durch folgenden Strukturblock dargestellt:



In dieser Graphik stellt "logischerAusdruck" einen logischen Ausdruck und "Anweisungblock1" sowie "Anweisungsblock2" Anweisungsblöcke dar.

Die bedingte Anweisung wird wie folgt in Java Syntax umgesetzt:

```
if (logischerAusdruck )
  Anweisungsblock1
else
  Anweisungsblock2 ;
```

Der obige Programmausdruck besteht also aus einer Kombination Java Schlüsselwörtern, einem Ausdruck sowie zwei Anweisungsblöcken. Zwingend erforderlich ist es, dass *logischerAusdruck* ein Ausdruck vom Datentyp **boolean** ist. Insgesamt bildet das Java Sprachelement zur bedingten Anweisung wieder eine Anweisung, weshalb das Sprachelement **if-Anweisung** heisst.

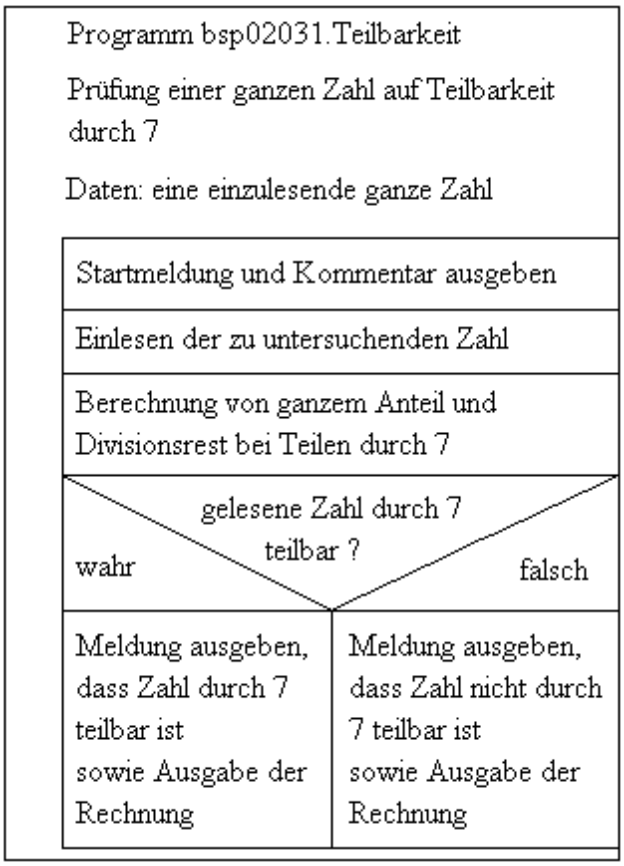
Zur if-Anweisung gibt es die folgende Variante: Sollen in dem Fall, dass die logische Bedingung *logischerAusdruck* falsch ist, keine Anweisungen ausgeführt werden, ist mit anderen Worten der Anweisungsblock *Anweisungsblock2* leer, so kann die folgende verkürzte Version der if-Anweisung verwendet werden:

```
if (logischerAusdruck )
  Anweisungsblock1 ;
```

Statt Anweisungsblöcken können in der if-Anweisung auch lediglich nur einzelne Anweisungen verwendet werden.

2.5.2.2 Beispiel zur if-Anweisung

Ein erstes Beispiel zur if-Anweisung ist das Programm bsp02031.Teilbarkeit, das eine long Ganzzahl einliest und daraufhin prüft, ob diese Zahl durch 7 teilbar ist. Eine mögliche Lösung dieser Programmieraufgabe beschreibt der folgende Modulblock:



Der Programmabschnitt der Blöcke "Einlesen der zu untersuchenden Zahl" bis zum Ende der main-Methode kann durch folgenden Programmcode realisiert werden:

```
// Einlesen der zu untersuchenden Zahl
    long zahl;
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Geben Sie eine zu testende Zahl ein: ");
    zahl = KeyboardIn.kReadLong();

// Berechnung von ganzem Anteil und Divisionsrest bei Teilen durch 7
// (Divisionsrest und ganzer Anteil haben Vorzeichen der Ausgangszahl)
    long mod7Rest,mod7GanzerAnteil;
    mod7GanzerAnteil = (zahl / 7);      // ganzer Anteil
    mod7Rest = zahl - (zahl / 7) * 7;   // Divisionsrest

// Prüfung, ob die gelesene Zahl durch 7 teilbar ist
// und Ausgabe des Ergebnisses
    ScreenOut.kPrintln();
    if (mod7Rest == 0)
    {
      ScreenOut.kPrintln("Die eingelesene Zahl ist durch 7 teilbar!");
      ScreenOut.kPrintln("Es gilt: "+zahl+" / 7 = "+mod7GanzerAnteil+".");
    }
    else
    {
      ScreenOut.kPrintln("Die eingelesene Zahl ist nicht durch 7 teilbar!");
      ScreenOut.kPrintln(
        "Es gilt: "+zahl+" / 7 = "+mod7GanzerAnteil+" Rest "+mod7Rest+".");
    }
```

Man beachte, dass sowohl für den Fall "wahr" als auch für den Fall "falsch" in der if-Anweisung jeweils ein Anweisungsblock aus mehr als einer Anweisung codiert wurde. Solche Blöcke müssen unter Verwendung von geschweiften Klammern gebildet werden, da ansonsten die Struktur der gesamten if-Anweisung nicht eindeutig ist. Die geschweiften Klammern können allerdings entfallen, wenn ein Anweisungsblock nur aus einer Anweisung besteht wie in folgendem Beispiel (Programm bsp02031.TeilbarkeitKurz):

```
    if (mod7Rest == 0)
      ScreenOut.kPrintln("Die eingelesene Zahl ist durch 7 teilbar!");
    else
      ScreenOut.kPrintln("Die eingelesene Zahl ist nicht durch 7 teilbar!");
```

Läßt man jetzt noch das Schlüsselwort else und die darauffolgende Anweisung weg, so erfolgt in dem Fall, dass die eingelesene Zahl nicht

durch 7 teilbar ist auch keine Verarbeitung. Der Programmcode sieht dann wie folgt aus (Programm bsp02031.TeilbarkeitOhneElse):

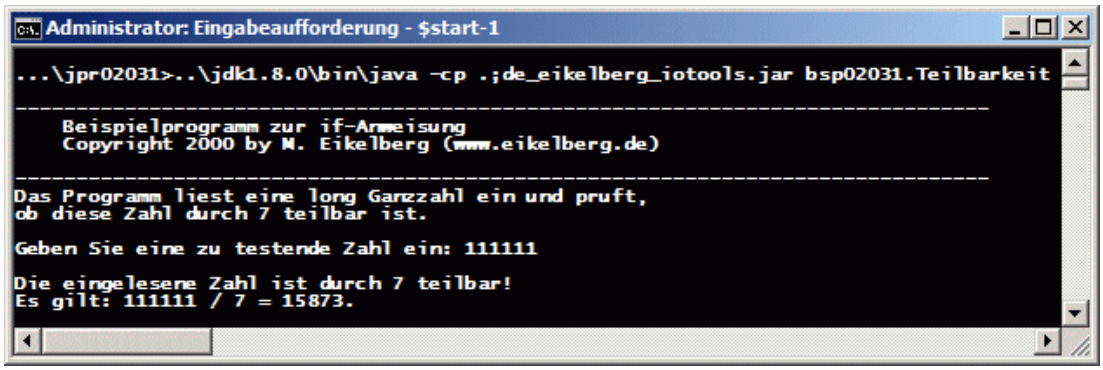
```
if (mod7Rest == 0)
  ScreenOut.kPrintln("Die eingelesene Zahl ist durch 7 teilbar!");
```

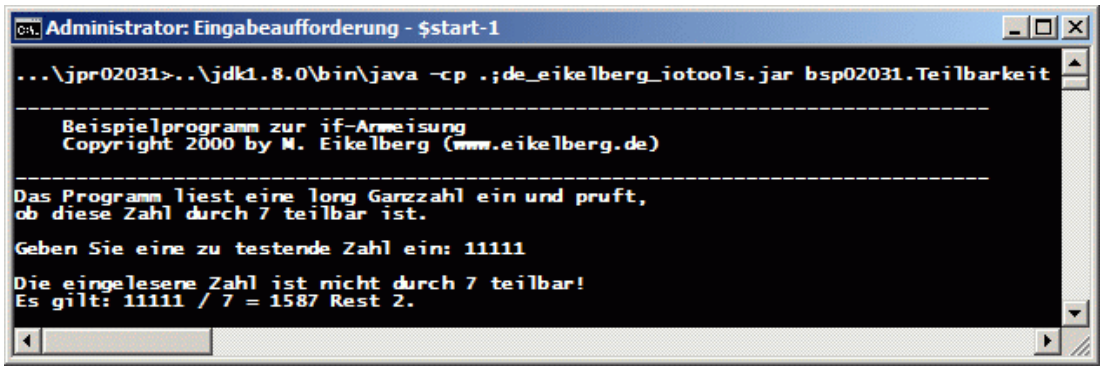
Man beachte im Programm bsp02031.Teilbarkeit im Übrigen auch folgende Anweisungen:

```
mod7GanzerAnteil = (zahl / 7);       // ganzer Anteil
mod7Rest = zahl - (zahl / 7) * 7;    // Divisionsrest
```

Die erste Anweisung ergibt - wie in 2.3.2.4 beschrieben - unter Verwendung der Regel "long durch long ergibt long" den ganzen Anteil bei der Division zweier ganzer Zahlen. Die zweite Anweisung verwendet die Regel "long durch long ergibt long" zur Berechnung des Divisionsrestes und stellt eine seit vielen Jahren von vielen Programmierern verwendete Vorgehensweise dar, für die heute als Alternative auch der Modulo-Operator zur Verfügung steht. Natürlich hätte in der zweiten Anweisung das Ergebnis der ersten Anweisung wiederverwendet werden können.

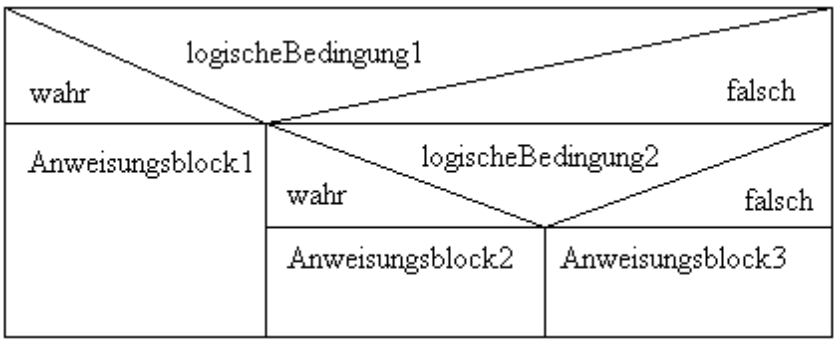
Programmausgaben, die die Wirkung der beiden verschiedenen Anweisungsblöcke der if-Anweisung illustrieren, zeigen die folgenden beiden Bildschirmausdrucke:





2.5.2.3 Schachtelung von if-Anweisungen

Oft ist es zur Umsetzung der Programmlogik erforderlich, if-Anweisungen ineinander zu schachteln wie in folgendem Struktogramm:

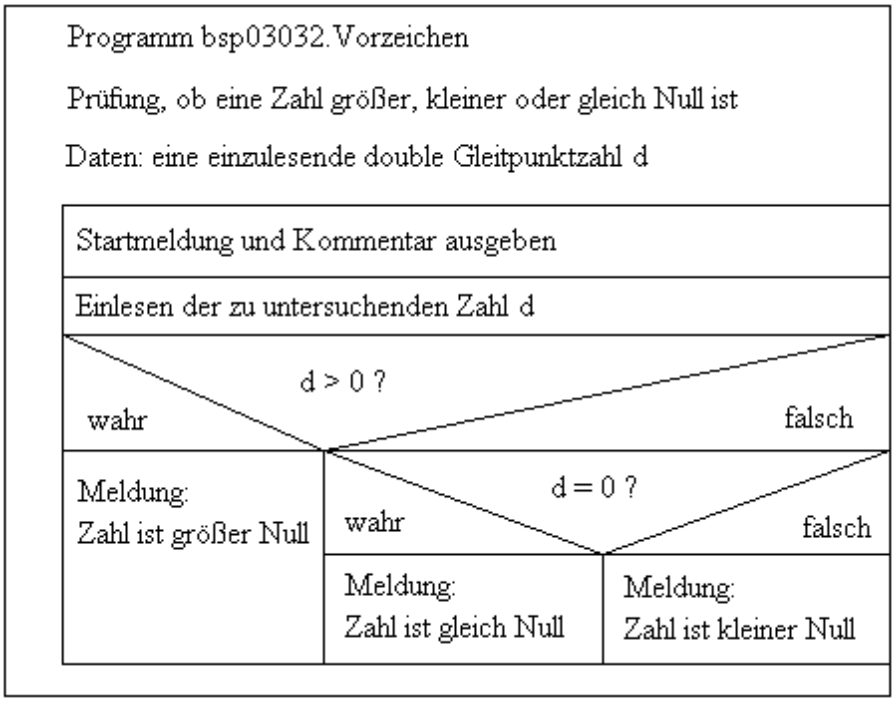


Bei der Programmierung müssen dann zwei if-Anweisungen wie folgt geschachtelt werden:

```
if (logischeBedingung1)
  Anweisungsblock1
else
  if (logischeBedingung2)
    Anweisungsblock2
  else
```

```
Anweisungsblock3
```

Ein Beispiel für eine solche Situation ist das Programm jpr02032.Vorzeichen. Dieses Programm liest eine Gleitpunktzahl ein und prüft, ob die Zahl positives oder negatives Vorzeichen hat oder aber ob die Zahl gleich Null ist. Der Programmentwurf von bsp02032.Vorzeichen begann mit der Erstellung des folgenden Struktogramms:



Die verschachtelten if-Abfragen des Programms wurden wie folgt codiert:

```
// Prüfung der eingelesenen Zahl
    ScreenOut.kPrintln();
    if (d > 0)
    {
      ScreenOut.kPrintln("Die Zahl ist größer Null.");
    }
    else
    {
      if (d == 0)
      {
        ScreenOut.kPrintln("Die Zahl ist gleich Null.");
      }
      else
      {
        ScreenOut.kPrintln("Die Zahl ist kleiner Null.");
      }
    }
```

Die geschweiften Klammern im Programmcode sind nicht notwendig, da die Anweisungsblöcke jeweils nur aus einer Anweisung bestehen. Daher hätte man auch folgenden Programmcode schreiben können:

```
// Prüfung der eingelesenen Zahl
    ScreenOut.kPrintln();
    if (d > 0)
      ScreenOut.kPrintln("Die Zahl ist größer Null.");
    else
      if (d == 0)
        ScreenOut.kPrintln("Die Zahl ist gleich Null.");
      else
        ScreenOut.kPrintln("Die Zahl ist kleiner Null.");
```

Ohne geschweifte Klammern wird jedoch die Blockstruktur im Programmcode nicht mehr so deutlich. Zudem ermöglicht das Setzen der Klammern auch eine einfache Kontrolle der Blockstruktur nach folgender Regel:

1. Nach jeder sich öffnenden geschweiften Klammer muss der Programmcode nach rechts eingerückt werden.
2. Nach jeder sich schließenden geschweiften Klammer muss der Programmcode nach links eingerückt werden.

Passen bei der Erstellung des Programmcodes die Einrückungen nicht mehr zusammen, so ist dies ein Anzeichen dafür, dass die Blockstruktur

im Programmcode nicht korrekt umgesetzt worden ist. Dementsprechend wird insgesamt die Beachtung der folgenden Regel dringend empfohlen:

Bei der Codierung von verschachtelten if-Anweisungen sollten stets mit Hilfe von geschweiften Klammern Anweisungsblöcke codiert werden!

2.5.2.4 Formulierung von Vergleichsoperationen für Gleitpunktzahlen

Bei der Formulierung von Verleichsoperationen für Gleitpunktzahlen muss die Rechengenauigkeit bei Berechnungen mit Programmen beachtet werden. Dies ist insbesondere von Bedeutung, wenn diese Vergleichsoperationen im Zusammenhang mit Steuerungselementen verwendet werden.

> Die Rechenungenauigkeit bei Rechnungen mit Gleitpunktzahlen führt dazu, dass am Ende einer berechneten Gleitpunktzahl eine oder mehrere Stellen ungenau sind. Daher ist eine Abfrage auf Gleichheit des Inhaltes einer Gleitpunktvariablen mit einer Konstanten in der Regel nicht sinnvoll. Auch bei der Formulierung von Ungleichungen muss die Rechenungenauigkeit berücksichtigt werden.

Die nachfolgende Grafik zeigt anhand eines Zahlenstrahls, wie die Bewertung erfolgen kann, ob eine Zahl als gleich Null angesehen werden kann. Hierzu definiert man zunächst eine Variable - zum Beispiel für die Verwendung in einer main-Methode als statische Klassenvariable - mit dem Namen EPS, die die maximal als Rechenfehler zu wertende Zahl angibt:

```
  private static final double EPS = 1e-12;
```

Alle im Intervall [-EPS,EPS] liegende Zahlen werden dann im Programm als gleich Null angesehen. Dabei ist es in der Praxis unerheblich, ob die Intervallgrenzen -EPS und EPS noch als gleich Null angesehen werden sollen oder nicht:

Im Programm erfolgt dann die Bewertung, ob eine Zahl gleich Null ist, wie in folgemdem Beispielprogramm jpr02232:

```
package bsp02232;

import de.eikelberg.iotools.*;

public class Vorzeichen
{
  private static final double EPS = 1e-12;

  public static void main(String[] args)
  {
    ...

// Einlesen der zu untersuchenden Zahl d
    double d;
    ScreenOut.kPrintln();
    ScreenOut.kPrint(
      "Geben Sie eine double Gleitpunktzahl ein: ");
    d = KeyboardIn.kReadDouble();

// Prüfung der eingelesenen Zahl
    ScreenOut.kPrintln();
    if (d > EPS)
    {
      ScreenOut.kPrintln("Die Zahl ist größer Null.");
    }
    else
    {
      if (Math.abs(d) <= EPS)
      {
        ScreenOut.kPrintln("Die Zahl ist gleich Null.");
      }
      else
```

```
      {
        ScreenOut.kPrintln("Die Zahl ist kleiner Null.");
      }
    }
  }
}
```

Es gilt dann:

`d`	`>`	`EPS`	=> Zahl gilt als **größer** Null
`Math.abs(d)`	`<=`	`EPS`	=> Zahl gilt als **gleich** Null
`d`	`<`	`-EPS`	=> Zahl gilt als **kleiner** Null

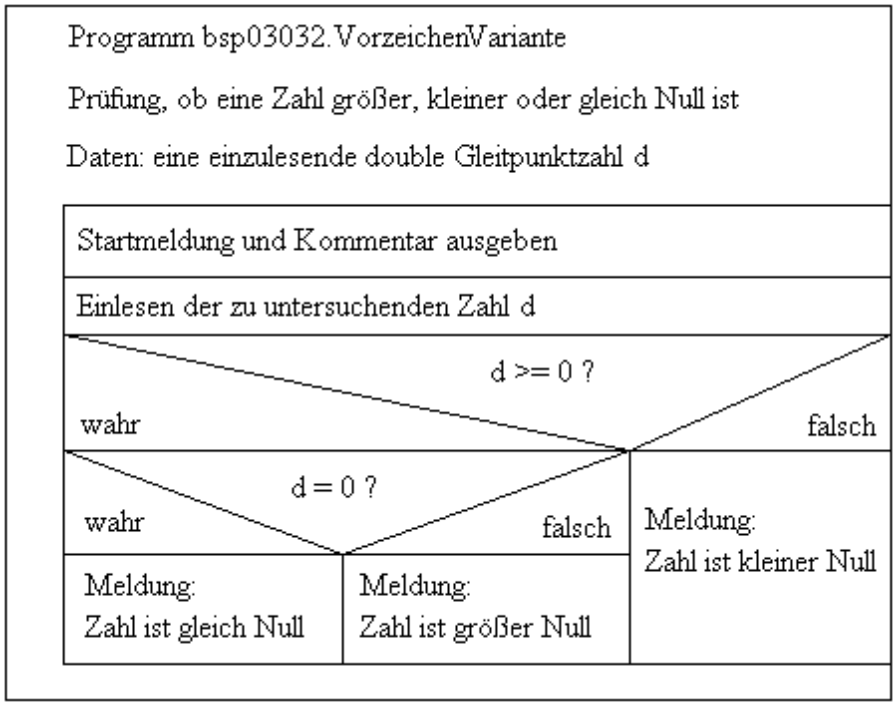
Die Wahl der Zahl für die Variable EPS hängt von den bei der Berechnung enstehenden Zahlen ab. Oft reicht es, 12 von Null verschiedene Stellen als gültig anzusehen und die nachfolgenden Stellen als durch Rechenungenauigkeit verändert. Dabei ist jeodch zu berücksichtigen, in welchem Größenbereich die auftretenden Zahlen liegen. Sind die Zahlen zum Beispiel in der Größenordnung 10^{-6} und sollen 12 Stellen als genau angesehen werden, so muss EPS gleich $10^{-6-12} = 10^{-18}$ gewählt werden.

2.5.2.5 dangling else

Wenn man den Programmcode von bsp02032.Vorzeichen betrachtet, stellt sich nicht die Frage, **welches else Schlüsselwort zu welchem if** gehört. Dies muss aber nicht unbedingt so sein, vielmehr kann man auch Java Programmcode schreiben, so dass es beim Lesen des Programmcodes zunächst nicht unbedingt klar ist, welches else Schlüsselwort zu welchem if gehört. Das entsprechende Problem ist in der Informatik als **dangling else** bekannt.

Im vorliegenden Beispiel hätte die Programmlogik zum Beispiel auch anders realisiert werden können. Liegt ein großer zu prüfender Datenbestand vor und ist die Mehrheit der Zahlen negativ, so müssen für die Mehrheit der Zahlen zwei if-Anweisungen durchlaufen werden. Um die Ausführungsgeschwindigkeit des Programms zu erhöhen, hätte man den Programmablauf wie im folgenden Beispielprogramm bsp02032.VorzeichenVariante umgesetzt planen können.

Die Programme bsp02032.Vorzeichen und bsp02032.VorzeichenVariante unterscheiden sich in Bezug auf das Ergebnis nicht, wohl jedoch in Bezug auf ihre innere Struktur. Das Programm bsp02032.VorzeichenVariante ist ausgehend von folgendem Struktogramm erstellt worden:



Zahlen kleiner Null durchlaufen jetzt nur noch eine if-Anweisung. Der Programmcode zu den geschachtelten if-Abfragen lautet wie folgt:

```
// Prüfung der eingelesenen Zahl
    ScreenOut.kPrintln();
    if (d >= 0)
    {
      if (d == 0)
      {
        ScreenOut.kPrintln("Die Zahl ist gleich Null.");
      }
```

```
      else
      {
        ScreenOut.kPrintln("Die Zahl ist größer Null.");
      }
    }
    else
    {
      ScreenOut.kPrintln("Die Zahl ist kleiner Null.");
    }
```

Läßt man jetzt wieder unter Missachtung der obigen Regel die geschweiften Klammern weg, so ist zunächst nicht mehr klar, welches else Schlüsselwort zu welchem if gehört, denn die Einrückung hat auf die Logik des Programms keine Auswirkung:

```
// Prüfung der eingelesenen Zahl
    ScreenOut.kPrintln();
    if (d >= 0)
      if (d == 0)
        ScreenOut.kPrintln("Die Zahl ist gleich Null.");
      else
        ScreenOut.kPrintln("Die Zahl ist größer Null.");
    else
      ScreenOut.kPrintln("Die Zahl ist kleiner Null.");
```

Eine solche Häufung von if und else Schlüsselwörtern wird vom Compiler gemäß folgender Regel verarbeitet:

Ein else Schlüsselwort wird stets an das im Programmcode nächste oberhalb stehende if angehängt.

Eine Schachtelung von if-Anweisungen der Form

```
if (logischeBedingung1) if (logischeBedingung2) Anweisung1 else Anweisung2
```

wird also vom Compiler wie folgt aufgelöst:

```
if (logischeBedingung1)
{
  if (logischeBedingung2)
  {
    Anweisung1
  }
  else
  {
    Anweisung2
  }
}
```

Diese Logik unterscheidet sich damit von folgender Logik, die in dieser Art nur mit Hilfe von geschweiften Klammern codiert werden kann:

```
if (logischeBedingung1)
{
  if (logischeBedingung2)
  {
    Anweisung1
  }
}
else
{
  Anweisung2
}
```

Im letzten Fall wird der `else`-Zweig ausgeführt, wenn *`logischeBedingung1`* falsch ist, im ersten Fall wird der else-Zweig dagegen ausgeführt, wenn *`logischeBedingung1`* wahr und *`logischeBedingung2`* falsch ist!

2.5.2.6 Laufzeitfehler wegen fehlerhafter if-Anweisung: irrtümliche Verwendung eine Zuweisung als logische Bedingung

Ein häufig auftretender Fehler bei der Formulierung von if-Anweisungen besteht darin, dass

statt einer logischen Bedingung in Form eines Vergleichs eine Zuweisung

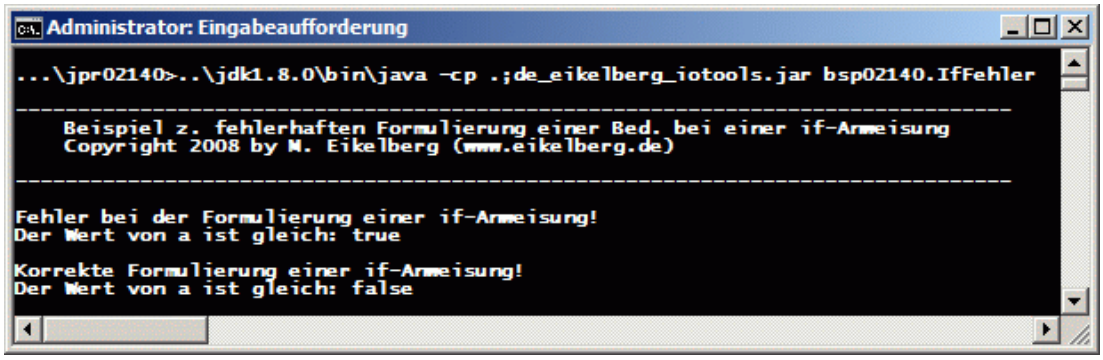
verwendet wird, d.h. es wird bei der Formulierung der logischen Bedingung in Form einer Vergleichsoperation

statt des Vergleichsoperators ein Zuweisungsoperator

codiert - also schlichtweg ein Gleichheitszeichen zu wenig codiert. Der Java-Compiler findet diesen Fehler stets für den Fall, dass die ungewollte Zuweisung nicht zwischen boolean Größen geschieht. Das folgende Beispiel bsp02140.IfFehler zeigt, dass der Compiler jedoch bei boolean Größen nicht stets den Fehler findet:

```
// Fehlerhafte if-Anweisung
    ScreenOut.kPrintln();
    boolean a=false;
    if (a=true)
      ScreenOut.kPrintln("Fehler bei der Formulierung einer if-Anweisung!");
    else
      ScreenOut.kPrintln("Korrekte Formulierung einer if-Anweisung!");
    ScreenOut.kPrintln("Der Wert von a ist gleich: "+a);

// Korrekte if-Anweisung
    ScreenOut.kPrintln();
    a=false;
    if (a==true)
      ScreenOut.kPrintln("Fehler bei der Formulierung einer if-Anweisung!");
    else
      ScreenOut.kPrintln("Korrekte Formulierung einer if-Anweisung!");
    ScreenOut.kPrintln("Der Wert von a ist gleich: "+a);
```

Dieser Quellcode wird vom Compiler anstandslos übersetzt. Bei einem Programmlauf erhält man folgendes Ergebnis:



Der hier gezeigte Effekt lässt sich wie folgt erklären:

- Bei der logischen Bedingung zur fehlerhaften if-Anweisung wird der Variablen a zunächst mit dem Zuweisungsoperator "=" (!) der Wert true zugewiesen. Anschließend wird der Wert von a ausgewertet und - da dieser Wert ja nun wahr geworden ist - nicht in den else-Zweig verzweigt.
- Bei der logischen Bedingung zur korrekten if-Anweisung wird der Wert der Variablen a mit dem Wert true verglichen und es erfolgt eine Verarbeitung je nachdem, ob der Vergleich den Wert wahr ergibt oder nicht.

> Ein irrtümliches Weglassen eines Gleichheitszeichens in einer Vergleichsoperation, die als logische Bedingung eingesetzt wird, führt also bei Verwendung des Datentyps boolean zu einem Laufzeitfehler. Bei anderen Datentypen findet der Java-Compiler - anders als einige Compiler anderer Programmiersprachen (z. Bsp. C/C++) - jedoch den Fehler.

Man beachte, dass die "beste" Codierung des obigen Programmstücks darin besteht, überhaupt kein Gleichheitszeichen zu verwenden, denn die Vergleichsoperation ist bei booleschen Größen unnötig:

```
// Optimale if-Anweisung
    ScreenOut.kPrintln();
    a=false;
    if (a)
      ScreenOut.kPrintln("Fehler bei der Formulierung einer if-Anweisung!");
    else
      ScreenOut.kPrintln("Korrekte Formulierung einer if-Anweisung!");
    ScreenOut.kPrintln("Der Wert von a ist gleich: "+a);
```

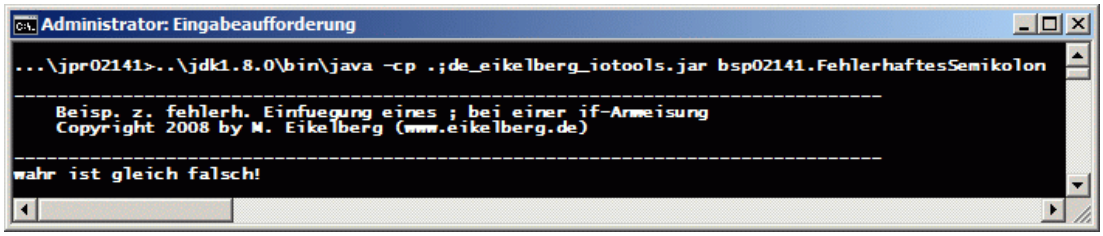
2.5.2.7 Laufzeitfehler wegen irrtümlicher Codierung eines Semikolons

Ein insbesondere bei Programmieranfängern sehr weit verbreiteter Laufzeitfehler entsteht, wenn nach der schließenden Klammer um die logische Bedingung zur if-Anweisung ein Semikolon eingefügt wird. Ein derartiger Fehler kann übrigens auch bei anderen Steuerungselementen zu Laufzeitfehlern führen. Beispiel bsp02141.FehlerhaftesSemikolon zeigt das Problem auf:

```
...

  public static void main(String[] args)
  {
  ...

//  fehlerhafte if-Anweisung
    if (true == false);  // man beachte das Semikolon!!!
```

```
    ScreenOut.kPrintln("wahr ist gleich falsch!");
}
```

Beim Programmlauf liefert das Programm die Ausgabe "wahr ist gleich falsch":



Dieses Verhalten des Programms liegt darin begründet, dass durch das Semikolon direkt nach der schließenden Klammer zur logischen Bedingung der if-Anweisung ein leerer Block codiert wird. Die if-Anweisung bezieht sich dann auf diesen leeren Block, d.h. der leere Block wird je nach Wert der logischen Bedingung ausgeführt oder nicht. Die auf die if-Anweisung folgende cout-Anweisung wird dagegen stets ausgeführt!

2.5.3 switch-Anweisungen

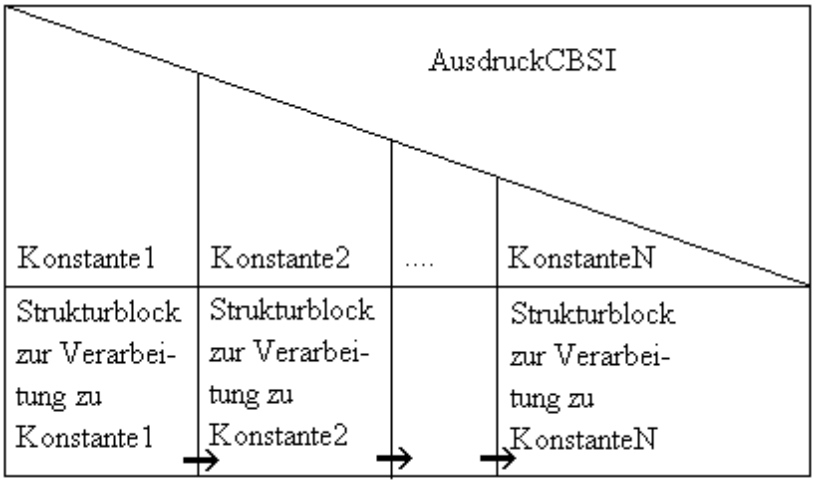
Hinweis: Die in diesem Kapitel folgende Darstellung von Strukturblöcken zu Auswahlanweisungen stellt eine Erweiterung der Definition der Struktogramme dar. Diese Erweiterung ist erforderlich, um die Verarbeitung einer Auswahlanweisung durch die Programmiersprache C bzw. ihre Nachfolgesprachen wie z. Bsp. C++ und Java in einem Struktogramm abbilden zu können. Sie wird im Skript Eikelberg, Grundlagen der Programmierung, Kapitel 1.5 bzw. in **Kapitel 1.5.2 Struktogramme** genauer erklärt.

2.5.3.1 Variante 1 und 2 der Auswahlanweisung

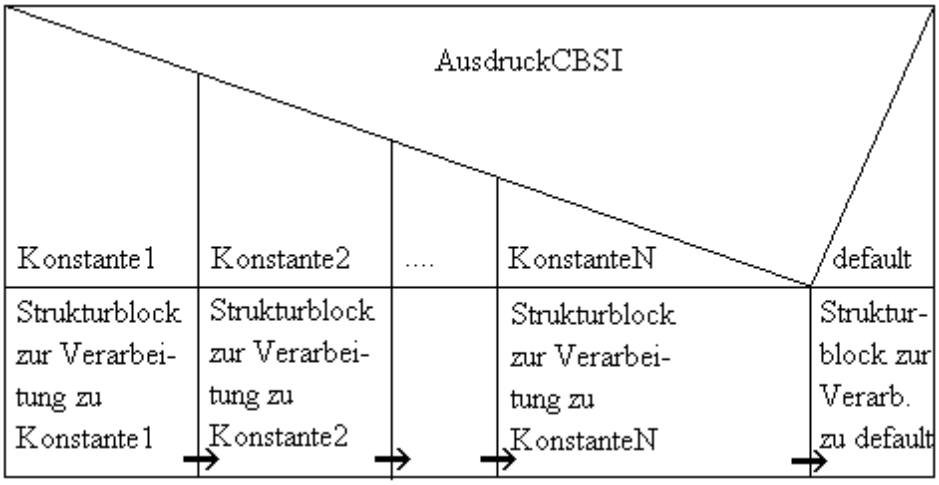
Erfordert die Programmlogik eine Auswahl zwischen zwei Verarbeitungswegen, so wird in der Regel hierfür die if-Anweisung verwendet. Ist jedoch eine Auswahl zwischen mehr als zwei Verarbeitungswegen erforderlich, so ist die **switch-Anweisung** eine gute Alternative zu der ansonsten erforderlichen Schachtelung von if-Anweisungen. Je nach Formulierung des Programmierproblems kann es aber auch sinnvoll sein, eine Auswahl aus zwei Verarbeitungswegen mit Hilfe der switch-Anweisung zu formulieren.

Die switch-Anweisung gibt es in mehreren Varianten, die durch die 4 verschiedenen in Kapitel 1.5.2.4 erklärten Typen der **Auswahlanweisung** dargestellt werden. Zu jeder der 4 Varianten der Auswahlanweisung gibt es eine entsprechende Variante der switch-Anweisung.

Die beiden ersten Typen der Auswahlanweisung sind wie folgt aufgebaut:



Auswahlanweisung Typ 1



Auswahlanweisung Typ 2

Der `AusdruckCBSI` in den beiden obigen Struktogrammen muss stets von Ergebnistyp character, byte, short, oder integer sein, wenn einfache Datentypen verwendet werden. Ab Java 1.7 sind als Referenztypen (vgl. Kapitel 2.6) auch die Datentypen Character, Byte, Short, Integer, String oder ein Aufzähungstyp (enum type) möglich. Da Referenztypen erst später erklärt werden, befassen sich die nachfolgenden Ausführungen hauptsächlich mit einfachen Datentypen als Ergebnistyp für den `AusdruckCBSI` . Man beachte ferner:

Der `AusdruckCBSI` unterscheidet sich von dem Ausdruck `logischerAusdruck` im Struktogramm zur bedingten Anweisung, denn der Ausdruck `logischerAusdruck` muß vom Ergebnistyp boolean sein!

Die Werte für `Konstante1` bis `KonstanteN` müssen Konstanten (vgl. Kapitel 2.3.1.2) von dem Datentyp sein, der Ergebnistyp des Ausdrucks `AusdruckCBSI` ist. Mit anderen Worten müssen die Konstanten `Konstante1` bis `KonstanteN` mögliche Ergebnisse des Ausdrucks `AusdruckCBSI` sein.

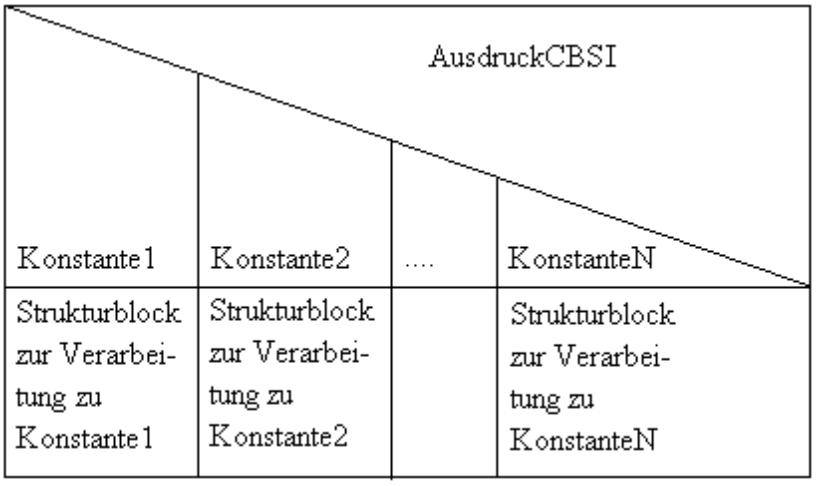
Zur Verarbeitung wird beim Ablauf des Programmstücks zu Typ 1 und 2 der Auswahlanweisung der Ausdruck `AusdruckCBSI` ausgewertet. Diese Auswertung ergibt - wenn man einfache Datentypen verwendet - je nach Ergebnistyp des Ausdrucks einen Wert vom Datentyp character, byte, short, oder integer. Anschließend wird unter den Konstanten `Konstante1` bis `KonstanteN` diejenige gesucht, deren Wert mit dem Ergebnis der Auswertung von `AusdruckCBSI` übereinstimmt. Ist `KonstanteX` die gesuchte Konstante, so setzt die Verarbeitung mit dem Strukturblock zur Verarbeitung zu KonstanteX fort. Nach Verarbeitung dieses Strukturblocks werden alle *rechts* von diesem Block stehenden Strukturblöcke ebenfalls ausgeführt.

Falls eine passende Konstante `KonstanteX` gefunden werden kann, unterscheidet sich die Verarbeitung gemäß Typen 1 und 2 nur dadurch, dass bei Typ 2 der Strukturblock zur Verarbeitung zu default zusätzlich ausgeführt wird. Kann keine passende Konstante `KonstanteX` gefunden werden, so erfolgt bei Typ 1 keine Verarbeitung, bei Typ 2 jedoch die Verarbeitung des Strukturblocks zu default. Man beachte, dass also bei Typ 2 der Strukturblock zu default stets ausgeführt wird.

2.5.3.2 Variante 3 und 4 der Auswahlanweisung

Die in 2.5.3.1 beschriebene Eigenart, dass alle rechts neben dem Strukturbock zur Verarbeitung zu `KonstanteX` stehenden Strukturblöcke ebenfalls ausgeführt werden, ist eine Eigenart der Programmiersprache C. Java hat - wie auch andere Programmiersprachen - von C diese Eigenart geerbt.

Von den Erfindern der Struktogramme, den Herren Nassi und Schneidermann, ist ein solches Verhalten in der Form nicht vorgesehen gewesen. Vielmehr sollte zu jeder Konstante `KonstanteX` genau **nur der "Strukturblock zur Verarbeitung zur KonstanteX" ausgeführt** werden. Für den Fall, dass keine Konstante `KonstanteX` zutrifft, sollte entsprechend **genau nur der "Strukturblock zur Verarbeitung zu default" verarbeitet** werden. Ein solches Verhalten der Auswahlanweisung ist zum Beispiel in der Programmiersprache PASCAL realisiert worden. Nassi und Schneidermann hatten die Struktogramme zur Auswahlanweisung daher auch nur wie zu den folgenden beiden Varianten der Auswahlanweisung dargestellt definiert:



Auswahlanweisung Typ 3

AusdruckCBSI				
Konstante1	Konstante2	....	KonstanteN	default
Strukturblock zur Verarbeitung zu Konstante1	Strukturblock zur Verarbeitung zu Konstante2		Strukturblock zur Verarbeitung zu KonstanteN	Struktur-block zur Verarb. zu default

Auswahlanweisung Typ 4

Bei diesen Diagrammen fehlen die bei Typ 1 und 2 vorhandenen Pfeile vollständig. Dementsprechend soll bei der Verarbeitung jeweils genau nur der Strukturblock zur jeweils passenden Konstante `KonstanteX` oder aber nur der Strukturblock zu `default` ausgeführt werden.

> Die zusätzlichen Pfeile in den Diagrammen zu Typ 1 und 2 wurden vom Autor eingefügt, um Struktogramme zu den entsprechenden Typen der switch-Anweisung in Java zu erhalten. Prinzipiell ist ein Einfügen oder Weglassen eines Pfeiles an jeder Stelle möglich, in der in obigen Diagrammen zu Typ 1 oder 2 ein Pfeil eingefügt ist. Alle entsprechenden Varianten der Auswahlanweisung können in Java Syntax umgesetzt werden. **Mit Typ 3 und Typ 4 der Auswahlanweisung soll die Form der Auswahlanweisung gemeint sein, bei der kein Pfeil im Diagramm vorhanden ist. Wenn mindestens ein Pfeil vorhanden ist, liege dagegen Typ 1 bzw. Typ 2 vor!**

2.5.3.3 Syntax der switch-Anweisungen Typ 1 und 2

Die Java Syntax zu Typ 1 der Auswahlanweisung lautet wie folgt:

```
  switch (AusdruckCBSI )
  {
// Start der Verarbeitung zu Konstante1
    case Konstante1 :
      StrukturblockZuKonstante1
// Start der Verarbeitung zu Konstante2
    case Konstante2 :
      StrukturblockZuKonstante2
// Hier können Abschnitte zu weiteren Konstanten eingefügt werden
// ...
// Start der Verarbeitung zu KonstanteN
    case KonstanteN :
      StrukturblockZuKonstanteN
  }
```

Dabei bezeichnet `StrukturblockZuKonstanteM` den "Strukturblock zur Verarbeitung zu KonstanteM" im Struktogramm für jedes *M* zwischen *1* und *N* . Die in den Programmcode eingefügten Kommentare können natürlich weggelassen werden. Bestehen die Blöcke `StrukturblockZuKonstanteM` aus mehr als einer Anweisung, müssen wie üblich geschweifte Klammern zur Blockbildung verwendet werden.

Das Java Schlüsselwort `case` kennzeichnet also die Stelle, bei der die Verarbeitung zur switch-Anweisung beginnen soll. Solche Schlüsselwörter bezeichnet man in der Informatik auch als **Sprungmarke** beziehungsweise **Label**. Zu dem `case`-Label gehört der auf das Label folgende Bezeichner, hier also der Bezeichner zu `KonstanteM` . Durch das Hinzufügen des Bezeichners wird das Label eindeutig gekennzeichnet. Dies ist notwendig, da Label stets eindeutig sein müssen. Ein `case`-Label zu einem Bezeichner `KonstanteM` darf somit in einer `switch`-Anweisung nur genau einmal codiert werden.

Typ 2 der Auswahlanweisung wird wie folgt in Java Syntax umgesetzt:

```
  switch (AusdruckCBSI )
  {
// Start der Verarbeitung zu Konstante1
    case Konstante1 :
      StrukturblockZuKonstante1
// Start der Verarbeitung zu Konstante2
    case Konstante2 :
```

```
      StrukturblockZuKonstante2
// Hier können Abschnitte zu weiteren Konstanten eingefügt werden
// ...
// Start der Verarbeitung zu KonstanteN
    case KonstanteN :
      StrukturblockZuKonstanteN
// Start der Verarbeitung zu default
    default:
      StrukturblockZuDefault
  }
```

Hier bezeichnet `StrukturblockZuDefault` den "Strukturblock zur Verarbeitung zu default". Man beachte, dass `default` ebenfalls ein Label ist, das jedoch nicht durch einen Bezeichner ergänzt wird. Daher kann es also in einer `switch`-Anweisung genau nur ein default-Label geben. Dies ist sinnvoll, denn die Verarbeitung der `switch`-Anweisung beginnt nach dem Label `default`, wenn die Auswertung von `AusdruckCBSI` *keinen* der Werte von `Konstante1` bis `KonstanteN` ergeben hat.

2.5.3.4 Syntax von Typ 3 und 4 der switch-Anweisung

Typ 3 und 4 der Auswahlanweisung unterscheiden sich von Typ 1 und 2 durch *Weglassen von Pfeilen* im Struktogramm. Diese Weglassen von Pfeilen wird in Java-Syntax wie folgt umgesetzt:

> Wird im Struktogramm zu einer Auswahlanweisung ein Pfeil zwischen zwei Strukturblöcken **weggelassen**, so muß im Programmcode vor dem `case`-Label des Strukturblocks, auf den der weggelassene Pfeil zeigen würde, eine `break`-Anweisung **eingefügt** werden.

Mit anderen Worten: Fehlt der Pfeil, so muss eine `break`-Anweisung eingefügt werden. Die `break`-Anweisung bewirkt dann, dass die Programmausführung am Ende der `switch`-Anweisung fortgesetzt wird.

In den Struktogrammen in Kapitel 2.5.3.2 fehlen sämtliche Pfeile. Daher lautet die Syntax der `switch`-Anweisung zu diesen Struktogrammen wie folgt:

```
// Syntax zu Typ 3
  switch (AusdruckCBSI )
  {
// Start der Verarbeitung zu Konstante1
    case Konstante1 :
    {
      StrukturblockZuKonstante1
      break;
    }
// Start der Verarbeitung zu Konstante2
    case Konstante2 :
    {
      StrukturblockZuKonstante2
      break;
    }
// Hier können Abschnitte zu weiteren Konstanten eingefügt werden
// ...
// Start der Verarbeitung zu KonstanteN
    case KonstanteN :
    {
      StrukturblockZuKonstanteN
      break;
    }
  }
```

Bei der Codierung der Strukturblöcke `StrukturblockZuKonstante1` bis `StrukturblockZuKonstanteN` können die geschweiften Klammern um die Strukturblöcke entfallen. Man beachte auch, dass eine `break`-Anweisung nach dem `StrukturblockZuDefault` sinnlos ist. Die Syntax zur Auswahlanweisung Typ 4 lautet daher wie folgt:

```
// Syntax zu Typ 4
  switch (AusdruckCBSI )
  {
// Start der Verarbeitung zu Konstante1
    case Konstante1 :
    {
      StrukturblockZuKonstante1
      break;
    }
// Start der Verarbeitung zu Konstante2
    case Konstante2 :
```

```
    {
      StrukturblockZuKonstante2
      break;
    }
// Hier können Abschnitte zu weiteren Konstanten eingefügt werden
// ...
// Start der Verarbeitung zu KonstanteN
    case KonstanteN :
    {
      StrukturblockZuKonstanteN
      break;
    }
// Start der Verarbeitung zu default
    default:
      StrukturblockZuDefault
  }
```

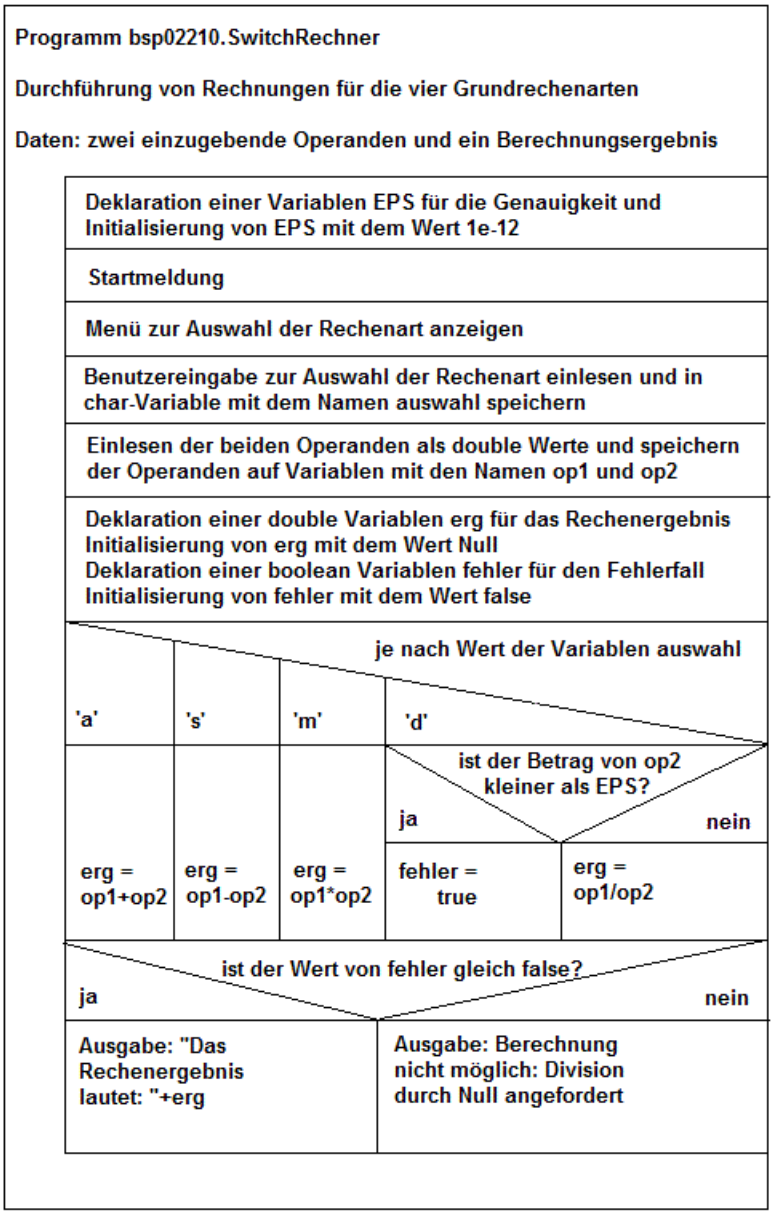
2.5.3.5 Beispielprogramme zur switch-Anweisung

Als Beispiele zu den switch-Anweisungen dienen die Projekte jpr02210 bis jpr02214. Diese Programme führen Berechnungen in den vier Grundrechenarten durch.

2.5.3.5.1 switch-Anweisung mit break und ohne default

Für jedes der fünf Beispielprogramme wird zu Beginn des Programms ein Menü ausgegeben, das die Auswahl der Grundrechenart ermöglicht, die vom Programm durchgeführt werden soll. In den Beispielen jpr02210 bis jpr02213 erfolgt die Menüauswahl durch Eingabe eines Schriftzeichens, im Beispiel jpr02214 durch Eingabe eines Textes aus zwei Schriftzeichen. Die Programme unterscheiden sich anschließend bei der Verarbeitung in Bezug auf die codierte switch-Anweisung.

Das Beispielprogramme bsp02210.SwitchRechner wurde gemäß folgendem Struktogramm codiert:



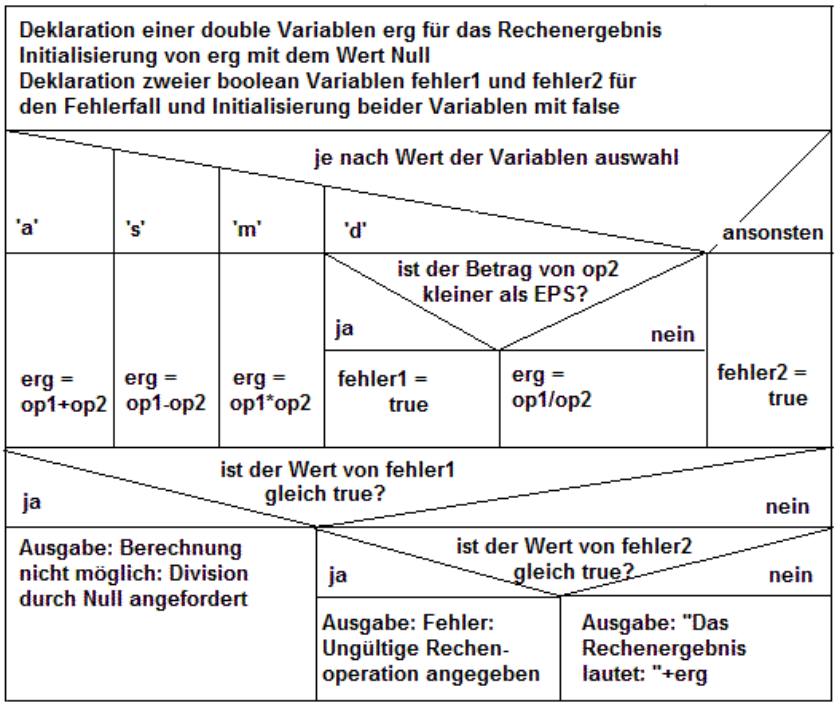
Man sieht, dass das Struktogramm eine switch-Anweisung vom Typ 3 enthält, d.h. der switch-Block entählt keine Pfeile. Dieser Block wird wie folgt in Java Quellcode umgesetzt:

```
switch(auswahl)
{
case 'a':
  erg = op1 + op2;
  break;
case 's':
  erg = op1 - op2;
  break;
case 'm':
  erg = op1 * op2;
  break;
case 'd':
  if (Math.abs(op2) < EPS)
    fehler = true;
  else
    erg = op1 / op2;
  break;
}
```

Man beachte in obigem Quellcode, dass die festen Buchstaben nach dem Schlüsselwort case mit einfachen Hochkommata umgeben sind.

2.5.3.5.2 switch-Anweisung mit break und ohne default

Beispiel jpr02211.SwitchRechner erweitert Beispiel jpr02210 um einen default-Eintrag. Hierdurch wird erreicht, dass das Programm bei der Auswahl einer ungültigen Rechenart eine Fehlermeldung ausgibt. Im nachfolgenden Programmausschnitt wird wieder wie in Beispiel jpr02210 eine Variable `EPS` verwendet, die den Wert `1e-12` hat. Der Teil der Verarbeitung mit der switch-Anweisung ist gemäß dem folgenden Struktogramm-Ausschnitt codiert:



Das Struktogramm ist vom Typ 4. Die switch-Anweisung enthält somit einen Zweig für den Fall, dass keiner der angegebenen Fälle zutrifft. Entsprechend enthält der Quellcode, mit dem der switch-Blcok aus dem obigen Struktogramm umgesetzt wurde, das Schlüsselwort `default`:

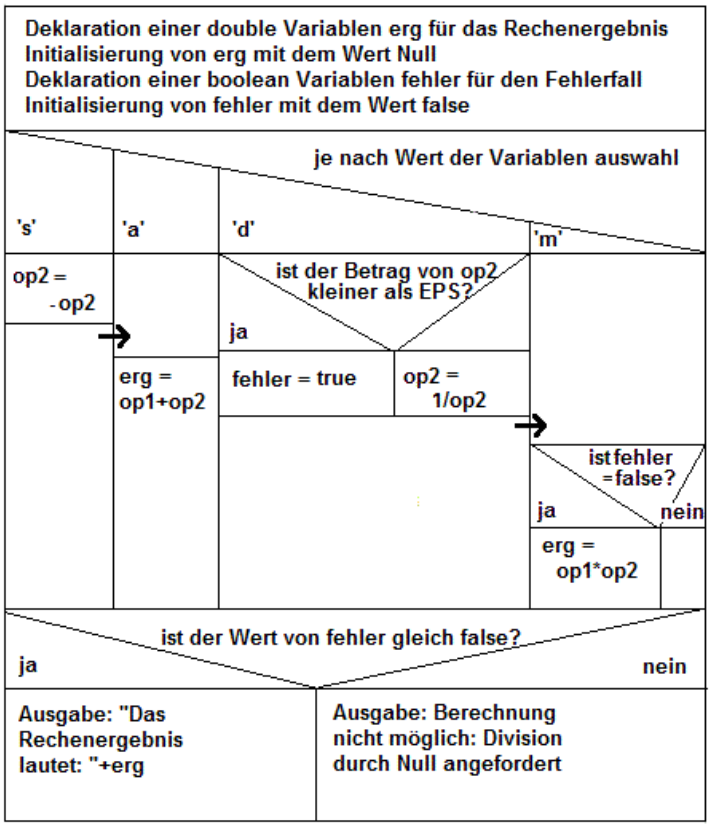
```
switch(auswahl)
{
case 'a':
  erg = op1 + op2;
  break;
case 's':
  erg = op1 - op2;
  break;
case 'm':
  erg = op1 * op2;
  break;
case 'd':
  if (Math.abs(op2) < EPS)
    fehler1 = true;
  else
    erg = op1 / op2;
  break;
default:
  fehler2 = true;
}
```

2.5.3.5.3 switch-Anweisung teilweise ohne break und ohne default

Beispiel jpr02212 zeigt eine Variante des Programms zur Durchführung on Berechnungen in den vier Grundrechenarten, für das zwei case-Blöcke *nicht* mit einer `break`-Anweisung enden. Mathematisch wird die Idee umgesetzt,

- dass die Subtraktion die Addition mit dem inversen Element und
- dass die Division eine Multiplikation mit dem inversen Element

ist. Entsprechend wird der Algorithmus so gestaltet, dass im Falle einer Subtraktion bzw. einer Division zunächst das inverse Element gebildet wird und anschließend mit dem Zweig für die Addition bzw. Multiplikation fortgesetzt wird. In diesem Fall entfällt jeweils die `break`-Anweisung am Ende des case-Blocks:



Das Struktogramm ist vom Typ 3 - stellenweise sind Pfeile zum rechts nebenstehenden Block eingezeichnet. Entsprechend der Regeln

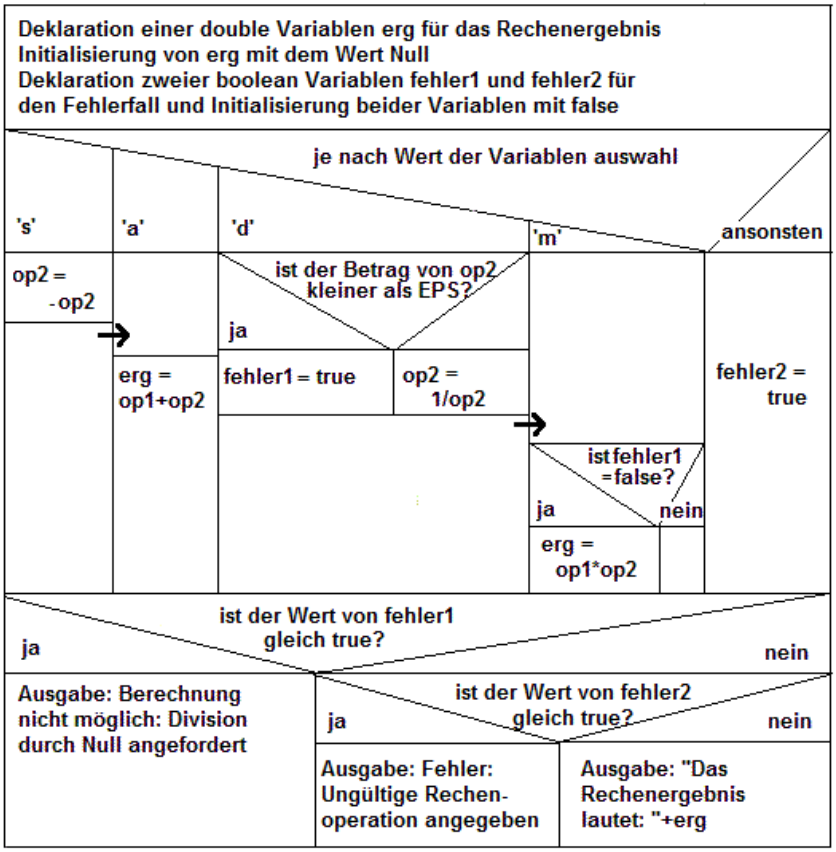
- kein Pfeil => `break` muss codiert werden
- Pfeil vorhanden => `break` darf nicht codiert werden

sind im nachfolgenden Quellcode von bsp02212.SwitchRechner die `break`-Anweisungen genau dann *nicht* vorhanden, wenn ein Pfeil im Struktogramm vorhanden ist:

```
switch(auswahl)
{
case 's':
  op2 = - op2;
case 'a':
  erg = op1 + op2;
  break;
case 'd':
  if (Math.abs(op2) < EPS)
    fehler = true;
  else
    op2 = 1 / op2;
case 'm':
  if (!fehler)
    erg = op1 * op2;
  break;
}
```

2.5.3.5.4 switch-Anweisung teilweise ohne break und mit default

Die vierte Variante des Programms zur Durchführung von Berechnungen in den vier Grundrechenarten kombiniert die Ideen aus 2.5.3.5.2 (default-Anweisung zur Behandlung von fehlerhaften Eingaben) und 2.5.3.5.3 (Berechnung von inversen Elementen zur Durchführung von Subtraktion und Division). Entsprechend sind im Struktogramm stellenweise Pfeile vorhanden und die `switch`-Anweisung endet mit einer `default`-Spalte:



Im Quellcode fehlen entsprechend stets genau die `break`-Anweisungen, wenn ein Pfeil im Struktogramm vorhanden ist. Ferner enthält die `switch`-Anweisung das Schlüsselwort `default`:

```
switch(auswahl)
{
case 's':
  op2 = - op2;
case 'a':
  erg = op1 + op2;
  break;
case 'd':
  if (Math.abs(op2) < EPS)
    fehler1 = true;
  else
    op2 = 1 / op2;
case 'm':
  if (!fehler1)
    erg = op1 * op2;
  break;
default:
  fehler2 = true;
}
```

2.5.3.6 Exkurs: switch-Anweisung unter Verwendung von Zeichenketten

Ein immer wieder geäußerter Wunsch vieler Programmierung ist die Bereitstellung einer Variante der switch-Anweisung, für die der Ausdruck vom Typ Zeichekette ist, anhand derer die Selektion vorgenommen wird. Mit anderen Worten soll als *AusdruckCBSI* eine Zeichenkette (String-Objekt bzw. String-Literal) eingesetzt werden können. String-Objekte und String-Literale werden in Kapitel 2.6.3 erläutert. Insofern stellen die Ausführungen in diesem Abschnitt einen Exkurs dar.

Für die Java-Version 1.6.0 war es noch nicht möglich, Zeichenketten in einer switch-Anweisung zu verwenden. Ab Java 1.7.0 wird diese Möglichkeit jedoch bereitgestellt. Beispiel jpr02214 ist wie alle anderen Beispiele dieses Kurses mit Java 1.8.0 compiliert und lauffähig. Es enthält im Prinzip die gleiche Verarbeitung wie das Beispiel jpr02213, allerdings erfolgt jetzt die Auswahl der Rechenart über eine Zeichenkette (String-Literal):

```
...

// Menü
    ScreenOut.kPrintln("\nWelche Rechnung wollen Sie durchführen?");
    ScreenOut.kPrintln("Addition          [ad]");
```

```
    ScreenOut.kPrintln("Subtraktion        [su]");
    ScreenOut.kPrintln("Multiplikation     [mu]");
    ScreenOut.kPrintln("Division           [di]");
    ScreenOut.kPrint("Geben Sie Ihre Auswahl ein: ");

    String auswahl = KeyboardIn.kReadLine();

...

    switch(auswahl)
    {
    case "su":
      op2 = - op2;
    case "ad":
      erg = op1 + op2;
      break;
    case "di":
      if (Math.abs(op2) < EPS)
        fehler1 = true;
      else
        op2 = 1 / op2;
    case "mu":
      if (!fehler1)
        erg = op1 * op2;
      break;
    default:
      fehler2 = true;
    }
```

2.5.4 while-Schleifen

2.5.4.1 Überblick über die Schleifenkonstruktionen in Java

Die bisherigen Programme dieses Java-Kurses hatten den Nachteil, dass die Programme nach einmaligem Ablauf der programmierten Abläufe und Berechnungen endeten. In der Regel ist dies aber nicht das erwünschte Verhalten. Somit werden Sprachkonstrukte zur Codierung von wiederholten Ausführungen benötigt. Diese Sprachkonstrukte sind in Java vorhanden und können mit Hilfe der **iterierenden Strukturblöcke** (vgl. Kapitel 1.5.2.5) dargestellt werden. Die folgende Tabelle listet die drei in Kapitel 1.5.2.5 genannten iterierenden Strukturblöcke auf sowie die zugehörigen Java Sprachkonstrukte:

Name des Strukturblocks	Name der Schleifenkonstruktion	Name der Schleife in Java	Java Sprachelement
bedingte Ausführung mit Wiederholung	abweisende Schleife	**while**-Schleife	**while** (...) ...;
bedingte Wiederholung	nichtabweisende Schleife	**do**-Schleife	**do** ... **while** (...);
Zählschleife	Zählschleife	**for**-Schleife	**for** (...) ...;

Man sieht, dass das Schlüsselwort `while` sowohl bei der bedingten Wiederholung als auch bei der bedingten Ausführung mit Wiederholung auftritt. Dies liegt daran, dass beide Schleifen eng verwandt sind. Bei beiden Schleifen hängt nämlich die Zahl der Wiederholungen vom Verhalten eines logischen Ausdrucks ab. Bei der for-Schleife wird dagegen die Zahl der Wiederholungen vom Programmierer fest vorgegeben.

2.5.4.2 Das Sprachelement zur while-Schleife

Die while-Schleife wird durch das Struktogramm zur **bedingten Ausführung mit Wiederholung** dargestellt:

Bei der Verarbeitung dieses Strukturblocks wird zunächst *vor einer erstmaligen Ausführung des "Anweisungsblocks"* die Gültigkeit des Ausdrucks "logischerAusdruck" geprüft. Der Rückgabewert des Ausdrucks "logischerAusdruck" muss vom Typ boolean sein, also entweder gleich `true` oder `false`.

Ist der Rückgabewert des logischen Ausdrucks gleich `false`, wird die Abarbeitung der bedingten Ausführung mit Wiederholung sofort beendet, *ohne dass der Anweisungsblock ein einziges Mal abgearbeitet wurde*.

Ist der Rückgabewert dagegen gleich `true`, so wird der Anweisungsblock einmal abgearbeitet. Anschließend wird in diesem Fall der logische Ausdruck erneut geprüft und je nach Ergebnis die Bearbeitung der Schleife beendet oder der Anweisungsblock erneut abgearbeitet, bis die Prüfung des logischen Ausdrucks den Wert `false` ergibt.

Es ist also durchaus möglich, dass man eine sich unendlich wiederholende while-Schleife programmiert. Dieser Effekt kann auch bei den anderen Schleifenkonstruktionen auftreten.

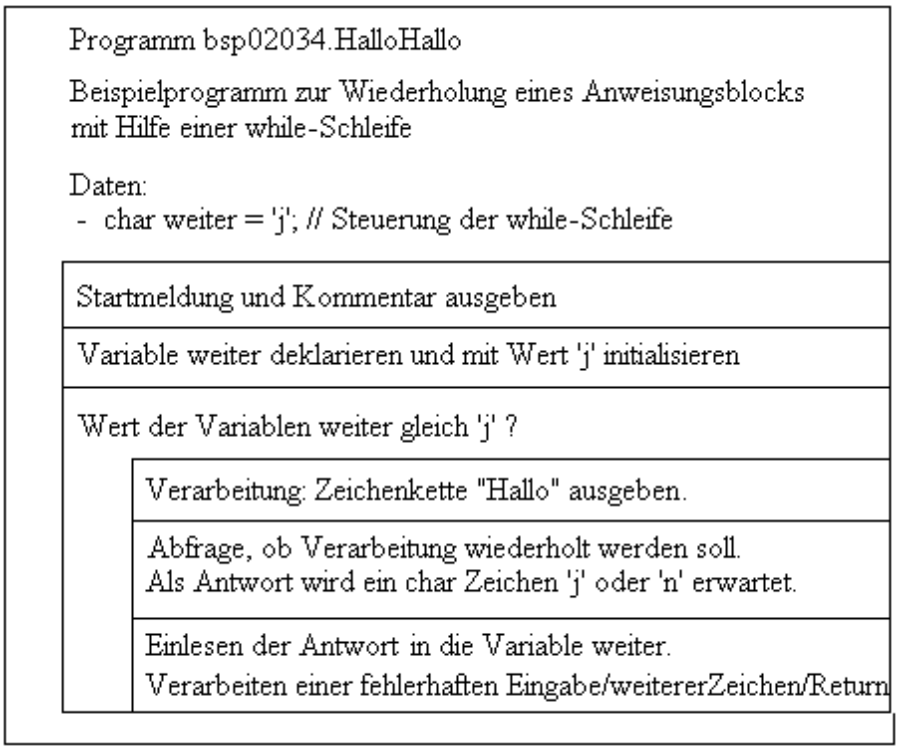
Die bedingte Ausführung mit Wiederholung wird wie folgt in Java Syntax umgesetzt:

```
while (logischerAusdruck)
  Anweisungsblock;
```

Dabei kann der Anweisungsblock aus einer einzigen Anweisung bestehen. In diesem Fall ist eine Blockbildung mit geschweiften Klammern nicht erforderlich. Besteht der Anweisungsblock dagegen aus mehreren Anweisungen, so muss eine Blockbildung mit geschweiften Klammern erfolgen, wobei das in obiger Syntax vorhandene schließende Semikolon dann fehlen darf.

2.5.4.3 Beispielprogramm zur while-Schleife

Ein erstes Beispielprogramm zur while-Schleife ist das Programm bsp02034.HalloHallo. Dieses Programm gibt des Text "Hallo" auf dem Bildschirm aus und wiederholt diese Ausgabe bei Bedarf. Es wurde ausgehend von folgendem Struktogramm codiert:



Man sieht deutlich die mit dem Text "Wert der Variablen weiter gleich 'j'" beginnende bedingte Ausführung mit Wiederholung. Sie kann wie folgt in Java Programmcode umgesetzt werden:

```
// while-Schleife
    char weiter = 'j';
    while (weiter == 'j')
    {

// Verarbeitung in der while-Schleife / Anweisungsblock
      ScreenOut.kPrintln();
      ScreenOut.kPrintln("Hallo");
      ScreenOut.kPrintln();

// Abfrage, ob Verarbeitung wiederholt werden soll
      ScreenOut.kPrint("Verarbeitung wiederholen? [j/n] ");
      weiter = KeyboardIn.kReadChar();  // Einlesevorgang 1
      if (weiter == '\r')
// kein Zeichen vor Return eingegeben - Newline Character überlesen:
        KeyboardIn.kReadChar();         // Einlesevorgang 2a
      else
// ein oder mehr Zeichen eingegeben - Rest der Eingabe incl. Return überlesen
        KeyboardIn.kReadLine();         // Einlesevorgang 2b
    }
```

Die Steuerung, ob die Verarbeitung in der while-Schleife wiederholt werden soll, geschieht unter Verwendung der Variablen `weiter`. Diese Variable ist vom Datentyp char. Man beachte daher die bereits in Kapitel 2.3.5.4 angesprochene Besonderheit beim Einlesen von char Zeichen: Beim Einlesen eines char Zeichens mit der Methode kReadChar()

- wird stets genau ein Zeichen eingelesen,
- ein Zeilenwechsel ergibt zwei Schriftzeichen (Zeichen Return und Newline), die jeweils einzeln eingelesen werden müssen,
- Dateiende ergibt ein Leerzeichen sowie einen Rückgabewert 4 bei Aufruf der Methode dGetStatus() und
- Betätigung der Taste F6 bei der Tastatureingabe ergibt ebenfalls ein Leerzeichen sowie einen Rückgabewert 4 bei Aufruf der Methode kGetStatus().

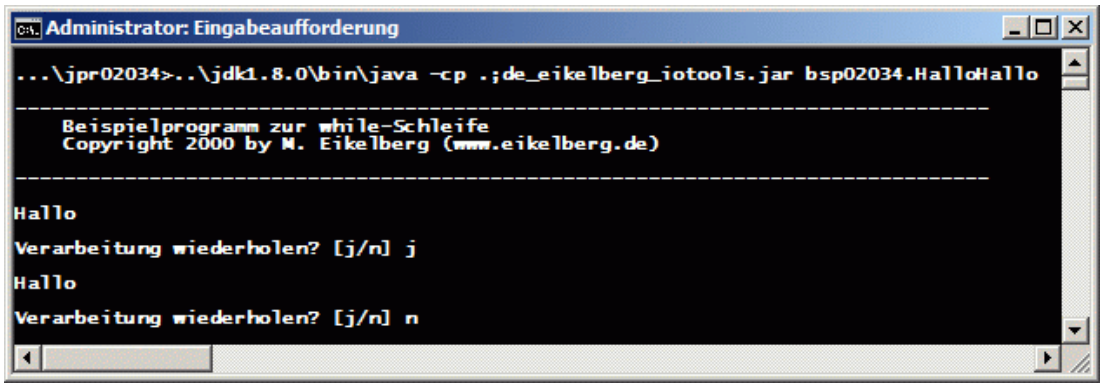
Der obige Programmcode berücksichtigt nicht die Möglichkeit, dass der Nutzer die Taste F6 betätigt statt eine geeignete Antwort auf die Frage "Verarbeitung wiederholen ..." zu geben. Ein solches Verhalten ist auch nicht zu erwarten. Es kann allerdings passieren, dass der Nutzer nicht genau ein Schriftzeichen eingibt und anschließend die Return-Taste betätigt. Vielmehr können folgende Fälle auftreten:

1. *Der Nutzer gibt kein Zeichen ein und betätigt direkt Return.*
   Die Betätigung der Return-Taste bewirkt, dass die zwei nicht druckbaren Steuerzeichen Return (char Wert `'\r'`) und Newline (char Wert `'\n'`) an das Programm weitergegeben werden. Durch den erstmaligen Aufruf von kReadChar() ist in diesem Fall bereits das Zeichen Return eingelesen worden. Es muss also im Einlesevorgang 2a noch das Zeichen Newline verarbeitet werden.
2. *Der Nutzer gibt genau ein Schriftzeichen ein und betätigt danach die Return-Taste.*
   Dies ist das eigentlich zu erwartende Verhalten. Das eingegebene Schriftzeichen wird bei der Verarbeitung des Einlesevorgangs 1

eingelesen und ausgewertet. Es folgen von der Tastatur die Zeichen Return und Newline. Diese beiden Zeichen werden bei der Verarbeitung des Einlesevorgangs 2b von der Methode kReadLine() verarbeitet.

3. *Der Nutzer gibt mehr als ein Schriftzeichen ein und betätigt danach die Return-Taste.*
   In diesem Fall wird das erste Schriftzeichen beim Einlesevorgang 1 gelesen und ausgewertet. Die folgenden Schriftzeichen und das abschließende Return werden bei der Verarbeitung des Einlesevorgangs 2b von der Methode kReadLine() verarbeitet.

Der Aufruf von bsp02034.HalloHallo führt zu folgender Bildschirmausgabe:



2.5.4.4 Codierung von unendlichen while-Schleifen (Loops)

Bei der Erstellung von Programmen passiert es immer wieder, dass Programme in Schleifen geraten, deren logische Bedingung niemals falsch wird. In diesem Fall endet das Programm nicht von selbst und muss vom Nutzer abgebrochen werden. Vielmehr dreht sich das Programm quasi im Kreis. Man sagt daher auch, dass das Programm in einen **Loop** (ausgesprochen 'luup') geraten ist bzw. dass das Programm **loopt** (ausgesprochen 'luupt'). Sehr selten ist ein solches Verhalten des Programms gewünscht, in der Regel jedoch nicht.

Ein Beispiel für ein Loop ist im Programm bsp02035.HalloLoop zu finden. bsp02035.HalloLoop ist aus dem Programm bsp02034.HalloHallo durch Änderung der logischen Bedingung der while-Schleife entstanden: Die Codezeile

```
while (weiter == 'j')
```

in bsp02034.HalloHallo wurde dabei in bsp02035.HalloLoop durch folgende Zeile ersetzt:

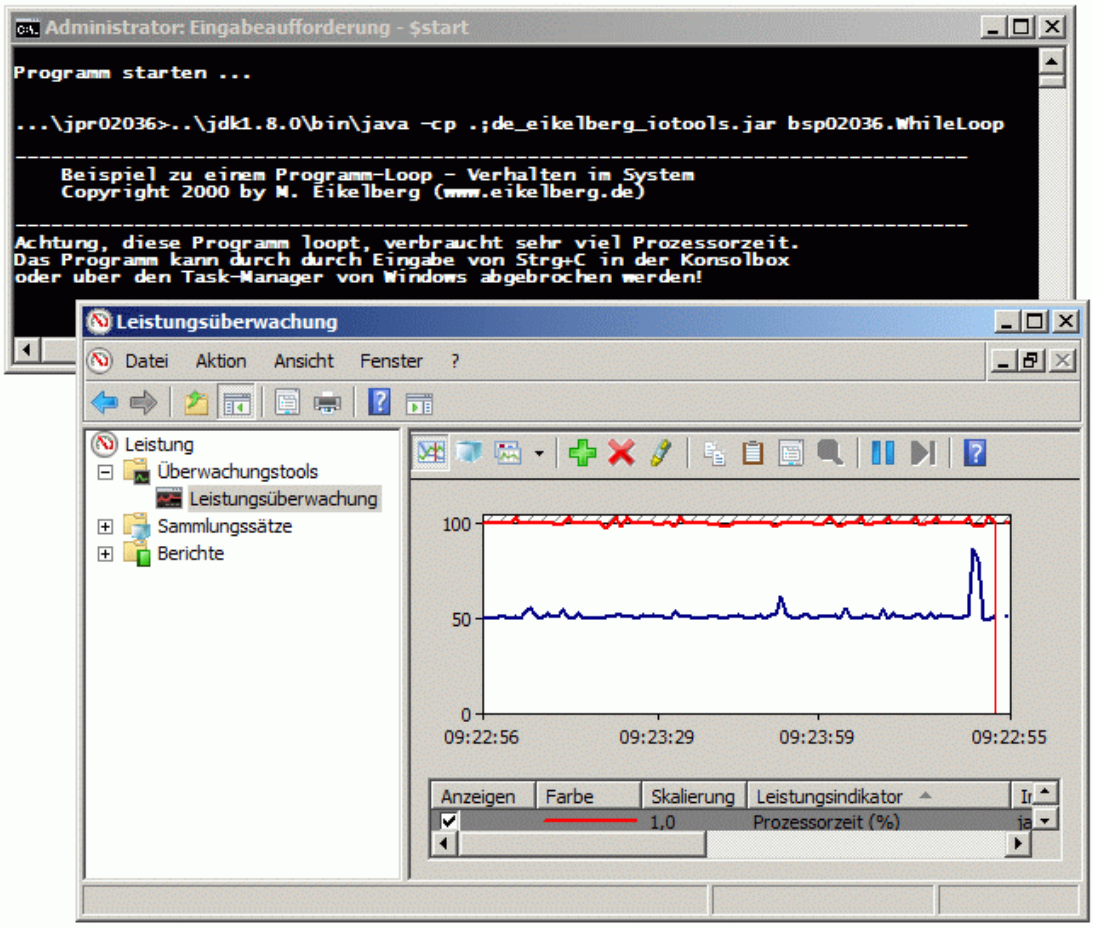
```
while (weiter != 'n' | weiter != 'N')
```

Der Fehler in dieser Zeile besteht in der falschen Auswahl des logischen Operators, mit dem die beiden Vergleichsoperationen verknüpft wurden: Statt des logischen Oder hätte ein logisches Und eingefügt werden müssen. So aber ist die Bedingung immer wahr, denn jedes Zeichen ist entweder von 'n' oder 'N' verschieden. Damit ist die Schleifenbedingung immer wahr und es besteht außer einem Programmabbruch keine Möglichkeit, das Programm zu verlassen. Ein solcher Programmabbruch geschieht durch Eingabe von Strg+C in der Konsolbox bzw. unter Windows NT auch mit Hilfe des Task-Managers. Besonders spektakulär sind im übrigen die Programmloops, bei denen das Programm ununterbrochen einen sehr großen Anteil der Prozessorzeit verbraucht.

Ein Beispiel für einen Programmloop mit hohem Verbrauch an Prozessorzeit ist das Programm bsp02036.WhileLoop. Die in diesem Programm enthaltene while-Schleife besteht aus folgendem Programmcode:

```
// Verarbeitung
    double d;
    while (true)
    {
      d = Math.random();
      Math.atan(d);
    }
```

Da das Programm während des Loops keine Daten einliest oder ausgibt und auch sonst mit der Außenwelt keinen Kontakt aufnimmt, kann es die volle Prozessorleistung eines Prozessors bzw. eines Prozessor-Kerns ausnutzen. Daher führt bsp02036.WhileLoop auf einem Einprozessorrechner zu einer 100%igen Auslastung eines Prozessor(kerns), wie der folgende Ausschnitt eines Bildschirmausdrucks zeigt :



Dieser Bildschirmausdruck wurde im Jahr 2016 mit Hilfe des Programms Windows-Programms perfmon.exe auf einem Windows 7-Rechner des Autors angefertigt (1 CPU Intel Core 2 duo aus 2 Kernen mit 2,33 GHz). Die durchgehende blaue Linie in Höhe von etwa 50% im Systemmonitor zeigt die gesamte Rechnerauslastung des mit 2 Kernen ausgestatteten Systems. Die rote Linie im Diagramm zeigt den CPU-Bedarf der in der Konsolbox unter Kontrolle der Java Virtuellen Maschine (java) laufenden Anwendung bsp0236.WhileLoop - das Programm lastet einen der 2 Kerne zu 100% aus.

2.5.4.5 Laufzeitfehler wegen irrtümlicher Codierung eines Semikolons direkt nach der logischen Bedingung einer while-Schleife

Ein häufig auftretender Fehler, der vom Compiler nicht gefunden wird und der zum Programm-Loop führt, ist das irrtümliche Codieren eines Semikolons direkt nach der logischen Bedingung einer while-Schleife. Beispielprogramm jpr02235 enthält diesen Fehler:

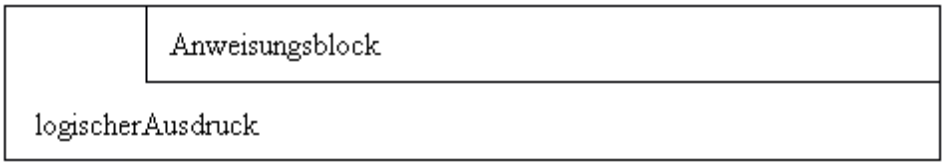
```
package bsp02235;

...

public class HalloLoop
{

  public static void main(String[] args)
  {
    ...

// while-Schleife
    char weiter = 'j';
    while (weiter != 'n' & weiter != 'N');  // Fehler - hier darf kein Semikolon stehen!
    {

// Verarbeitung in der while-Schleife / Anweisungsblock
        ...
    }
  }
}
```

Das am Ende der Zeile mit dem Schlüsselwort while eingefügte Semikolon bewirkt, dass die while-Schleife sich nicht mehr auf den nachfolgenden Block bezieht und diesen Block wiederholt, sondern dass die while-Schleife den leeren Block wiederholt, der sich zwischen der schließenden runden Klammer um die logische Bedingung und dem nachfolgenden Semikolon befindet. In diesem Block befindet sich keine Verarbeitung, also verändert sich im Beispiel der Inhalt der Variablen `weiter` nicht und das Programm gerät in einen Loop.

2.5.5 do-Schleifen

2.5.5.1 Das Sprachelement zur do-Schleife

Die do-Schleife wird durch das Struktogramm zur **bedingten Wiederholung** dargestellt:



Das Diagramm zeigt, dass bei der Verarbeitung der bedingten Wiederholung stets zunächst die Anweisungen des Anweisungsblocks ausgeführt werden. Anschließend wird der Ausdruck "logischerAusdruck" geprüft. Dieser Ausdruck muss einen boolean Wert ergeben. Ist der Wert des Ausdrucks `false`, so wird der Block verlassen und mit dem nachfolgenden Strukturblock fortgesetzt. Ist der Wert von "logischerAusdruck" gleich `true`, so wird die Abarbeitung der bedingten Wiederholung erneut durchgeführt.

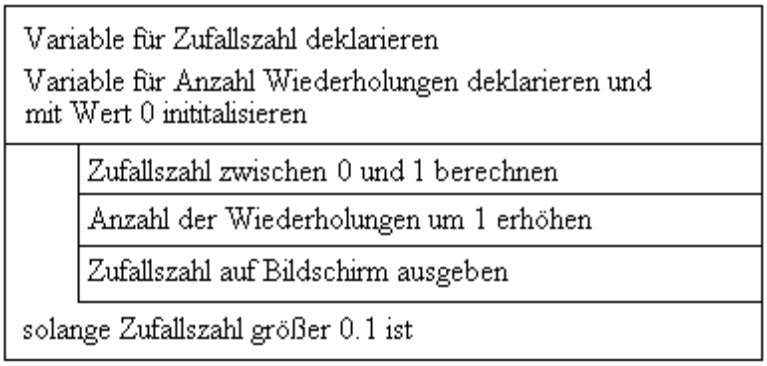
Das Java Sprachelement zur bedingten Wiederholung ist die **do-Schleife**:

```
do
{
  Anweisungsblock
} while (logischerAusdruck);
```

Wie bei jedem Steuerungselement wird der logische Ausdruck stets mit runden Klammern umschlossen. Wenn der "Anweisungsblock" aus mehr als einer Anweisung besteht, muss dieser auch mit geschweiften Klammern umschlossen werden.

2.5.5.2 Beispielprogramm zur do-Schleife

Das Programm bsp02037.Zufallszahlen enthält eine do-Schleife, die sich ohne Eingriff des Nutzers mehrfach wiederholt. Innerhalb der do-Schleife wird eine Zufallszahl zwischen 0.0 und 1.0 berechnet und auf dem Bildschirm ausgegeben. Dieser Vorgang wird solange wiederholt, wie die Zufallszahlen größer als 0.1 sind. Die Verarbeitung wird durch folgenden Strukturblock dargestellt:



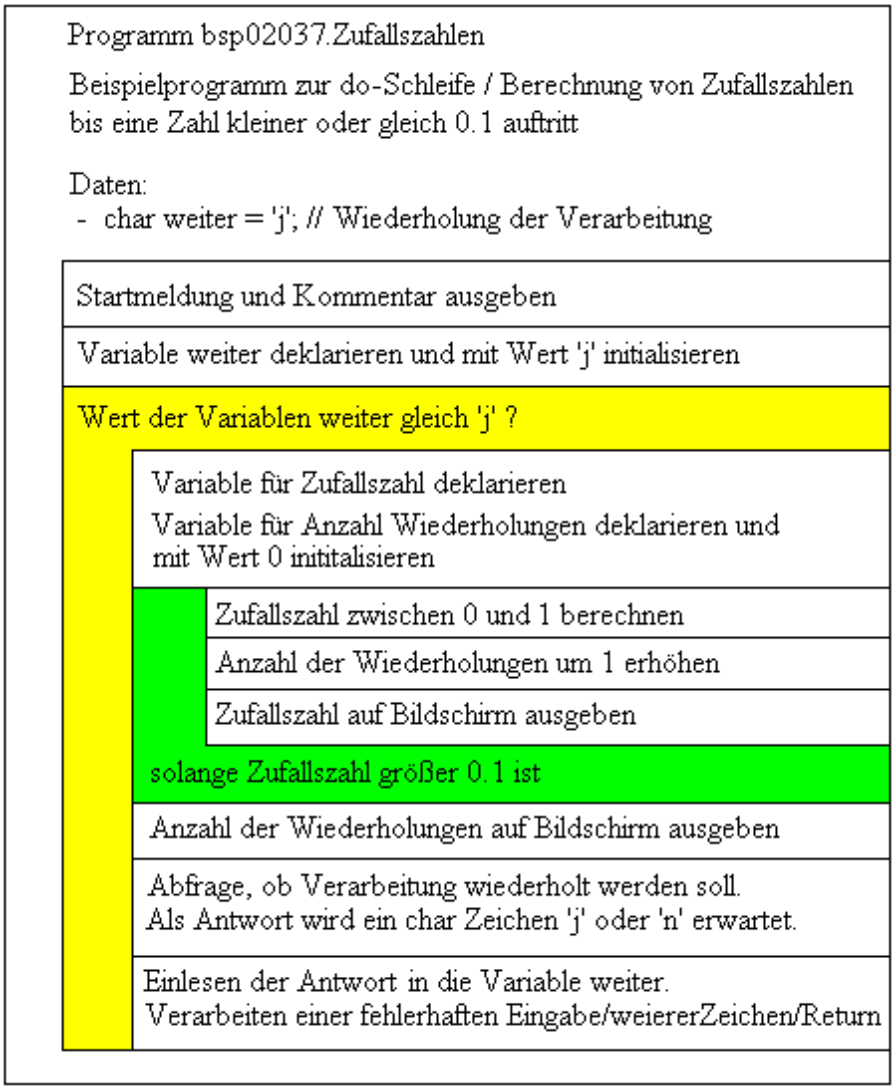
Dieses Struktogramm ist ein Ausschnitt aus dem Modulblock zum Programm bsp02037.Zufallszahlen. Es beginnt mit einem sequentiellen Block, in dem die Variablen deklariert und initialisiert werden. An den sequentiellen Block schliesst sich der Strukturblock zur bedingten Wiederholung an. Die beiden Blöcke in dem obigen Ausschnitt werden wie folgt in Java Syntax umgesetzt:

```
double zahl;
int wiederholungen = 0;
do
{
  zahl = Math.random();
  wiederholungen = wiederholungen + 1;
  ScreenOut.kPrintln(zahl);
} while (zahl > 0.1d);
```

Mit Hilfe der Variablen `wiederholungen` wird die Anzahl der Schleifendurchgänge bis zum Abbruch der do-Schleife gezählt.

Damit die gesamte do-Schleife mehrfach abgearbeitet werden kann, enthält das Programm bsp02037.Zufallszahlen zusätzlich eine while-

Schleife, wie sie auch im Programm bsp02034.HalloHallo verwendet wurde. Insgesamt sieht daher das Struktogramm von bsp02037.Zufallszahlen wie folgt aus:



Zur Verdeutlichung sind die while-Schleife und die do-Schleife im Struktogramm farbig hervorgehoben. Beim Ablauf erzeugt das Programm einen Bildschirmausdruck der folgenden Art, wobei die Anzahl der ausgegebenen Zufallszahlen mindestens gleich 1 ist, ansonsten jedoch zufällig ist. (Bei einer genügend hohen Anzahl von Durchläufen müssten durchschnittlich jeweils 10 Zufallszahlen ausgegeben werden.)

```
Administrator: Eingabeaufforderung - $start
...\jpr02037>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02037.Zufallszahlen
--------------------------------------------------------------------------------
    Beispiel zur do-Schleife
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Das Programm erzeugt so lange Zufallszahlen zwischen 0.0 und 1.0,
bis eine Zahl kleiner oder gleich 0.1 auftritt

0.4771748107672259
0.917729283573515
0.7343613695152 33
0.02418655470299 8464

Insgesamt 4 Wiederholungen.

Verarbeitung wiederholen? [j/n] j

0.8519411962359186
0.27386257366340094
0.915123059391526
0.8050085488403744
0.6969177694712072
0.6821607272816858
0.5233583642818502
0.5576753795766733
0.4730695320486 4366
0.4675526492206654
0.3005639810056496
0.9534726844461784
0.05907331186553 4106

Insgesamt 13 Wiederholungen.

Verarbeitung wiederholen? [j/n] j
```

2.5.5.3 Codierung unendlicher do-Schleifen (Loops)

Auch bei der Codierung von do-Schleifen können ungewollt oder gewollt unendliche Programmschleifen entstehen. Im Prinzip liegt hier der gleiche Sachverhalt vor wie bei den unendlichen while-Schleifen. Daher kann auch bei den do-Schleifen eine einfache Programmänderung wieder zu einem Programm-Loop führen.

Um aus dem Programm bsp02037.Zufallszahl ein loopendes Programm bsp02038.DoLoop zu erhalten, brauchen wir in der Verarbeitung die logische Bedingung der do-Schleife lediglich wie folgt abzuändern. Wir ersetzen die Codezeile

```
} while (zahl > 0.1d);
```

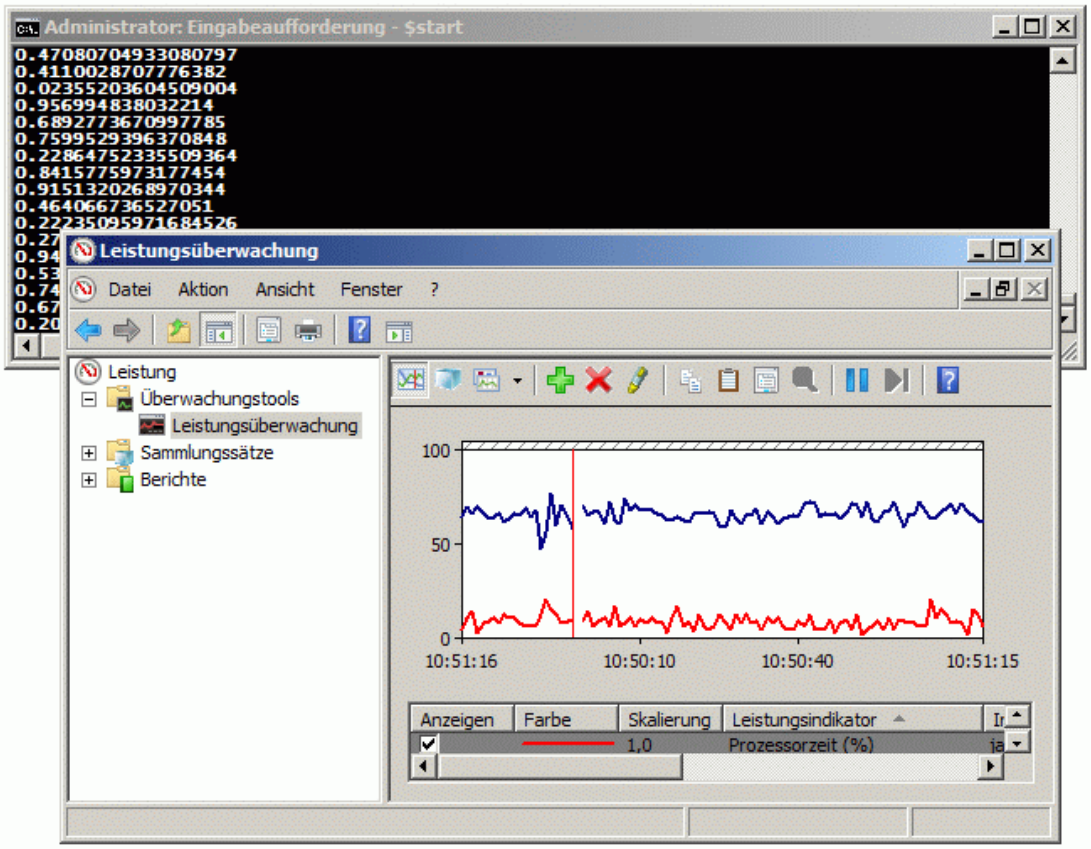
durch die Zeile:

```
} while (zahl >= 0.0d);
```

Die letzte Bedingung ist stets erfüllt, da die Methode `Math.random()` stets Zahlen zwischen 0.0 und 1.0 zurückgibt. Daher wird die do-Schleife niemals beendet.

Wenn das Programm bsp02038.DoLoop läuft, kann man beobachten, dass es zwar ebenfalls zu einer 100prozentigen Rechnerauslastung kommt, dass die Java Virtuelle Maschine anteilig jedoch erheblich weniger CPU-Leistung zugeteilt bekommt als beim Ablauf von bsp02036.WhileLoop. Dies liegt daran, dass innerhalb der do-Schleife eine Ausgabe in die Konsolbox erfolgt und diese Ausgabe einen wesentlichen Teil der CPU-Leistung beansprucht. Besonders groß ist dieser Anteil in dem Fall, dass das Fenster zur Konsolbox im Vordergrund angeordnet ist.

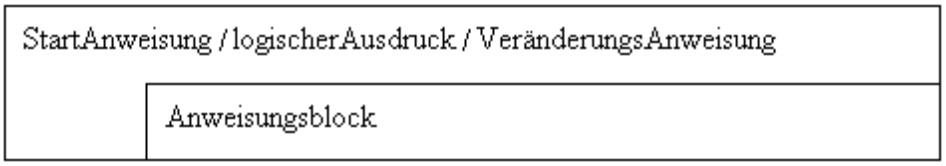
Der folgende Ausschnitt eines Bildschirmausdrucks illustriert das beschriebene Verhalten. Der dickere blaue Strich in Höhe 75 im Diagramm zeigt die 75prozentige CPU-Auslastung des mit 2 Kernen ausgestatteten Prozessors. Zum Zeitpunkt der Messung war neben den Systemprogrammen und dem Systemmonitor lediglich das Programm bsp02038.DoLoop auf dem Rechner des Autors aktiv. Den Anteil von bsp02038.DoLoop an der Prozessorauslastung zeigt die rote Linie im Diagramm, er liegt bei etwa 20% und schwankt stark. Die verbleibende Prozessorleistung wurde im Wesentlichen für die Ausgabe der Zahlen in der Konsolbox benötigt (!):


Administrator: Eingabeaufforderung - $start
0.4708070493308079 7
0.411002870777638 2
0.0235520360450900 4
0.956994838032214
0.689277367099778 5
0.759952939637084 8
0.228647523355093 64
0.841577597317745 4
0.915132026897034 4
0.464066736527051
0.222350959716845 26
Leistungsüberwachung
Datei Aktion Ansicht Fenster ?
Leistung
Überwachungstools
Leistungsüberwachung
Sammlungssätze
Berichte
100
50
0
10:51:16
10:50:10
10:50:40
10:51:15
Anzeigen
Farbe
Skalierung
Leistungsindikator
1,0
Prozessorzeit (%)

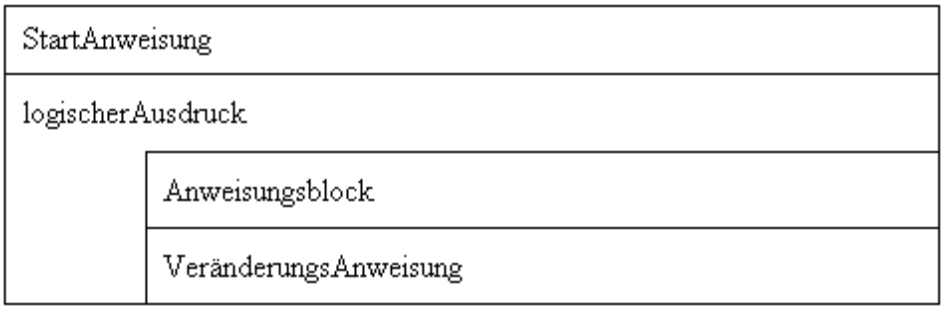
2.5.6 for-Schleifen

2.5.6.1 Das Sprachelement zur for-Schleife

Die for-Schleife ist das Java Sprachelement zur Codierung einer Zählschleife:



Das Struktogramm zur for-Schleife ist also ähnlich zum Struktogramm der while-Schleife. Der Unterschied zwischen for-Schleife und while-Schleife besteht lediglich in der syntaktischen Form. Je nach Situation ist die syntaktische Form der for-Schleife oder der while-Schleife jeweils günstiger. Im Prinzip besteht in Bezug auf die Verarbeitung zwischen der for-Schleife und while-Schleife jedoch kein Unterschied. Eine for-Schleife kann also stets wie folgt in eine while-Schleife umgewandelt werden:



Die "**Startanweisung**" ist eine Anweisung oder ein Anweisungsblock, in dem die Schritte codiert werden sollten, die einmalig vor dem Abarbeiten der Zählschleife durchgeführt werden sollen. Im einfachsten Fall geschieht an dieser Stelle die Deklaration und Initialisierung einer Variablen (Zählvariablen), die die Anzahl der Durchläufe zählt.

Die "**VeränderungsAnweisung**" wird nach jedem Durchlauf der Anweisungen des "Anweisungsblocks" durchgeführt. Durch die "VeränderungsAnweisung" sollte in der Regel die Gültigkeit der logischen Bedingung "logischerAusdruck" verändert werden, so dass "logischerAusdruck" irgendwann den Wert `false` erhält. In dem einfachen Fall, dass die for-Schleife mit einer Zählvariablen formuliert wird, geschieht in der "VeränderungsAnweisung" die Veränderung des Wertes der Zählvariablen.

"**logischerAusdruck**" muss ein logischer Ausdruck sein, dessen Gültigkeit jeweils *vor* der Ausführung des "Anweisungsblocks" geprüft wird. Der "Anweisungsblock" wird genau dann ausgeführt, wenn die Auswertung des Ausdrucks "logischerAusdruck" den Wert `true` ergibt. Ergibt sich der Wert `false`, so wird die Ausführung der for-Schleife beendet. "logischerAusdruck" nimmt also genau die Funktion des Ausdrucks "logischerAusdruck" in der while-Schleife ein. Wenn die for-Schleife mit einer Zählvariablen gesteuert wird, so besteht "logischerAusdruck" aus einem Ausdruck, der den Wert der Zählvariablen prüft.

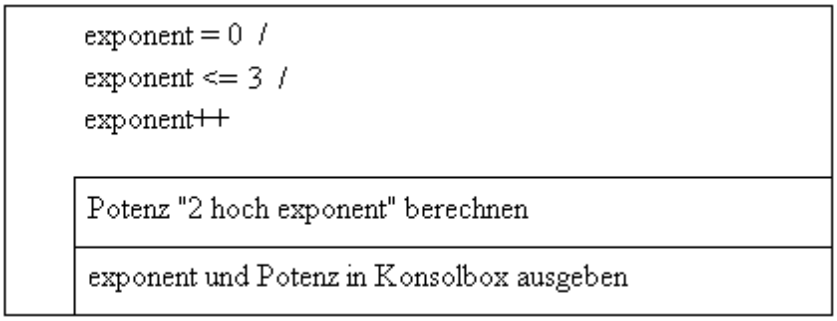
Die for-Schleife hat in Java folgende Syntax:

```
for (StartAnweisung ; logischerAusdruck ; VeränderungsAnweisung)
  Anweisungsblock
```

"StartAnweisung" und "VeränderungsAnweisung" bestehen in der Regel jeweils aus einer einzigen Anweisung. An diesen Stellen kann jedoch auch ein Anweisungsblock stehen, dessen einzelne Anweisungen bei der Codierung ausnahmsweise mit einem Komma statt mit einem Semikolon voneinander getrennt werden. Die Syntax ist hier allerdings etwas eingeschränkt, einige Sprachkonstruktionen können hier nicht eingefügt werden. Der Ausweg an dieser Stelle ist die gleichzeitig auch elegante Möglichkeit, eine aus mehreren Anweisungen bestehende "StartAnweisung" oder "VeränderungsAnweisung" in einer Methode auszulagern.

2.5.6.2 for-Schleifen mit fest vorgegebener Anzahl von Wiederholungen

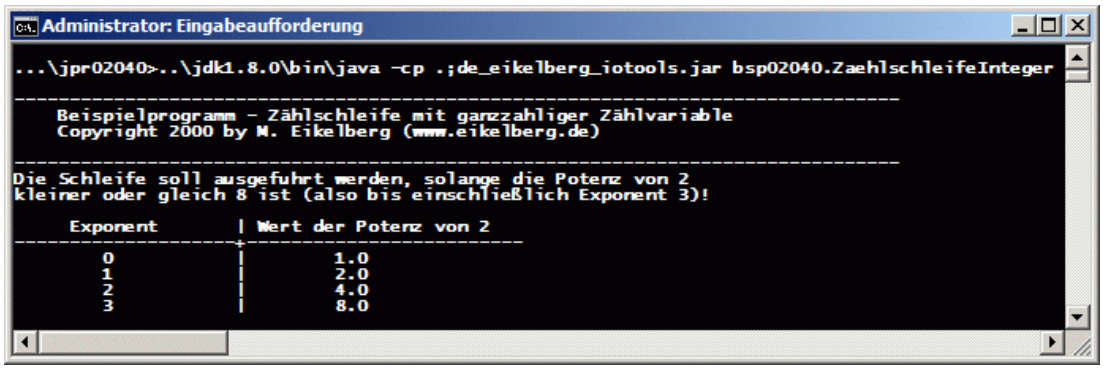
In vielen Situationen ist es erforderlich, dass eine Verarbeitung eine bestimmte feste Anzahl oft durchlaufen werden muss. Für diese Aufgabe ist die for-Schleife das geeignete Sprachelement. Beispiel hierzu ist das Programm bsp02040.ZaehlschleifeInteger, das die Potenzen von 2 für die Exponenten von 0 bis 3 berechnet. Der Strukturblock zu dieser Schleife ist wie folgt aufgebaut:



Dieses Struktogramm kann wie folgt in Java Quellcode umgesetzt werden:

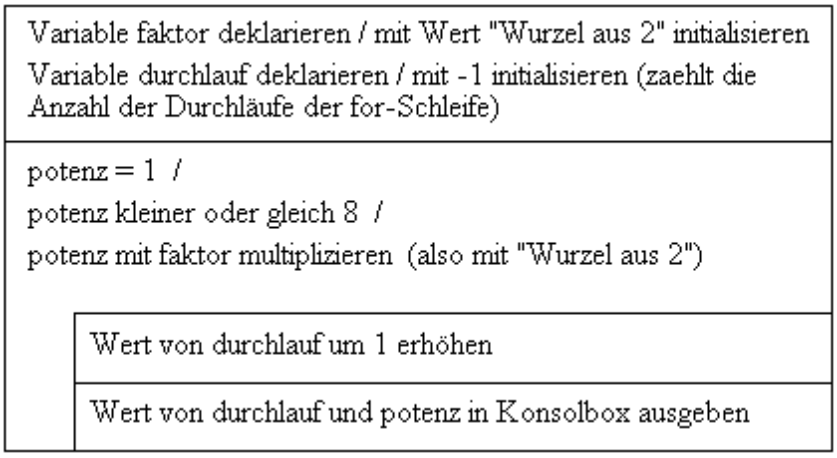
```
double potenz;
for (int exponent=0; exponent <= 3; exponent++)
{
  potenz = Math.pow(2.0D,(double)exponent);
  ScreenOut.kPrintln(
    "        "+exponent+"            |        "+potenz);
}
```

Wie erwartet rechnet das Programm bsp02040.ZaehlschleifeInteger die Potenzen zu den Exponenten 0 bis 3 aus:

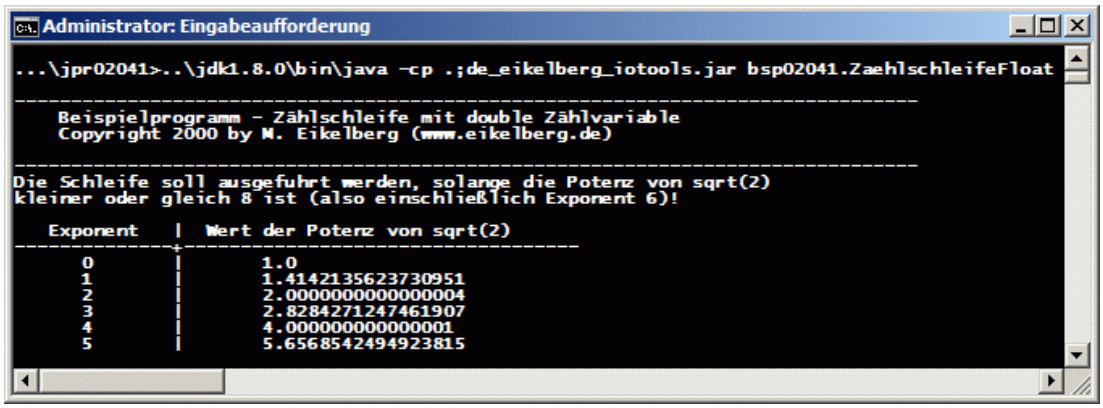


2.5.6.3 for-Schleifen mit Gleitpunkt-Zählvariablen

Die bereits in Kapitel 2.3.2.5 angesprochene Rechenungenauigkeit bei Gleitpunktarithmetik kann zu Verarbeitungsfehlern führen, wenn als Zählvariable in einer for-Schleife eine Gleitpunktzahl gewählt wird und wenn dabei die Abbruchbedingung "logischerAusdruck" ungeschickt gewählt wird. Beispiel für eine entsprechende Situation ist das Beispielprogramm bsp02041.ZaehlschleifeFloat. In diesem Programm ist eine for-Schleife enthalten, die gemäß folgendem Struktogramm programmiert worden ist:



Nach den Regeln der Mathematik wäre die Ausgabe für die Werte der Variablen `potenz` von 1 bis einschließlich 8 zu erwarten. Der Rechenfehler bei Gleitpunktarithmetik führt jedoch dazu, dass vor dem letzten geplanten Durchlauf der for-Schleife der Wert der Variablen `potenz` etwas größer als 8 wird und damit der letzte Durchlauf unerwarteterweise entfällt:



Die Zählschleife in bsp02041.ZaehlschleifeFloat wurde wie folgt codiert:

```
double faktor = Math.sqrt(2);
int durchlauf = -1;
for (double potenz=1; potenz <= 8; potenz *= faktor)
{
  durchlauf = durchlauf + 1;
  ScreenOut.kPrintln(
    "      "+durchlauf+"       |       "+potenz);
}
```

2.5.6.4 for-Schleifen mit Funktionsaufruf für die Prüfung, ob ein erneuter Durchlauf erfolgen soll

Wegen der prinzipiellen Ähnlichkeit der while- und for-Schleife wird als weiteres Beispielprogramm zur for-Schleife das Programm bsp02039.HalloHallo angeführt. bsp02039.HalloHallo ist lediglich eine Umformulierung des Beispielprogramms zur while-Schleife bsp02034.HalloHallo. Nach Ersetzen der while-Schleife in bsp02034.HalloHallo durch eine for-Schleife entsteht das folgende Programm bsp02039.HalloHallo:

Programm bsp02039.HalloHallo

Beispielprogramm zur Wiederholung eines Anweisungsblocks
mit Hilfe einer for -Schleife

Daten:
- char weiter = 'j'; // Steuerung der for -Schleife

Startmeldung und Kommentar ausgeben

Variable weiter deklarieren und mit Wert 'j' initialisieren /
Wert der Variablen weiter gleich 'j' ? /
Frage mit der Methode weiterPruefung() ab, ob die Verarbeitung
wiederholt werden soll - falls ja, liefert weiterPrüfung den Wert 'j'

Zeichenkette "Hallo" ausgeben.

Methode weiterPruefung()

Methode liest ein char Zeichen ein, verarbeitet den Rest der Eingabe
und gibt das char Zeichen an die rufende Methode zurück

Abfrage, ob Verarbeitung wiederholt werden soll.
Als Antwort wird ein char Zeichen 'j' oder 'n' erwartet.

Einlesen der Antwort in die Variable weiter.
Verarbeiten einer fehlerhaften Eingabe/weitererZeichen/Return

Weitergabe des eingelesenen Zeichens an das rufende Prog.

Die syntaktischen Beschränkungen bei der Formulierung der "StartAnweisung" und der "VeränderungsAnweisung" haben bei der Entwicklung des Programms dazu geführt, dass die Abfrage, ob die Verarbeitung wiederholt werden soll, in die Methode weiterPrüfung() ausgelagert wurde.

Die Methode weiterPrüfung() besteht aus folgendem Java Code:

```
  private static char weiterPruefung()
  {
    char weiter = 'j';
// Abfrage, ob Verarbeitung wiederholt werden soll
    ScreenOut.kPrint("Verarbeitung wiederholen? [j/n] ");
    weiter = KeyboardIn.kReadChar();
    if (weiter == '\r')
// kein Zeichen vor Return eingegeben - Newline Character überlesen:
      KeyboardIn.kReadChar();
    else
// ein oder mehr Zeichen eingegeben - Rest der Eingabe incl. Return überlesen
      KeyboardIn.kReadLine();
    return weiter;
  }
```

Methoden werden in Kapitel 2.7 genauer erklärt. Ein Aufruf der Form

```
  char wtr = weiterPuefung()
```

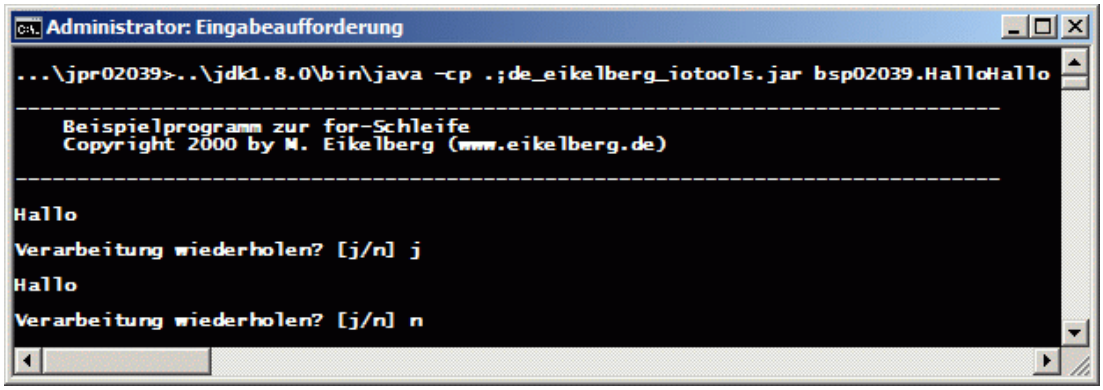
führt dazu, dass alle Anweisungen der Methode weiterPruefung() abgearbeitet werden und dass das in der Variablen `weiter` gespeicherte Schriftzeichen in der Variablen `wtr` gespeichert wird. Wegen der Regeln der Sichtbarkeit kann die Variable `wtr` auch `weiter` heissen. Die entsprechende Anweisung lautet dann

```
  char weiter = weiterPuefung()
```

Das Programm bsp02039.HalloHallo enthält jetzt die durch das obige Struktogramm beschriebene for-Schleife, die aus folgendem Java Quellcode besteht:

```
// for-Schleife
    for (char weiter = 'j'; weiter == 'j'; weiter=weiterPruefung())
    {
// Anweisungsblock in der for-Schleife
      ScreenOut.kPrintln();
      ScreenOut.kPrintln("Hallo");
      ScreenOut.kPrintln();
    }
```

Wie erwartet ergibt ein Durchlauf des Programms bsp02039.HalloHallo bis auf den Programmaufruf und die Startmeldung die gleiche Bildschirmausgabe wie bsp02034.HalloHallo:



2.5.6.5 Besondere Formulierungen der for Schleife

Die for-Schleife ermöglicht eine Vielzahl von Varianten bei der Festlegung des Schleifenindex. Einige Beispiele für diese Varianten zeigt das Beispiel jpr02144.

Standard ist das in Einserschritten aufsteigende Fortzählen der Schleifenvariablen. Hier zum Beispiel werden aufsteigend alle Elemente eines Arrays ausgegeben

```
int[] zweierPot = {1,2,4,8,16};
for (int i=0;i<zweierPot.length;i++)
  ScreenOut.kPrint(zweierPot[i]+" ");
```

Eine for-Schleife kann aber auch absteigend formuliert werden. Man erhält *absteigend alle Indizes i mit 4 >= i >= 0* auf folgende Weise::

```
for (int i=4;i>=0;i--)
  ScreenOut.kPrint(i+" ");
```

Der Schleifenindex muss nicht stets mit Schrittweite Eins codiert werden, es sind auch andere Schrittweiten möglich. Im folgenden Beispiel ist die Schrittweite 2 gewählt, man erhält so *aufsteigend alle ungeraden Indizes i mit -5 <= i <= 5*:

```
for (int i=-5;i<=5;i=i+2)
  ScreenOut.kPrint(i+" ");
ScreenOut.kPrintln();
```

Diese Zahlen können auch in einer anderen Reihenfolge erzeugt werden. Hierbei kann man die folgende for-Schleife verwenden, die zunächst *die Zahlen -1 und +1* liefert:

```
for (int i=-1;i<=1;i=i+2)
  ScreenOut.kPrint(i+" ");
```

Mit dieser Schleife können dann wieder *alle ungeraden Indizes i mit -5 <= i <= 5* erzeugt werden, jetzt aber in einer anderen Reihenfolge:

```
for (int i=1;i<=5;i=i+2)
  for (int j=+1;j>=-1;j=j-2)
    ScreenOut.kPrint(i*j+" ");
```

Die Wirkung dieser Varianten der for-Schleife zeigt der Bildschirmausdruck des Programms jpr02144:

```
Administrator: Eingabeaufforderung
...\jpr02144>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02144.ForSchleifeVari
--------------------------------------------------------------------------------
    Beispiele fur Varianten der for-Schleife
    Copyright 2018 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------

aufsteigend fur alle Elemente eines Arrays ...
1 2 4 8 16

absteigend fur i mit 4 >= i >= 0 ...
4 3 2 1 0

aufsteigend fur alle ungeraden i mit -5 <= i <= 5 ...
-5 -3 -1 1 3 5

fur -1 und +1 ...
-1 1

fur plusminus 1, plusminus 3, plusminus 5 ...
1 -1 3 -3 5 -5

Drücken Sie eine beliebige Taste . . .
```

2.5.6.6 for-each Schleife

Die for-each Schleife ist eine Variante der for-Schleife, die ab Java 6 verfügbar ist. Der Name ist etwas irreführend, da das Schlüsselwort each *nicht* auftaucht. Die for-each Schleife ist in erster Linie in folgenden Fällen interessant:

- Schleifen über die Elemente von Arrays
- Schleifen über Arrays aus Objekten
- Schleifen über Kollektionen

Die obige Liste zeigt, dass die for-each Schleife für Programmieranfänger erst einmal nicht interessant ist. Der Vollständigkeit halber werden dennoch hier grundlegende Beispiele für die Anwendung der for-each-Schleife vorgestellt. Dabei werden allerdings Sprachelemente verwendet, die erst in nachfolgenden Kapitel behandelt werden. Insofern kann dieses Unterkapitel beim ersten Lesen übersprungen werden. Die for-each Schleife wird entsprechend auch nicht für die Lösung der Übungsaufgaben zu Kapitel 2.5.6 benötigt.

Die einfachste Anwendung der for-each Schleife zeigt der folgende Abschnitt aus dem Programm bsp02145.ForEachElemArr:

```
// eindimensionaler Array
    int[] einDim = {1,2,3,5,7};

// Ausgabe der Elemente eines eindimensionalen des Arrays
    ScreenOut.kPrintln("Ausgabe eines eindimensionalen Arrays:");
    int durchlaufNr = 1;
    for (int i:einDim)
    {
      ScreenOut.kPrintln("Durchlauf: "+durchlaufNr+" - Wert des Zählers i: "+i);
      durchlaufNr++;
    }
```

Man sieht, dass der Array einDim einmal durchlaufen wird und dass aufsteigend die Werte des Arrays in der Konsole ausgegeben werden:

```
Administrator: Eingabeaufforderung
...\jpr02145>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02145.ForEachElemArr
--------------------------------------------------------------------------------
    for-each Schleife uber Elemente eines Arrays
    Copyright 2012 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Ausgabe eines eindimensionalen Arrays:
Durchlauf: 1 - Wert des Zählers i: 1
Durchlauf: 2 - Wert des Zählers i: 2
Durchlauf: 3 - Wert des Zählers i: 3
Durchlauf: 4 - Wert des Zählers i: 5
Durchlauf: 5 - Wert des Zählers i: 7
```

Im Quellcode der for-each Anweisung

- wird eine Laufvariable i vom Typ int deklariert, die nacheinander die Werte des Arrays einDim annehmen soll.

Dabei muss beachtet werden,

- dass die Variable i den gesamten Array einDim durchläuft vom Anfang bis zum Ende und
- dass beim Durchlaufen jedes Element des Arrays einDim einen Durchlauf der Schleife ergibt

Vorteile der for-each-Schleife sind:

- Der Programmierer muss nicht mehr überlegen, über welche Indexwerte die Schleife laufen soll.

- Eine IndexOutOfBoundsException wegen Herauslaufen aus dem Array (vgl. 2.6.2) kann nicht auftreten.

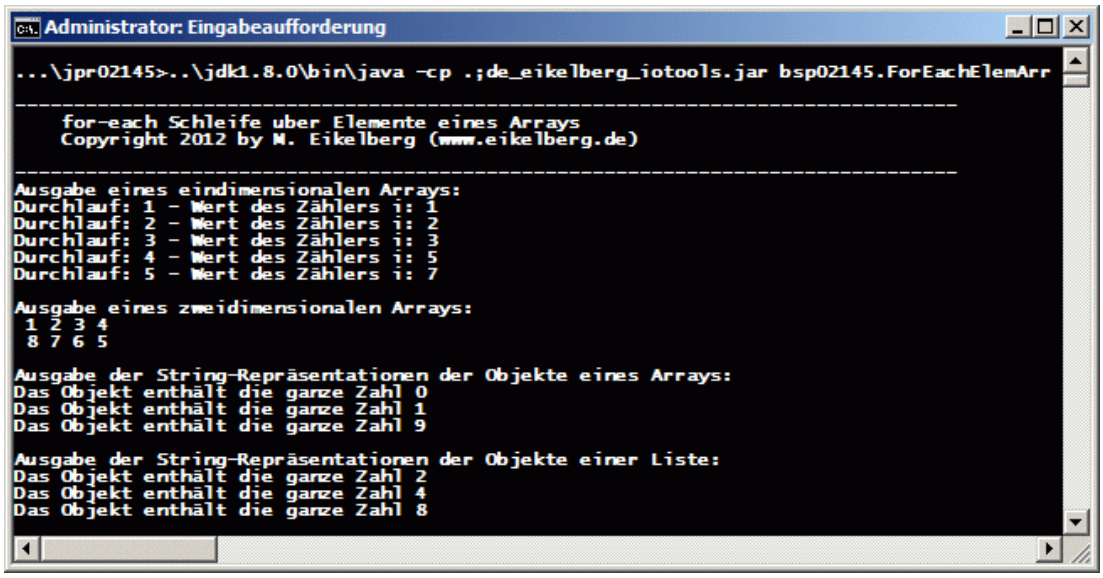
Die weiteren Vorteile erkennt der erfahrenere Programmierer in den übrigen Anwendungsmöglichkeiten der for-each Schleife, die im folgenden Listing des Programms bsp02145.ForEachElemArr aufgezeigt werden. Man beachte dabei, dass für das Verständnis des Programms bsp02145.ForEachElemArr die Kenntnis einiger in nachfolgenden Kapiteln erklärter Sprachelemente erforderlich ist (Arrays siehe Kapitel 2.6.2, Klassen siehe Kapitel 2.8, Interfaces siehe Kapitel 2.9.4, Generics siehe Kapitel 2.11). Dies zeigt, dass die for-each Schleife eher von geübten und erfahrenen Programmierern genutzt werden kann.

Es folgt der Quellcode des Programms bsp02145.ForEachElemArr mit den vier wesentlichen Anwendungen der for-each Schleife:

```
package bsp02145;

import java.util.Arrays;
import java.util.List;

import de.eikelberg.iotools.*;

public class ForEachElemArr
{
  public static void main(String[] args)
  {
//  Startmeldung
    ScreenOut.kStartScreenOut(
      "for-each Schleife über Elemente eines Arrays",
      "M. Eikelberg (www.eikelberg.de)","2012");

// eindimensionaler Array
    int[] einDim = {1,2,3,5,7};

// Ausgabe der Elemente eines eindimensionalen des Arrays
    ScreenOut.kPrintln("Ausgabe eines eindimensionalen Arrays:");
    int durchlaufNr = 1;
    for (int i:einDim)
    {
      ScreenOut.kPrintln("Durchlauf: "+durchlaufNr+" - Wert des Zählers i: "+i);
      durchlaufNr++;
    }

// zweidimensionaler Array
    int[][] zweiDim = {
                       {1,2,3,4},
                       {8,7,6,5}
                      };

// Ausgabe der Elemente eines zweidimensionalen Arrays
    ScreenOut.kPrintln("\nAusgabe eines zweidimensionalen Arrays:");
    for (int[] zeile:zweiDim)                        // für jede Zeile des Arrays
    {
      for (int element:zeile)                        // für jedes Element der Zeile
      {
        ScreenOut.kPrint(" "+element);
      }
      ScreenOut.kPrintln();
    }

// Array aus Objekten
    GanzZahl[] gzArray = {
        new GanzZahl(0),
        new GanzZahl(1),
        new GanzZahl(9),
    };

// Schleife zur Ausgabe der Objekte des Arrays
    ScreenOut.kPrintln("\nAusgabe der String-Repräsentationen der Objekte eines Arrays:");
    for (GanzZahl gz:gzArray)
    {
      ScreenOut.kPrintln(gz.toString());
    }

// (einfach verkettete) Liste anlegen (List ist ein Interface, das vom Interface Collection
erbt)
    List<GanzZahl> gzList = Arrays.asList(
        new GanzZahl(2),
```

```
        new GanzZahl(4),
        new GanzZahl(8)
    );

// Schleife zur Ausgabe der Objekte der Liste
    ScreenOut.kPrintln("\nAusgabe der String-Repräsentationen der Objekte einer Liste:");
    for (GanzZahl gz:gzList)
    {
      ScreenOut.kPrintln(gz.toString());
    }
  }
}

class GanzZahl
{
  private int gz;

  public GanzZahl(int gz)
  {
    this.gz = gz;
  }

  @Override
  public String toString()
  {
    String ret = "Das Objekt enthält die ganze Zahl "+gz;
    return ret;
  }
}
```

Beim Programmlauf erfolgt aufsteigend für jedes Element des angegebenen Arrays bzw. der angegebenen Liste ein Schleifendurchlauf:



2.5.6.7 Loops bei der Codierung von for-Schleifen

Wie bei den anderen Schleifenkonstrukten auch können bei der Codierung von for-Schleifen durch Programmierfehler unbeabsichtigte Programmloops hervorgerufen werden. Dies geschieht in der Regel durch ungeeignete Codierung des Ausdrucks "logischerAusdruck" im Kopf der for-Schleife. Durch Abwandlung des Programms bsp02041.ZaehlschleifeFloat kann sehr leicht ein entsprechendes Beispiel gewonnen werden.

Ersetzt man im Programm bsp02041.ZaehlschleifeFloat die Quellcode-Zeile

```
for (double potenz=1; potenz <= 8; potenz *= faktor)
```

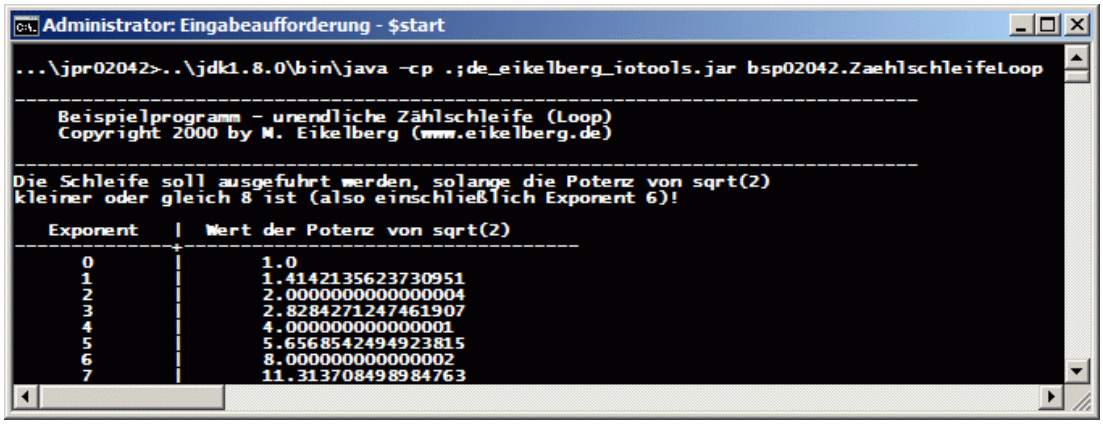
durch die Zeile

```
for (double potenz=1; potenz != 8; potenz *= faktor)
```

so erhält man mathematisch gesehen die gleiche Verarbeitung: Das Programm soll dann enden, wenn die Potenz der Wurzel aus 2 den Wert 8 erhält. Dies geschieht mathematisch gesehen bei Potenz 6. In 2.5.6.4 war jedoch bereits gezeigt worden, dass der Wert 8 niemals genau erreicht wird. Die logische Bedingung `potenz <= 8` sorgte sicher - jedoch leider im falschen Schritt - für die Beendigung der for-Schleife.

Fordert man jetzt jedoch, dass `potenz != 8` verletzt werden soll, so wird dies niemals eintreten und es entsteht eine unendliche Schleife.

Das Beispielprogramm zur unendlichen Schleife bsp02042.ZaehlschleifeLoop stimmt im Prinzip vollständig mit dem Programm bsp02041.ZaehlschleifeFloat überein. Der einzige wesentliche Unterschied besteht in der oben beschriebenen Änderung der Abbruchbedingung der for-Schleife.



Man sieht, dass im Durchlauf zu Exponent 6 das Abbruchkriterium der for-Schleife knapp verfehlt wurde. Anschliessend steigen die Werte der Variablen `potenz` weiter an. Das Programm loopt, bis es - zum Beispiel mit Strg+C - abgebrochen wird.

2.5.6.8 Laufzeitfehler wegen irrtümlicher Codierung eines Semikolons direkt nach den Anweisungen für die Steuerung der Wiederholungen einer for-Schleife

Ein häufig auftretender Fehler, der vom Compiler nicht gefunden wird und der zu einer fehlerhaften Verarbeitung führt, ist das irrtümliche Codieren eines Semikolons direkt nach dem Paar runder Klammern mit den Anweisungen für die Steuerung der Wiederholungen einer for-Schleife. Beispielprogramm jpr02239 enthält diesen Fehler:

```
package bsp02239;

...

public class ForFehler
{

// for Schleife
    for (int i=0;i<100;i++);  // Fehler - hier darf kein Semikolon stehen!
    {
// Verarbeitung in der for-Schleife / Anweisungsblock
        ...
    }
  }
}
```

Das am Ende der Zeile mit dem Schlüsselwort for eingefügte Semikolon bewirkt, dass die for-Schleife sich nicht mehr auf den nachfolgenden Block bezieht und diesen Block wiederholt, sondern dass die for-Schleife den leeren Block wiederholt, der sich zwischen der schließenden runden Klammer um die Anweisungen bzw. die logische Bedingung zur Steuerung der Schleife und dem nachfolgenden Semikolon befindet. In diesem Block befindet sich keine Verarbeitung, also wird im Beispiel 100 mal ein leerer Block wiederholt. Der nachfolgende mit dem Kommentar "Verarbeitung in der for-Schleife / Anweisungsblock" gekennzeichnete Block wird dagegen nicht wiederholt, sondern nur einmal ausgeführt!

2.5.7 Verwendung von break und continue in Schleifen

2.5.7.1 break und continue-Anweisungen ohne Label

Bei der Codierung von Schleifen können zwei Sprachelemente verwendet werden, die bei einem Programmentwurf mit Hilfe von Struktogrammen nicht erforderlich sind. Diese beiden Sprachelemente ermöglichen es, den Ablauf des Anweisungsblocks in den Schleifen abzubrechen:

- Mit der Anweisung **`break;`** kann die Bearbeitung des Anweisungsblocks abgebrochen werden. Bei der Verwendung von `break;` setzt das Programm mit der ersten Anweisung nach der Schleifenanweisung fort.
- Verwendet man die Anweisung **`continue;`**, so wird die Ausführung des Anweisungsblocks ebenfalls abgebrochen und zum Ende des Anweisungsblocks gesprungen. Anschließend wird jedoch die Bearbeitung der Schleifenanweisung fortgesetzt.

> **Hinweis:**Generell muss von der Verwendung von `break;` und `continue;` bei der erstmaligen Codierung eines Programms dringend abgeraten werden. Die scheinbare Notwendigkeit, eine dieser Anweisungen verwenden zu müssen, ist in der Regel durch Mängel beim Programmentwurf zu erklären. Lediglich Tuningmaßnahmen in fertigen Programmen können die Verwendung von `break;` und `continue;` eventuell rechtfertigen. Aber auch hier kann in aller Regel die gleiche Wirkung durch Verbesserung des Programmdesign erreicht werden. Generell verschlechtern die Verwendung von `break;` und `continue;` die Lesbarkeit und Wartbarkeit eines Programms. Daher muss die Verwendung dieser beiden Anweisungen *in Schleifen* eher als schlechter Programmierstil angesehen werden.

Beispiel für die Verwendung von break; und continue; ist das Programm bsp02043.NoLabelJump. Zweck dieses Programms ist

- das Lesen aller Zeichen einer Datei daten\eingabe.txt mit anschließendem
- Schreiben der gelesenen Zeichen in eine Datei daten\ausgabe.txt,
- wobei alle Leerzeichen aus dem Text entfernt werden.

Diese Verarbeitung geschieht in folgendem Teil des Java Quellcodes von bsp02043.NoLabelJump:

```
// Dateien öffnen
    DatasetOut dsOut1;
    dsOut1 = new DatasetOut("daten\\ausgabe.txt");
    DatasetIn dsIn1;
    dsIn1 = new DatasetIn("daten\\eingabe.txt");

// Schleife für die Verarbeitung
    char zeichen;
    int status;
    while (true)
    {
      zeichen = dsIn1.dReadChar();
      status = dsIn1.dGetStatus();
// Beenden der Schleife bei Dateiende
      if (status != 0) break;
      if (zeichen == ' ')
// Abbruch der Verarbeitung des Anweisungsblocks bei Leerzeichen
        continue;
      else
        dsOut1.dPrint(zeichen);
    }
```

Wird also vom Programm bsp02043.NoLabelJump das Dateiende gefunden, so wird die Verarbeitung der while-Schleife ganz abgebrochen. Wird dagegen ein Leerzeichen gefunden, so entfällt lediglich die weitere Verarbeitung des Anweisungsblocks und es wird mit der nächsten Verarbeitung des Anweisungsblocks der while-Schleife fortgesetzt. In den Augen des Autors ist der obige Code schlecht geschrieben und wird nur als ein Beispiel für die Verwendung von break und continue angeführt. Die gleiche Verarbeitung hätte man eleganter und besser wie folgt codieren können:

```
// Dateien öffnen
    DatasetOut dsOut1;
    dsOut1 = new DatasetOut("daten\\ausgabe.txt");
    DatasetIn dsIn1;
    dsIn1 = new DatasetIn("daten\\eingabe.txt");

// erstmaliger Lesevorgang
    char zeichen;
    int status;
    zeichen = dsIn1.dReadChar();
    status = dsIn1.dGetStatus();
```

```
// Schleife für die weitere Verarbeitung
    while (status == 0)
    {
      if (zeichen != ' ')
        dsOut1.dPrint(zeichen);
      zeichen = dsIn1.dReadChar();
      status = dsIn1.dGetStatus();
    }
```

Diese zur bsp02043.NoLabelJump äquivalente Verarbeitung ist in dem Programm bsp02043.NoJumpIsBetter realisiert.

2.5.7.2 break und continue-Anweisungen mit Label

Mit den Anweisungen `break;` und `continue;` ist es sogar möglich, bei mehrfach verschachtelten Schleifen in höher liegende Schleifen zu springen. Hierzu muss im Programm eine Stelle markiert werden, an die gesprungen werden soll. Selbstverständlich gelten für diese Form der beiden Sprunganweisungen erst recht alle kritischen Anmerkungen in 2.5.7.1.

Das Markieren eines Sprungziels zu `break;` bzw. `continue;` geschieht mit Hilfe eines sogenannten Labels. Ein **Label** markiert eine eine Schleife,

- indem ein Bezeichner für die Schleife gewählt wird und
- indem dieser Bezeichner vor das erste Schlüsselwort der Schleife - also vor while, do oder for - eingefügt wird,
- wobei Bezeichner und erstes Schlüsselwort der Schleife durch einen Doppelpunkt getrennt werden.

Bei der Codierung einer break- oder continue-Anweisung fügt man ein solches Label rechts neben dem Schlüsselwort break bzw. continue in die Anweisung ein, um anzuzeigen, an welche Stelle man springen will. Gibt man kein Label an, so bezieht sich der Sprung stets auf die kleinste umgebende Schleife.

Die Verwendung von break- und continue-Anweisungen mit Labeln zeigt das Beispielprogramm bsp02044.LabelJump. bsp02044.LabelJump verarbeitet die Eingabedatei daten\eingabe.txt im Prinzip wie bsp02043.NoLabelJump bis auf den Unterschied, dass von bsp02044.LabelJump alle Punkte in der Eingabedatei durch Zeilenumbrüche in der Ausgabedatei ersetzt werden. Ferner fügt bsp02044.LabelJump eine Leerzeile zu Beginn der Ausgabedatei ein. Die beschriebene Verarbeitung geschieht in folgendem Java Quellcode:

```
// Dateien öffnen
    DatasetOut dsOut1;
    dsOut1 = new DatasetOut("daten\\ausgabe.txt");
    DatasetIn dsIn1;
    dsIn1 = new DatasetIn("daten\\eingabe.txt");

// Schleife für die Verarbeitung
    char zeichen;
    int status;
ende: while (true)
    {
      dsOut1.dPrintln();
weiter: while (true)
      {
        zeichen = dsIn1.dReadChar();
        status = dsIn1.dGetStatus();
        if (status != 0) break ende;
        if (zeichen == ' ')
          continue weiter;
        else
          if (zeichen == '.')
            break weiter;
          else
            dsOut1.dPrint(zeichen);
      }
    }
```

Nach Meinung des Autors sieht dieser Programmcode sehr verworren aus. Im Code sind zwei while-Schleifen vorhanden, die mit den Labeln `ende` und `weiter` versehen wurden. Die Wirkung der break- und continue-Anweisungen ist wie folgt:

- Wenn ein Leerzeichen eingelesen wird, wird die Verarbeitung des Anweisungsblocks der inneren (mit weiter gekennzeichneten) Schleife abgebrochen und mit der nächsten Verarbeitung des Anweisungsblocks der weiter-Schleife fortgesetzt.
- Wird ein Punkt gefunden, wird die Verarbeitung der weiter-Schleife ganz abgebrochen und mit der Verarbeitung der mit dem Label ende versehenen Schleife fortgesetzt.
- Wird Dateiende festgestellt (status != 0 ist erfüllt), so wird auch die Verarbeitung der äußeren ende-Schleife beendet.

Wenn Sie Schwierigkeiten haben, diese Verarbeitung zu verstehen, so sehen Sie direkt, warum die Verwendung von break und continue nachteilig ist. *Versuchen Sie, auf die Verwendung von* `break` *und* `continue` *ganz zu verzichten!*

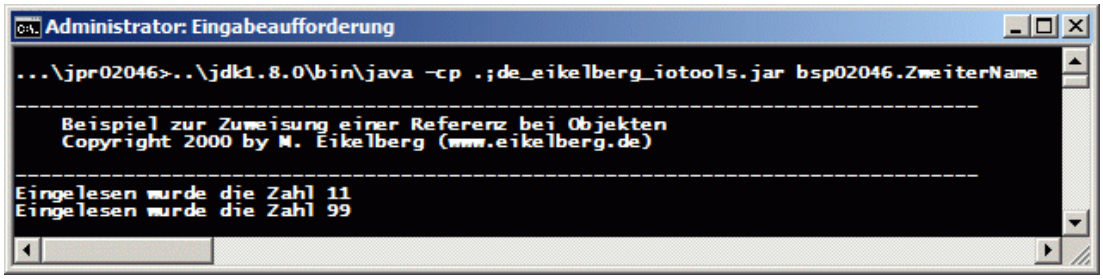
2.6 Referenztypen

2.6.1 Eigenschaften von Referenztypen

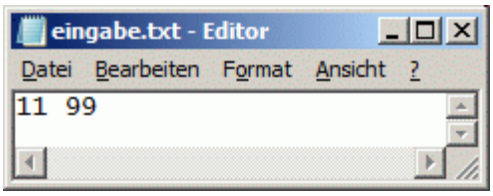
Wie in Kapitel 2.3.3.2 erklärt müssen sowohl Variablen zu elementaren Datentypen als auch Objekte vor der Nutzung im Programm deklariert werden.

- Ein Unterschied zwischen Variablen zu elementaren Datentypen und Objekten besteht darin, dass Variablen zu elementaren Datentypen vom Compiler automatisch angelegt werden, Objekte jedoch explizit mit Hilfe des new-Operators instantiiert werden müssen.
- Ein zweiter Unterschied zwischen Variablen zu elementaren Datentypen und Objekten besteht darin, dass elementare Datentypen durch die Sprachdefinition festgelegt werden, Objekte jedoch anhand von Klassen angelegt werden, die nicht notwendig Bestandteil der Sprachdefinition sein müssen.

Mit diesem Unterschied hängt ein weiterer Unterschied zwischen den Variablen zu elementaren Datentypen und den Objekten zusammen. Dieser Unterschied wirkt sich bei der Verwendung des Gleichheitsoperators, des Ungleichheitsoperators und des Zuweisungsoperators aus. Wir betrachten hierzu das folgende Beispielprogramm bsp02046.ZweiterName, das aus einer Eingabedatei die beiden Zahlen 11 und 99 einliest und auf dem Bildschirm ausgibt. Ein Durchlauf dieses Programms ergibt folgende Bildschirmausgabe:



Das Programm bsp02046.ZweiterName hat die beiden in einer Datei daten\eingabe.txt gespeicherten Zahlen gelesen und in den Konsolbildschirm geschrieben:



Betrachtet man den Programmcode von bsp02046.ZweiterName, so findet man eine Besonderheit - auf *eine Datei* wird über *zwei Objektnamen* zugegriffen:

- Im Programm werden zwei Objekte dsIn1 und dsIn2 der Klasse DatasetIn deklariert.
- Das Objekt dsIn1 wird instantiiert, das Objekt dsIn2 wird jedoch nicht instantiiert.
- Dem Objekt dsIn2 wird mit dem Zuweisungsoperator das Objekt zu dsIn1 zugewiesen.
- Die Lesevorgänge über dsIn1 und dsIn2 erfolgen damit beide aus derselben Datei.

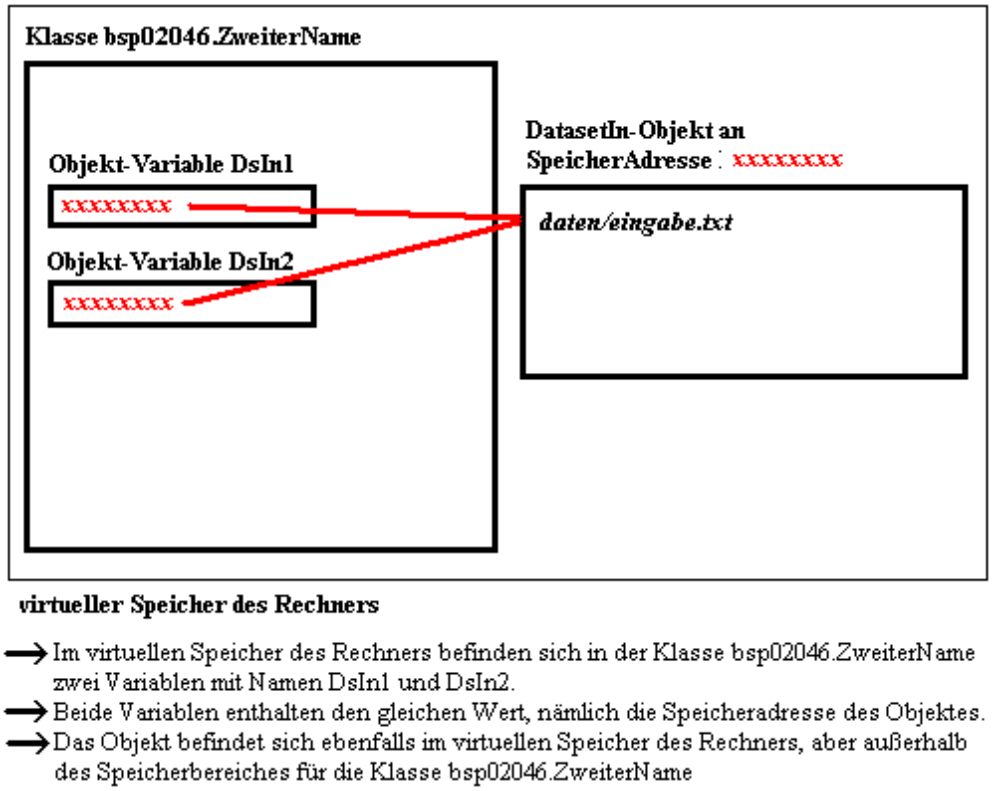
Die entsprechenden Bereiche des Programmcodes sind im folgenden Listing (Ausdruck des Programmcode) hervorgehoben:

```
package bsp02046;

import de.eikelberg.iotools.*;

public class ZweiterName
{

  public static void main(String[] args)
  {
// Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispiel zur Zuweisung einer Referenz bei Objekten",
      "M. Eikelberg (www.eikelberg.de)","2000");

// Objekt zur Eingabedatei anlegen
    DatasetIn dsIn1;
    dsIn1 = new DatasetIn("daten\\eingabe.txt");

// Erste Zahl in der Eingabedatei lesen und auf dem Bildschirm ausgeben
    int i;
```

```
    i = dsIn1.dReadInt();
    ScreenOut.kPrintln("Eingelesen wurde die Zahl "+i);

// Dem Objekt einen zweiten Namen geben (Zuweisung einer Referenz)
    DatasetIn dsIn2;
    dsIn2 = dsIn1;

// Einlesen über dsIn2 führt zur Verwendung des Objektes zu dsIn1
    i = dsIn2.dReadInt();
    ScreenOut.kPrintln("Eingelesen wurde die Zahl "+i);
  }
}
```

Offensichtlich führt die Verwendung von dsIn1 und dsIn2 zum Zugriff auf dieselbe Datei. Auch der Zugriffsweg muss derselbe sein, da beim Zugriff über dsIn2 der vorherige Lesevorgang über dsIn1 bekannt ist und der Lesevorgang über dsIn2 entsprechend hinter der bereits über dsIn1 gelesenen Zahl erfolgt. Der Grund für dieses Verhalten lässt sich wie folgt erklären:

Die Verwendung des Zuweisungsoperators zwischen Objekten führt dazu, dass den Daten zum rechts neben dem Zuweisungsoperator stehenden Objekt lediglich ein zweiter Objektname zugewiesen wird:



Dem Objekt dsIn1 wird also im vorliegenden Beispiel zusätzlich der zweite Name dsIn2 zugewiesen. Die Verwendung beider Namen führt de facto zum Zugriff auf *die selben Daten*. Diesen Sachverhalt kann man in folgender Regel zusammenfassen:

Erfolgt eine Zuweisung zwischen Objekt-Variablen,

- so wird anders als bei elementaren Datentypen kein Duplikat der Daten angelegt, sondern es wird der Inhalt der Objekt-Variablen - also die Objekt-Adresse (Referenz) - kopiert.
- De facto wird also für das links vom Zuweisungsoperator stehende Objekt werden **die selben Daten** wie zum rechts vom Zuweisungsoperator stehenden Objekt verwendet.
- Die beiden Objekte sind also nach der Zuweisung **nicht lediglich gleich**, sondern beziehen sich auf **identische Daten** im Speicher.
- Durch die Zuweisung erhält man also lediglich zwei Namen, die beide auf einen einzigen Datenbereich (eine einzige Referenz) verweisen. Man spricht daher von **Referenzgleichheit**.

Das Phänomen der Unterscheidung zwischen inhaltlicher Gleichheit und Referenzgleichheit betrifft

- nicht nur den **Zuweisungsoperator**,
- sondern auch **Gleichheits-** und **Ungleichheitsoperator**.

Bei der Abfrage auf Gleichheit wird also nicht lediglich nach inhaltlicher Gleichheit gefragt, sondern strenger geprüft, ob beide Objektnamen auf dieselben Daten zeigen.

Alle Java Datentypen, für die der Zuweisungsoperator Referenzgleichheit herbeiführt, heißen **Referenzdatentypen**. Es gibt die folgenden drei großen Gruppen von Referenzdatentypen:

- Klassen
- Schnittstellen (Interfaces)
- Arrays

Die Schnittstellen werden nicht in der klassenbasierten Programmierung, sondern erst in der objektorientierten Programmierung verwendet und sind daher Thema von Kapitel 2.8ff dieses Kurses. Für die klassenbasierte Programmierung besonders wichtig sind dagegen die folgenden drei Referenzdatentypen, die Thema der Kapitel 2.6.2 bis 2.6.4 sind:

- **Array**
- **String**
- **StringBuffer**

Die Datentypen String und StringBuffer sind Java-Standardklassen und gehören zum sehr oft verwendeten Teil des Sprachumfangs von Java und ermöglichen die Verarbeitung von Zeichenketten. Der Datentyp Array ist keine Java Standardklasse, er ist jedoch klassenähnlich aufgebaut. Arrays sind insbesondere bei der Codierung von Schleifen sehr nützlich.

2.6.2 Arrays

2.6.2.1 Begriffe

Ein **Array** besteht aus einer beliebigen Anzahl von Variablen gleichen Typs. Diese Variablen können einfache Datentypen, Objekte oder auch wieder Arrays sein und werden als **Komponenten** des Arrays bezeichnet. Besteht ein Array aus n vielen Komponenten, so werden die Komponenten von 0 bis n-1 durchnummeriert. Die Nummer einer Komponenten heißt **Index**, die Gesamtzahl der Komponenten n heißt **Länge** des Array. Statt des Begriffs Array wird manchmal auch der deutsche Begriff **Feld** verwendet.

2.6.2.2 Syntax

Ein Array wird wie ein Objekt deklariert und instantiiert. Die Deklaration eines Arrays mit Namen *ArrayName* und Komponenten vom Datentyp *DatenTyp* kann wie folgt codiert werden:

```
Datentyp [] ArrayName ;
```

Alternativ ist aber auch die folgende Form erlaubt:

```
Datentyp  ArrayName [];
```

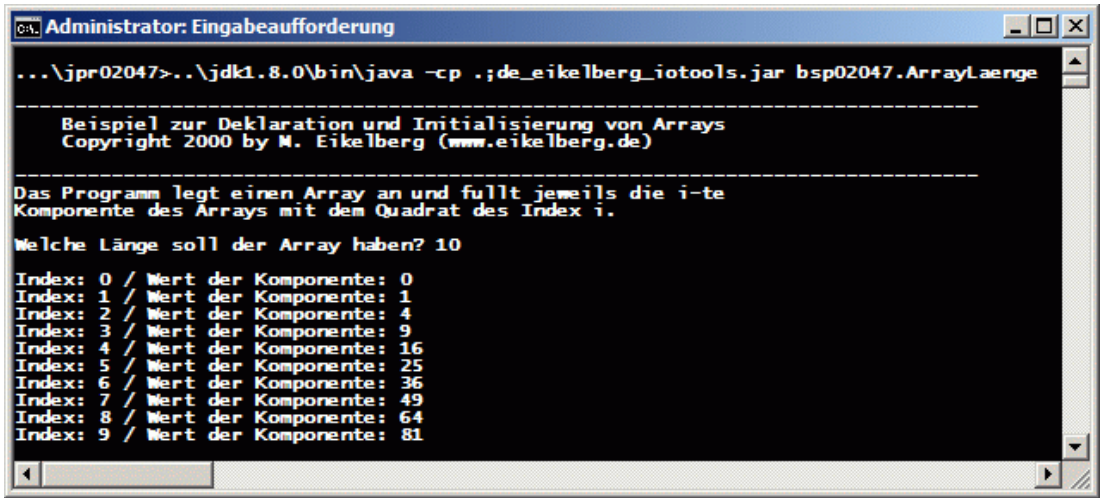
Mit diesen Anweisungen wird lediglich ein Array mit Name *ArrayName* deklariert und der Datentyp der Komponenten des Arrays festgelegt. Für den Array ist jedoch noch kein Speicherplatz reserviert worden. Ferner ist die Anzahl der Elemente des Arrays ebenfalls noch nicht bestimmt. Die Reservierung von Speicherplatz und die Festlegung der Länge des Arrays geschieht bei der Initialisierung des Arrays, die wie folgt codiert wird:

```
ArrayName = new Datentyp [n ];
```

wobei *n* die Länge des Arrays ist. Man sieht in Bezug auf die Syntax deutlich die Ähnlichkeit zur Deklaration und Instantiierung von Objekten. Es bestehen jedoch auch Unterschiede. So kann der *DatenTyp* zum Beispiel ein einfacher Datentyp sein und braucht nicht notwendig eine Klasse zu sein. Auch treten bei Arrays eckige Klammern in der Syntax auf, die bei Objekten fehlen.

In einigen Programmiersprachen ist es erforderlich, dass die Länge eines Arrays fest im Programmcode eingetragen wird. Dies ist bei der Programmierung in Java nicht notwendig. Vielmehr kann die Länge je nach Programmsituation unterschiedlich groß gewählt werden. Ist die Länge des Arrays jedoch erst einmal festgelegt, so kann sie anschließend nicht mehr verändert werden.

Ein Beispiel zur Deklaration und Initialisierung eines Arrays ist das Programm bsp02047.ArrayLaenge. Dieses Programm fragt zunächst ab, wie groß die Länge des Arrays sein soll und legt anschließend einen Array aus Komponenten des Datentyps int mit der richtigen Länge an. Anschließend werden die Komponenten mit dem Quadrat des Index-Wertes der jeweiligen Komponenten gefüllt und schließlich der Inhalt der Komponenten wieder ausgegeben. Ein Programmlauf von bsp02047.ArrayLaenge ergibt bei Eingabe einer 10 folgende Bildschirmausgabe:



Im Programm ist folgender Java Programmcode enthalten:

```
// Länge des Arrays erfragen
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Welche Länge soll der Array haben? ");
    int n;
    n = KeyboardIn.kReadInt();
    ScreenOut.kPrintln();
```

```
// Array deklarieren und initialisieren
    long[] quadrate;
    quadrate = new long[n];
```

Man sieht in diesem Programmstück die Deklaration und Initialisierung des Arrays mit Namen `quadrate`. Die Komponenten des Arrays sind vom Datentyp long. Der Index eines Arrays muss stets vom Typ int sein, daher ist im obigen Programmstück die Variable `n` als Variable vom Typ int vereinbart.

Das Programm bsp02047.ArrayLaenge füllt die Komponenten des Arrays mit long Zahlen. Hierzu ist der Zugriff auf die Komponenten des Arrays erforderlich. Ein Zugriff auf die i-te Komponente eines Arrays mit Namen *`ArrayName`* erhält man durch folgenden Ausdruck:

```
ArrayName [i]
```

Dementsprechend erfolgt die Verarbeitung des Programms bsp02047.ArrayLaenge wie folgt:

```
// Array mit Werten füllen
    for (int i=0; i<n; i++)
      quadrate[i] = i*i;

// Werte des Arrays ausgeben
    for (int i=0; i<n; i++)
      ScreenOut.kPrintln("Index: "+i+" / Wert der Komponente: "+quadrate[i]);
```

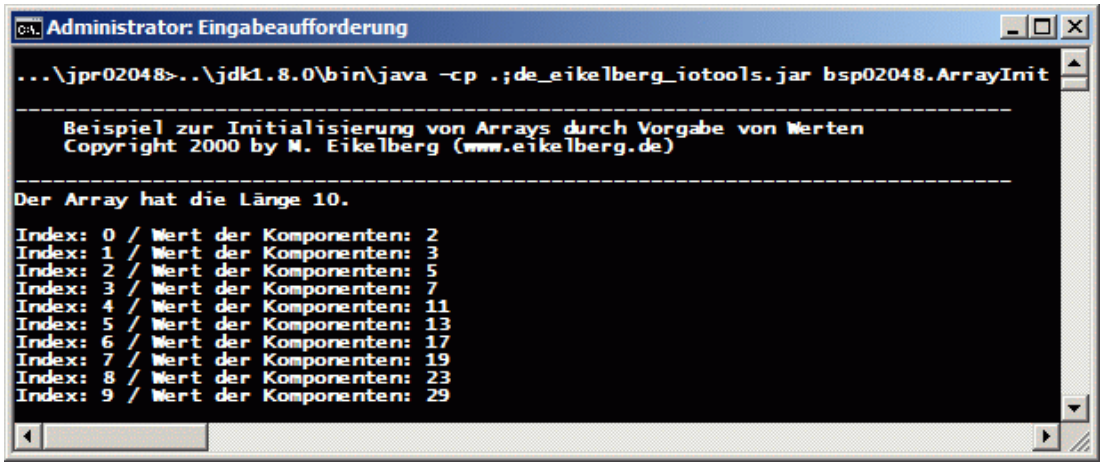
Man sieht deutlich den Vorteil der Verwendung von Arrays im Zusammenhang mit Schleifenkonstruktionen.

In Programm bsp02047.ArrayLaenge geschieht die Initialisierung des Arrays ohne Zuweisung von Werten zu den Komponenten des Arrays. Vielmehr werden die Werte durch den obigen Programmcode während des Programmlaufs zugewiesen. Alternativ können den Komponenten eines Arrays bei der Deklaration Werte zugewiesen werden. In diesem Fall entfällt die Initialisierung des Arrays durch die Angabe der Anzahl der Elemente, da sich diese ja aus der Anzahl der bei der Initialisierung angegebenen Werte ergibt. Im Programmlauf ist diese Länge des Arrays unabhängig von der Art der Initialisierung stets in der Variablen **length** des Arrays gespeichert.

Ein Beispiel zur Initialisierung eines Arrays durch Angabe von Werten für die Komponenten mit anschließender Abfrage der Länge des Arrays ist das Programm bsp02048.ArrayInit. Es enthält den folgenden Programmcode:

```
// Array deklarieren und initialisieren
    long[] primzahlen = {2,3,5,7,11,13,17,19,23,29};

// Länge des Arrays feststellen
    int laenge;
    laenge = primzahlen.length;
    ScreenOut.kPrintln("Der Array hat die Länge "+laenge+".");
    ScreenOut.kPrintln();

// Inhalt des Arrays ausgeben
    for (int i=0; i<laenge; i++)
      ScreenOut.kPrintln("Index: "+i+" / Wert der Komponenten: "+primzahlen[i]);
```

Man beachte, dass die Zuweisung der Werte direkt bei der Deklaration geschehen muss. Eine Teilung der Deklaration und der Zuweisung in zwei Anweisungen ist nicht möglich. Ein Durchlauf des Programms bsp02048.ArrayInit ergibt folgende Bildschirmanzeige:



2.6.2.3 Laufzeitfehler

Ein häufiges Problem bei der Verarbeitung von Arrays ist darin begründet, dass im Programm Zugriffe auf Komponenten eines Arrays versucht werden, die nicht existieren, da der Index-Bereich des Arrays verlassen wurde. Hat zum Beispiel ein Array die Länge n, so gibt es lediglich Komponenten mit Indizes 0 bis n-1, da die Kompontenten eines Arrays mit 0 beginnend indiziert werden. Wird ein Zugriff unter

Verwendung eines Index außerhalb dieses Bereiches versucht, so erhält man eine sogenannte **ArrayIndexOutOfBoundsException**. Eine solche ArrayIndexOutOfBoundsException tritt beim Durchlauf des Beispielprogramms bsp02049.ArrayException auf:

```
Administrator: Eingabeaufforderung
...\jpr02049>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02049.ArrayException
--------------------------------------------------------------------------------
    Beispiel zum Laufzeitfehler ArrayIndexOutOfBoundsException
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Der Array hat die Länge 5.

Index: 0 / Wert der Komponenten: 2
Index: 1 / Wert der Komponenten: 3
Index: 2 / Wert der Komponenten: 5
Index: 3 / Wert der Komponenten: 7
Index: 4 / Wert der Komponenten: 11
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at bsp02049.ArrayException.main(ArrayException.java:26)
```

Untersucht man den Programmcode von bsp02049.ArrayException, so bemerkt man den folgenden Fehler: Bei der Formulierung der Schleife für die Ausgabe der Komponenten des Arrays wurde irrtümlich die Schleife so codiert, dass sie auch noch einmal durchlaufen wird, wenn der Index des Arrays `primzahlen` den Wert der Länge `primzahlen.length` annimmt:

```
// Array deklarieren und initialisieren
    long[] primzahlen = {2,3,5,7,11};

// Länge des Arrays feststellen
    int laenge;
    laenge = primzahlen.length;
    ScreenOut.kPrintln("Der Array hat die Länge "+laenge+".");
    ScreenOut.kPrintln();

// Inhalt des Arrays ausgeben
    for (int i=0; i<=laenge; i++)
      ScreenOut.kPrintln("Index: "+i+" / Wert der Komponenten: "+primzahlen[i]);
```

Ein weiterer möglicher Laufzeitfehler bei der Verarbeitung von Arrays ist die sogenannte **NullPointerException**. Dieser Laufzeitfehler tritt auf, wenn Arrays nicht inititalisiert werden. Beispiel hierzu ist das Programm bsp02050.ArrayNullPExc, das drei verschiedene Situationen demonstriert, bei denen NullPointerException Ausnahmen auftreten können. Der folgende Bildschirmausdruck zeigt zunächst die Fehlermeldung für die erste Situation:

```
Administrator: Eingabeaufforderung
...\jpr02050>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02050.ArrayNullPExc
--------------------------------------------------------------------------------
    Beispiel zum Laufzeitfehler NullPointerException
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Welche Situation der Arrayverarbeitung möchten Sie testen?

    Wertzuweisung zur ersten Komponente        [1]:
    Feststellung der Länge des Arrays          [2]:
    Ausgabe des Wertes der ersten Komponente   [3]:

Geben Sie Ihre Auswahl ein: 1
Exception in thread "main" java.lang.NullPointerException
        at bsp02050.ArrayNullPExc.main(ArrayNullPExc.java:36)
```

Wählt man eine der beiden anderen Menüpunkte, so erhält man die folgenden Fehlermeldungen:

```
Administrator: Eingabeaufforderung
Geben Sie Ihre Auswahl ein: 2
Exception in thread "main" java.lang.NullPointerException
        at bsp02050.ArrayNullPExc.main(ArrayNullPExc.java:40)
```

```
Administrator: Eingabeaufforderung
Geben Sie Ihre Auswahl ein: 3
Exception in thread "main" java.lang.NullPointerException
        at bsp02050.ArrayNullPExc.main(ArrayNullPExc.java:46)
```

Alle drei Fehler wurden dadurch hervorgerufen, dass der zu verarbeitende Array nicht initialisiert wurde. Der zur Fehlersituation gehörende Teil des Programmcodes lautet wie folgt:

```
...
```

```
public class ArrayNullPExc
{

// Array deklarieren - keine Initialisierung
  private static long primzahlen[];

  public static void main(String[] args)
  {
...
    ScreenOut.kPrint("Geben Sie Ihre Auswahl ein: ");
// Als Ausblick auf Kapitel 2.6.3 die einfachste Art, ein Zeichen einzulesen:
    String s = KeyboardIn.kReadLine();  // Alle Zeichen bis Return einlesen
    auswahl = s.charAt(0);  // ersten Character aus Zeichenkette herauslösen

// Verarbeitung
    switch (auswahl)
    {
      case '1':
// Wertzuweisung zur ersten Komponente des Arrays
        primzahlen[0] = 1;             // Zeile 36
      case '2':
// Länge des Arrays feststellen
        int laenge;
        laenge = primzahlen.length;    // Zeile 40
        ScreenOut.kPrintln("Der Array hat die Länge "+laenge+".");
        ScreenOut.kPrintln();
        break;
      case '3':
// Inhalt des Arrays ausgeben - die nächste Zeile ist Zeile 46
        ScreenOut.kPrintln("Index: 0 / Wert der Komponenten: "+primzahlen[0]);
        break;
      default:
        ScreenOut.kPrintln("Keine Verarbeitung zur Auswahl verfügbar!");
    }
...
```

2.6.2.4 Inhaltliche Gleichheit und Referenzgleichheit bei Arrays

Wie in 2.6.1 bereits angesprochen wird bei Arrays zwischen inhaltlicher Gleichheit und Referenzgleichheit unterschieden. Es gelten folgende Regeln:

- Der Zuweisungsoperator "=" bezieht sich auf die Referenz.
- Die Vergleichsoperatoren "==" und "!=" beziehen sich auf die Referenz.
- Inhalte von Arrays können mit dem Befehl `System.arraycopy(...)` kopiert werden.
- Der Vergleich der Inhalte von Arrays erfolgt durch direkten Vergleich der Komponenten des Arrays.

Der Befehl

```
System.arraycopy(Object src,int src_position,Object dst,int dst_position,int length)
```

- kopiert length viele Werte von aufeinanderfolgenden Komponenten des Arrays scr in den Array dst,
- beginnend bei dem Element mit dem Index src_position,
- und speichert die Werte im Array dst beginnend mit der Komponente mit dem Index dst_position.

Der Befehl System.arraycopy(...) ist somit äquivalent zu folgender Schleife:

```
  for (int i=0;i<length;i++)
    c3[i+dst_position] = c1[i+src_position];
```

Die Regeln zur inhaltlichen Gleichheit und zur Referenzgleichheit werden durch das Programm bsp02045.GleichheitArray illustriert. Im Programm werden zunächst wie folgt vier Arrays deklariert:

```
// Vereinbarung von vier character Arrays
    char[] c1 = {'H','a','l','l','o',' ','W','e','l','t'};
    char[] c2 = c1;                  // Anweisung (1) - Zuweisung der Referenz !
    char[] c3 = new char[c1.length]; // Anweisung (2) - c3 wird unten gefüllt
    char[] c4 = {'H','a','l','l','o','_','W','e','l','t'};  // Anweisung (4)
```

Der Array c3 hat also die gleiche Länge wie der Array c1. Unter Verwendung von System.arraycopy(...) wird c3 wie folgt mit den Inhalten von c1 gefüllt.

```
/* Fülle c3 mit den Werten von c1:
   Kopiere Inhalt von c1, beginnend mit Offset 0, nach c3,
   beginnend mit Offset 0, und kopiere c1.length viele Zeichen:
*/
    System.arraycopy(c1,0,c3,0,c1.length); // Anweisung (3)
```

Anschließend wird c1 mit den anderen drei Arrays verglichen. Dieser Vergleich erfolgt nach folgendem Muster:

```
// Vergleich von c1 und c2
    ScreenOut.kPrintln("Vergleich von c1 und c2:");
    if (c1 == c2)
      meldeReferenzGleichheit("c1","c2");
    else
      meldeReferenzUngleichheit("c1","c2");
// Vergleich von c1 und c2
    vergleicheArray(c1,c2,"c1","c2");
```

meldeReferenzGleichheit(...) und meldeReferenzUngleichheit(...) sind einfache Methoden, die lediglich eine passende Bildschirmmeldung ausgeben. Die Methode vergleicheArray(...) führt den Vergleich der Inhalte der Arrays durch. Dieser Vergleich geschieht wie folgt:

```
    boolean equal = true;
    if (c1.length != c2.length)
      equal = false;     // ist die Länge verschieden, so sind c1 und c2 ungleich
    else
    {
      for (int i=0;i<c1.length & equal;i++)  // es genügt, einen Unterschied zu finden
      {
        if (c1[i] != c2[i])
        {
          equal = false;
        }
      }
    }
    if (equal == true)
      meldeWertGleichheit(name1,name2);
    else
      meldeWertUngleichheit(name1,name2);
```

Auch hier sind die Methoden meldeWertGleichheit(...) und meldeWertUngleichheit(...) einfache Methoden, die passende Bildschirmmeldungen ausgeben.

Gemäß den Erklärungen in 2.6.1 muss das Programm folgendes Ergebnis liefern (die Anweisungsnummern finden Sie in den Kommentaren im oben aufgelisteten Quelltext von bsp02045.GleichheitArray):

1. Vergleich von c1 und c2:
   - c1 und c2 sind referenzgleich wegen Anweisung (1).
   - c1 und c2 sind inhaltlich gleich wegen der Referenzgleichheit.
2. Vergleich von c1 und c3:
   - c1 und c3 sind nicht referenzgleich, da c3 in Anweisung (2) neu angelegt wird.
   - c1 und c3 sind inhaltlich gleich, da c3 unter Verwendung von System.arraycopy(...) in Anweisung (3) mit den Inhalten von c1 gefüllt wird.
3. Vergleich von c1 und c4:
   - c1 und c4 sind nicht referenzgleich wegen Anweisung (4).
   - c1 und c4 sind nicht inhaltlich gleich, da c1 und c4 für die sechste Komponente verschiedene Character zugewiesen werden.

Die Bildschirmausgabe von bsp02045.GleichheitArray zeigt, dass sich das Programm wie erwartet verhält:

```
Administrator: Eingabeaufforderung
...\jpr02045>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02045.GleichheitArray
--------------------------------------------------------------------------------
    Beispiel zu Gleichheit von Referenz und Wert bei Referenzdatentypen
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Vergleich von c1 und c2:
Die Array-Variablen c1 und c2 sind referenzgleich.
Die Array-Variablen c1 und c2 enthalten gleiche Werte.

Vergleich von c1 und c3:
Die Array-Variablen c1 und c3 sind nicht referenzgleich.
Die Array-Variablen c1 und c3 enthalten gleiche Werte.

Vergleich von c1 und c4:
Die Array-Variablen c1 und c4 sind nicht referenzgleich.
Die Array-Variablen c1 und c4 enthalten ungleiche Werte.
```

2.6.2.5 Beispiel: Einlesen von Gleitpunktzahlen aus einer Datei

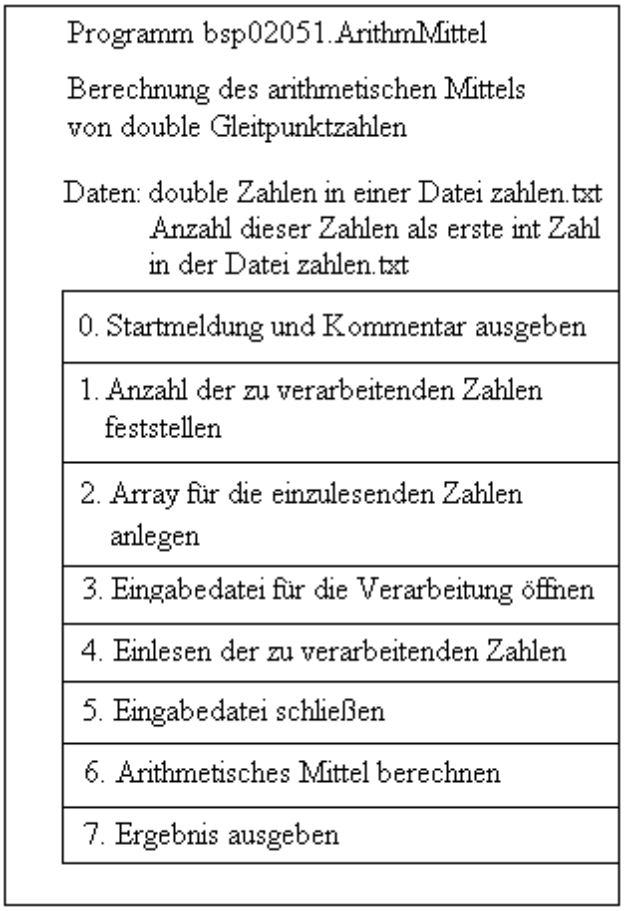
Wie bereits angedeutet sind Arrays insbesondere nützlich im Zusammenhang mit Schleifenkonstruktionen. Als Beispiel folgt hier ein Programm, das Gleitpunktzahlen aus einer Datei einliest und das arthmetische Mittel dieser Zahlen berechnet. Da bei der Verwendung von Arrays die Länge des Arrays bei der Initialisierung festgelegt werden muss, ist es erforderlich, dass vor dem eigentlichen Einlesen der Zahlen zunächst vom Programm festgestellt werden muss, wieviele Zahlen eingelesen werden müssen.

Die einzelnen Schritte des Programms sind im Quellcode dokumentiert:

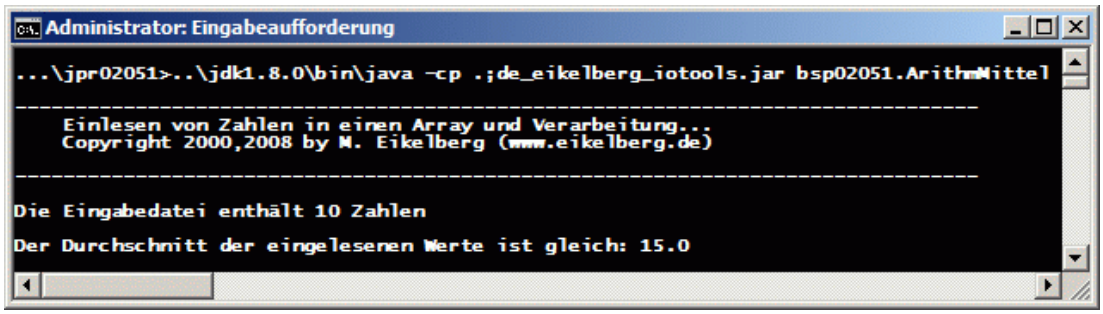
```
package bsp02051;

import de.eikelberg.iotools.*;

public class ArithmMittel
{

  public static void main(String[] args)
  {
// 0. Startmeldung
    ScreenOut.kStartScreenOut(
      "Einlesen von Zahlen in einen Array und Verarbeitung...",
      "M. Eikelberg (www.eikelberg.de)","2000,2008");

// 1. Anzahl der zu verarbeitenden Zahlen feststellen
// 1.1 Eingabedatei öffnen
    DatasetIn Eingabe;
    Eingabe = new DatasetIn("daten\\zahlen.txt");

// 1.2 Zahlen zählen
    int anzahl = 0;
    int status = 0;
    while (status == 0)
    {
        Eingabe.dReadDouble();
        status = Eingabe.dGetStatus();
        if (status == 0) anzahl = anzahl + 1;
    }
    ScreenOut.kPrintln("\nDie Eingabedatei enthält "+anzahl+" Zahlen");

// 1.3 Eingabedatei wieder schließen
    Eingabe.dClose();

// 2. Array für die einzulesenden Zahlen anlegen
    double werte[];
    werte = new double[anzahl];

// 3. Eingabedatei für die Verarbeitung neu öffnen
    Eingabe = new DatasetIn("daten\\zahlen.txt");

// 4. Einlesen der zu verarbeitenden Zahlen
    status = 0;
    for (int i=0; i<anzahl; i++)
    {
      werte[i] = Eingabe.dReadDouble();
      status = Eingabe.dGetStatus();
      if (status != 0)
      {
```

```
        ScreenOut.kPrintln(
          "\nZu wenig Werte in der Eingabedatei - Verarbeitung wird abgebrochen!");
        i = anzahl;   // Verarbeitung abbrechen
      }
    }

// 5. Eingabedatei schließen
    Eingabe.dClose();

// 6. arithmetisches Mittel berechnen
    double durchschnitt=0.0d;
    if (status == 0)
    {
      for (int i=0; i<anzahl; i++)
        durchschnitt += werte[i];
      durchschnitt /= (double)anzahl;
    }

// 7. Ergebnis ausgeben
    ScreenOut.kPrintln(
      "\nDer Durchschnitt der eingelesenen Werte ist gleich: "+durchschnitt);
  }
}
```

Der Ablauf des Programms erfolgt gemäß folgendem Struktogramm



Im 6. und 7. Programmabschnitt wird also der Durchschnitt der eingelesenen Zahlen berechnet und ausgegeben. Bei einer höheren Rechengenauigkeit als double - zum Beispiel bei Verwendung eines Computeralgebrasystems oder aber bei Rechnung 'zu Fuß' - ergeben die Werte in der Eingabedatei zahlen.txt den Durchschnittswert 14.9995295. Die Rechenungenauigkeit bei der Verwendung von double Zahlen führt dazu, dass bsp02051.ArithmMittel als Ergebnis den Durchschnittswert 15.0 liefert:



2.6.2.6 Mehrdimensionale Arrays

Bei der Definition des Begriffs Array in 2.6.2.1 wurde zugelassen, dass als Komponenten von Arrays wieder Arrays eingesetzt werden können. Dabei ist es nicht notwendig, dass alle als Komponenten verwendeten Arrays die gleiche Länge haben. Das Beispielprogramm bsp02052.MehrDimArray enthält die folgende Deklaration und Initialisierung eines mehrdimensionalen Arrays durch Angabe der Werte.

```
// Deklaration und Initialisierung des Arrays
    int zahlen[][] = {{1,2,3,5,7,9},
                      {11,13,17,19},
                      {23,29}};
```

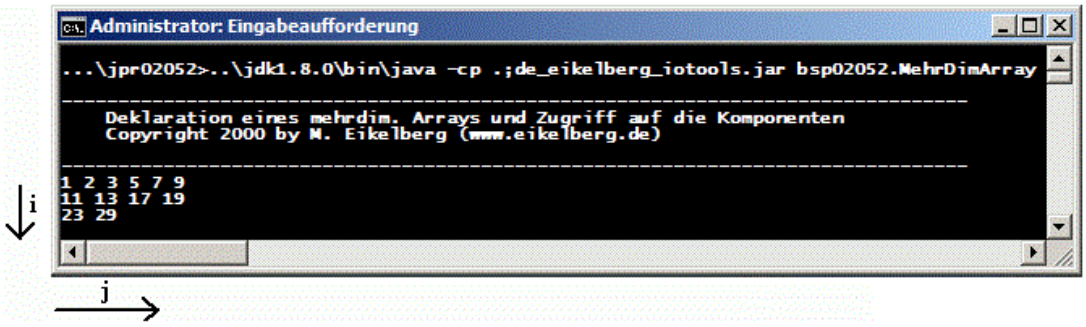
Alternativ kann auch eine verschachtelte Kombination der verschiedenen Möglichkeiten zur Deklaration und Initialisierung von Arrays codiert werden. Man sieht ferner, dass die einzelnen Komponenten-Arrays die unterschiedlichen Längen 6, 4 und 2 haben. Die Länge des Arrays zahlen sowie die Längen der Komponenten-Arrays kann man wie folgt ermitteln:

```
// Bestimmung der Längen von des Arrays zahlen sowie
// der Längen der Komponenten-Arrays

// Länge des Arrays zahlen feststellen
    int pzArrayLaenge = zahlen.length;
// Array für die Längen der Komponenten-Arrays anlegen
    int kompArrayLaenge[] = new int[pzArrayLaenge];
// Längen der Komponenten-Arrays feststellen
    for (int i=0; i<pzArrayLaenge; i++)
      kompArrayLaenge[i] = zahlen[i].length;
```

Der Ausdruck `zahlen.length;` ergibt also die Anzahl der Komponenten des Arrays zahlen und der Ausdruck `zahlen[i].length` ergibt die Länge der Arrays, die Komponenten von primzahl sind. Das Programm bsp02052.MehrDimArray gibt den Inhalt des Arrays zahlen zum Abschluss aus. Dabei erfolgt ein Zugriff auf die einzelnen Elemente von zahlen wie folgt:

```
// Ausgabe des Arrays
    for (int i=0; i<pzArrayLaenge; i++)
    {
      for (int j=0; j<kompArrayLaenge[i]; j++)
      {
        ScreenOut.kPrint(zahlen[i][j]);
        ScreenOut.kPrint(' ');
      }
      ScreenOut.kPrintln();
    }
```

Das j-te Element des Arrays, der seinerseits die i-te Komponente des mehrdimensionalen Arrays zahlen ist, erhält man also mit dem Ausdruck `zahlen[i][j]`. Die folgende Graphik zeigt neben der Ausgabe des Programms bsp02052.MehrDimArray auch die Indizierung der Elemente des Arrays durch den Index des Arrays zahlen (Index i) sowie die Indizes der Komponenten-Arrays (Indizes j).



2.6.2.7 Beispiel zur Verarbeitung von zweidimensionalen Arrays

Im Anwendungsbereich haben Tabellen eine große Bedeutung, weshalb die Programmierung und Verarbeitung zweidimensionaler Arrays ein wichtiges Thema der Programmierung ist. Vor dem Beispiel zur Verarbeitung zweidimensionaler Arrays hier jedoch zunächst folgende Vorbemerkung:

Exkurs: Formatierte Ausgabe von Zahlen

Eng mit dem Thema verbunden (vgl. Punkt 5 oben) ist die Notwendigkeit, für eine geeignete Ausgabe des Arrays die Zahlen "formatiert", also mit einer festen Anzahl von Stellen links und rechts vom Dezimalpunkt auszugeben. Dies kann mit Hilfe der folgenden Methoden geschehen:

```
// Methode gibt Gleitpunktzahlen formatiert aus mit "links" vielen Stellen
// vor dem Komma (mit Leerzeichen links aufgefüllt) und "rechts" vielen
// Stellen nach dem Komma (mit Nullen aufgefüllt

  private static String formatiere(double wert,int links, int rechts)
  {
        String druck = ""+wert;
        return formatiere(druck,links,rechts);
  }

  private static String formatiere(long wert,int links)
  {
        String druck = ""+wert;
        return formatiere(druck,links,0);
  }

  private static String formatiere(String wert,int links, int rechts)
  {
// Zahl in eine Zeichenkette umwandeln
        String druck = ""+wert;
//       Länge der Zahl feststellen
        int laenge = druck.length();
// Position des Gleitpunktes feststellen
        int punktPos = druck.indexOf(".");
        if (punktPos == -1)  // kein Gleitpunkt vorhanden
        {
// links mit Leerzeichen auffüllen
      for (int i=laenge;i<links;i++) druck = " "+druck;
        }
        else
        {
// Stellen der Zahl links und rechts vom Gleitpunkt feststellen
          int linkeLaenge = punktPos;
          int rechteLaenge = laenge-punktPos-1;
// links mit Leerzeichen auffüllen
          for (int i=linkeLaenge;i<links;i++) druck = " "+druck;
// rechts mit Nullen auffüllen
          for (int i=rechteLaenge;i<rechts;i++) druck = druck+"0";
        }
// Rückgabe der formatierten Zahl
        return druck;
  }
```

Die Methoden formatiere(...) sind *überladen*, d.h. es handelt sich um drei Methoden, die sich nur durch die Parameter unterscheiden:

- private static String formatiere(**String wert**,int links, int rechts) - formatiert eine Zeichenkette, die einen Dezimalpunkt enhalten kann, aber nicht enthalten muss. Sie ergänzt die als Parameter übergebene Zeichenkette links mit Leerzeichen sowie rechts mit Nullen, bis die Zeichenkette mit der Zahl links vom Dezimalpunkt "links" viele Stellen sowie rechts vom Dezimalpunkt "rechts" viele Stellen aufweist. Ist kein Dezimalpunkt enthalten, wird der Wert für den Parameter rechts entsprechend ignoriert.
- private static String formatiere(**long wert**,int links) - formatiert eine ganze Zahl. Sie wandelt die Zahl in eine Zeichenkette um und ergänzt die Zeichenkette nach links um Leerzeichen, bis sie "links" viele Zeichen lang ist. Intern ruft diese Methode die Methode String formatiere(**String wert**,int links, int rechts) auf.
- private static String formatiere(**double wert**,int links, int rechts) - formatiert eine Gleitpunktzahl. Sie wandelt die Zahl in eine Zeichenkette um und ergänzt die Zeichenkette nach links um Leerzeichen, bis sie "links" viele Zeichen lang ist, sowie nach rechts um Nullen, bis "rechts" viele Nachkommastellen vorhanden sind. Intern ruft diese Methode die Methode String formatiere(**String wert**,int links, int rechts) auf.

Beispiel zu zweidimensionalen Arrays

Das folgende Beispiel bsp02104.EinlesenQuadratDatenbestand zeigt die wichtigen Schritte zu diesem Thema (die Nummerierung der Schritte entspricht der Nummerierung in den Kommentaren im Programm). Damit das Beispiel nicht zu aufwändig wird, wird für das Beispiel

angenommen, dass der Datenbestand "rechteckig" ist, dass also alle Zeilen der Tabelle gleich lang sind.

1. Anzahl der Zeilen der Tabelle festtellen
2. Anzahl der Werte je Zeile feststellen
3. Deklarieren und Instanziieren des 2-dim Arrays
4. Einlesen der Werte und Speichern der Werte im Array
5. Ausgabe des Arrays
6. Spaltenweise den Mittelwert bilden

Mit Hilfe der Methodenformatiere(...) kann die Verarbeitung in bsp02104.EinlesenQuadratDatenbestand jetzt wie folgt codiert werden:

```
package bsp02104;

import java.util.StringTokenizer;
import de.eikelberg.iotools.*;

public class EinlesenQuadratDatenbestand
{
  public static void main(String[] args)
  {
// 0. Startmeldung des Programms
        ScreenOut.kStartScreenOut(
          "Beispielprogramm zur Verarbeitung zweidim. Arrays",
          "M. Eikelberg (www.eikelberg.de)","2008");

// 1. Anzahl der Zeilen der Tabelle festtellen
// 1.1 Datei zum ersten Mal öffnen
    DatasetIn messwDatei;
    messwDatei = new DatasetIn("daten/daten.txt");
// 1.2 Zeilen zählen
    int zeilenZahl = 0;
    int status = 0;
    while (status == 0)
    {
      String testZeile;
      testZeile = messwDatei.dReadLine();
      testZeile = testZeile.trim();
      status = messwDatei.dGetStatus();
// Aufpassen, Leerzeilen am Ende der Datei sollten nicht mitgezählt werden!
      if (status ==0 & ! testZeile.equals("")) zeilenZahl++;
    }
        ScreenOut.kPrintln("\nDer Datenbestand enthält "+zeilenZahl+" Zeilen.\n");
// 1.3 Datei schließend
    messwDatei.dClose();

// 2. Anzahl der Werte je Zeile feststellen
/* Es wird hier vorausgesetzt, dass alle "Zeilen" des Arrays - also
   die 1-dim Arrays für ein festes i - die gleiche Länge haben   */
// 2.1 Datei zum zweiten Mal öffnen
    messwDatei = new DatasetIn("daten/daten.txt");
// 2.2 Zahl der Werte feststellen
    String zeile;
    zeile = messwDatei.dReadLine();
    StringTokenizer st = new StringTokenizer(zeile," ");
    int spaltenZahl;
    spaltenZahl = st.countTokens();
    ScreenOut.kPrintln("Der Datenbestand enthält "+spaltenZahl+" Spalten.\n");
// 2.3 Datei wieder schließen
    messwDatei.dClose();

// 3. Deklarieren und Instanziieren des 2-dim Arrays
    double[][] messw;
    messw = new double[zeilenZahl][spaltenZahl];

// 4. Einlesen der Werte und Speichern der Werte im Array
// 4.1 Datei zum dritten Mal öffnen
    messwDatei = new DatasetIn("daten/daten.txt");
// 4.2 Einlesen der Werte
// In der nächsten Zeile wäre auch i<zeilenZahl möglich:
    for (int i=0; i<messw.length; i++)
    {
// messw[i] ist die i-te Zeile des Arrays, sie hat die
// Länge zahlen[i].length. j<spaltenZahl wäre auch möglich
```

```
      for (int j=0; j<messw[i].length; j++)
      {
        messw[i][j]=messwDatei.dReadDouble();
      }
    }
// 4.3 Datei zum dritten Mal schließen
    messwDatei.dClose();

// 5. Ausgabe des Arrays in der Konsole
    for (int i=0; i<messw.length; i++)
    {
// Jede Zeile einrücken
      ScreenOut.kPrint("     ");
      for (int j=0; j<messw[i].length; j++)
      {
// Zahl formatiert mit 3 Stellen links vom Komma und 2 Stellen
// rechts vom Komma ausgeben
        ScreenOut.kPrint(formatiere(messw[i][j],3,2));
        ScreenOut.kPrint(' ');
      }
      ScreenOut.kPrintln();
    }
    ScreenOut.kPrintln();

// 6. Spaltenweise den Mittelwert bilden
// 6.1 eindim Array für die Mittelwerte anlegen
    double[] mwert = new double[spaltenZahl];
// 6.2 Aufsummieren der Werte für jede Spalte
    for (int i=0; i<messw.length; i++)
    {
      for (int j=0; j<messw[i].length; j++)
      {
        mwert[j] = mwert[j] + messw[i][j];
      }
    }
// 6.3 Bildung des Mittelwertes
    for (int j=0;j<spaltenZahl;j++)
    {
      mwert[j] = mwert[j] / zeilenZahl;
    }
// 6.4 Ausgabe der Mittelwerte
    for (int j=0;j<spaltenZahl;j++)
    {
      ScreenOut.kPrintln(
        "Der Mittelwert fuer Spalte "+formatiere((j+1),2)
        +" ist gleich "+formatiere(mwert[j],4,15));
    }
  }

  ...
  private static String formatiere(double wert,int links, int rechts)
  {
    ...
  }

  private static String formatiere(long wert,int links)
  {
    ...
  }

  private static String formatiere(String wert,int links, int rechts)
  {
    ...
  }
}
```

Das Programm bsp02104.EinlesenQuadratDatenbestand liest aus der Datei daten/daten.txt einen tabellarischen Bestand aus 10 Spalten und 17 Zeilen und verarbeitet diesen Datenbestand:

```
Administrator: Eingabeaufforderung
...\jpr02104>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02104.EinlesenQuadratDaterbestand
--------------------------------------------------------------------------------
    Beispielprogramm zur Verarbeitung zweidim. Arrays
    Copyright 2008 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------

Der Daterbestand enthält 17 Zeilen.

Der Daterbestand enthält 10 Spalten.

      23.09  21.62  21.84 201.40 202.00 201.40 202.00 201.40  48.41  48.41
      22.33  23.46  21.99 201.40 202.00 201.40 202.00 201.40  48.38  48.44
      23.27  21.69  21.99 201.40 202.00 201.40 202.00 201.40  48.41  48.44
      23.88  21.78  22.08 201.40 202.00 201.40 202.00 201.40  48.41  48.44
      23.15  21.84  21.81 201.40 202.00 201.40 202.00 201.40  48.41  48.47
      23.27  21.29  22.05 201.40 202.00 201.40 202.00 201.40  48.38  48.41
      22.81  21.66  21.29 201.40 202.00 201.40 202.00 201.40  48.44  48.44
      23.58  21.44  21.87 201.40 202.00 201.40 202.00 201.40  48.44  48.47
      23.06  21.17  22.14 201.40 202.00 201.40 202.00 201.40  48.44  48.44
      21.41  21.66  22.11 201.40 202.00 201.40 202.00 201.40  48.44  48.41
      24.01  19.61  21.96 201.40 202.00 201.40 202.00 201.40  48.44  48.44
      22.48  21.75  21.32 201.40 202.00 201.40 202.00 201.40  48.44  48.47
      23.06  21.78  22.14 201.40 202.00 201.40 202.00 201.40  48.47  48.44
      23.27  22.02  22.11 201.40 202.00 201.40 202.00 201.40  48.41  48.38
      22.97  21.78  22.14 201.40 202.00 201.40 202.00 201.40  48.44  48.38
      23.03  21.56  19.40 201.40 202.00 201.40 202.00 201.40  48.41  48.41
      22.66  21.44  21.47 201.40 202.00 201.40 202.00 201.40  48.47  48.47

Der Mittelwert fuer Spalte  1 ist gleich   23.019411764705882
Der Mittelwert fuer Spalte  2 ist gleich   21.620588235294115
Der Mittelwert fuer Spalte  3 ist gleich   21.747647058823530
Der Mittelwert fuer Spalte  4 ist gleich  201.400000000000060
Der Mittelwert fuer Spalte  5 ist gleich  202.000000000000000
Der Mittelwert fuer Spalte  6 ist gleich  201.400000000000060
Der Mittelwert fuer Spalte  7 ist gleich  202.000000000000000
Der Mittelwert fuer Spalte  8 ist gleich  201.400000000000060
Der Mittelwert fuer Spalte  9 ist gleich   48.425882352941170
Der Mittelwert fuer Spalte 10 ist gleich   48.432941176470590
```

2.6.3 String-Objekte

2.6.3.1 String-Objekte, String Literale und Zuweisung auf Referenzebene

Wie in anderen Programmiersprachen auch gibt es in Java Sprachelemente für die Verarbeitung von Folgen zusammengehörender Schriftzeichen. Im Allgemeinen heißt eine solche Folge zusammengehörender Schriftzeichen **String** bzw. **Zeichenkette**. In der Java-Programmierung gibt es Datentypen für die Verarbeitung von Folgen von Schriftzeichen:

- Eine nach ihrer Instantiierung nicht mehr veränderbare Folge von Schriftzeichen wird in Java durch Objekte der Klasse **java.lang.String** bzw. kurz **String** realisiert. Wird eine Zeichenkette in einem Objekt der Klasse String gespeichert, so werden wir von einem **String-Objekt** sprechen.
- Möchte man in einem Programm eine Folge zusammengehörender Schriftzeichen später noch verändern können, so muß man in Java diese Zeichenkette durch Objekte der Klasse **java.lang.StringBuffer** bzw. kurz **StringBuffer** realisieren. Dementsprechend wird im Folgenden bei einer Realisierung einer Zeichenkette in einem Objekt der Klasse StringBuffer von einem StringBuffer-Objekt die Rede sein.

Um die Begriffe genauer auseinander halten zu können, werden im Folgenden entweder allgemein die Begriffe Zeichenkette oder String-Objekt bzw. StringBuffer-Objekt verwendet werden. Man beachte, dass in der Literatur stellenweise der Begriff String unterschiedlich verwendet wird.

In Java werden unveränderbare Zeichenketten als String-Objekte, also als Objekte der Klasse **java.lang.String** realisiert. Dabei wird zur Codierung der Schriftzeichen wie üblich die Unicode-Zeichentabelle zugrunde gelegt.

Objekte der Klasse java.lang.String können wie andere Objekte instantiiert werden. Da die Klasse String Bestandeil des Paketes java.lang ist, ist zur Verwendung der String-Klasse keine import-Anweisung notwendig.

Bei der Verwendung der Klasse String muss jedoch eine kleine Besonderheit beachtet werden, die ist im folgenden Java Code sichtbar wird. Dieser Beispiel-Code ist Bestandteil des Programms bsp02045.GleichheitString und führt zur *Deklaration* von 5 Namen zu String-Objekten:

```
// Vereinbarung von 5 Namen zu 3 String-Objekten
    String s1 = "Hallo Welt";  // (1) Standard-Verfahren
    String s2 = s1;            // (2) Zuweisung der Referenz
// In der folgenden Anweisung (3) wird wegen der Gleichheit des
// String-Literals mit dem String-Literal in Anweisung (1) nur eine
// Referenz auf das Objekt zu s1 erstellt:
    String s3 = "Hallo Welt";
    String s4 = new String("Hallo Welt");
                               // (4) neues Objekt wg. new-Operator
    String s5 = new String("Hallo_Welt");
                               // (5) neues Objekt wg. new-Operator
```

Ausgangspunkt für das richtige Verständnis der Verwendung der String-Klasse ist das Verständnis des String-Literals.

Ein **String-Literal** ist eine feste Folge von char Schriftzeichen, die zusammenhängend verarbeitet wird und die im Programm zwischen doppelten Hochkommata eingegeben wird. Die Codierung eines String-Literals führt automatisch zur Instantiierung eines String-Objektes, das das String-Literal aufnimmt. Dieses Objekt hat zwar keinen Namen, wird jedoch vom Compiler stets wieder verwendet, wenn das zugehörige String-Literal im Quellcode gefunden wird.

`"Hallo Welt"` ist ein solches String-Literal. Fügt man also in den Quellcode das String-Literal `"Hallo Welt"` ein, so wird ein passendes String-Objekt angelegt, das allerdings keinen Namen hat. Die Anweisungen (1) bis (5) in obigem Quelltext verwenden das Objekt zum String-Literal `"Hallo Welt"` und sind wie folgt zu verstehen:

- In **Anweisung (1)** und **Anweisung (3)** werden zum String-Literal `"Hallo Welt"` String-Objekte s1 und s3 deklariert. Über eine Zuweisung auf Referenzebene wird erreicht, dass s1 und s3 auf das Objekt zum String-Literal `"Hallo Welt"` zeigen. Sowohl s1 und s3 sind also Objekt-Namen, die beide auf das String-Objekt zu `"Hallo Welt"` zeigen.
  Die Anweisungen (1) und (3) haben also die gleiche Wirkung bis auf die Tatsache, dass bei der Verarbeitung von Anweisung (1) zusätzlich ein Objekt zum String-Literal `"Hallo Welt"` angelegt wird, wenn dieses String-Literal an dieser Stelle zum ersten Mal im Quelltext auftaucht. Die Form der Deklaration eines String-Objektes in Anweisung (1) und Anweisung (3) hat also zur Folge, dass die Variablen s1 und s3 auf Referenzebene gleich sind. Somit sind diese Anweisungen *nicht* äquivalent zur Deklaration eines String-Objektes mit Hilfe des new-Operators wie in Anweisung (4):
- In **Anweisung (4)** wird unter Verwendung des new-Operators ausdrücklich ein neues Objekt der Klasse String angelegt. Beim Anlegen dieses Objektes wird der Inhalt des String-Objektes zum String-Literal `"Hallo Welt"` vom Konstruktor in das neue Objekt kopiert. **Anweisung (5)** hat die gleiche Wirkung wie Anweisung (4), verwendet jedoch ein neues String-Literal.
- **Anweisung (2)** bewirkt eine Zuweisung auf Referenzebene und hat damit die gleiche Wirkung wie Anweisung (3). Insgesamt sind damit im Programm drei Objekte instantiiert worden, das Objekt zum String-Literal `"Hallo Welt"` sowie die beiden Objekte zu s4 und s5. Die Objekt-Namen s1, s2 und s3 zeigen alle auf das Objekt zu `"Hallo Welt"`. Dementsprechend sind s1,s2 und s3 auf Referenzebene gleich. s1,s2,s3 und s4 sind inhaltlich gleich und s5 unterscheidet sich von den anderen Objekten sowohl in bezug auf Inhalt als auch in bezug auf die Referenz.

Wie der folgende Bildschirmausdruck zeigt, ergibt das Programm bsp02045.GleichheitString das beschriebene Ergebnis:

```
Administrator: Eingabeaufforderung
...\jpr02045>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02045.GleichheitString
--------------------------------------------------------------------------------
    Beispiel zu Gleichheit von Referenz und Wert bei Referenzdatentypen
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Vergleich von s1 und s2:
Die String-Variablen s1 und s2 sind referenzgleich.
Die String-Variablen s1 und s2 enthalten gleiche Werte.

Vergleich von s1 und s3:
Die String-Variablen s1 und s3 sind referenzgleich.
Die String-Variablen s1 und s3 enthalten gleiche Werte.

Vergleich von s1 und s4:
Die String-Variablen s1 und s4 sind nicht referenzgleich.
Die String-Variablen s1 und s4 enthalten gleiche Werte.

Vergleich von s1 und s5:
Die String-Variablen s1 und s5 sind nicht referenzgleich.
Die String-Variablen s1 und s5 enthalten ungleiche Werte.
```

Man beachte, dass es bei den Methoden der String-Klasse oft um die Position einzelner Schriftzeichen in der Zeichenkette geht. Dabei spricht man wie bei Arrays vom **Index** des Schriftzeichens und zählt dabei die Zeichen der Zeichenkette mit Null beginnend von links nach rechts. Erzeugt man also ein String-Objekt aus einen char Array, so bleibt der Index eines jeden Zeichens gleich.

2.6.3.2 Konstruktoren der String-Klasse

Die Klasse String kennt ingesamt die folgenden 11 Konstruktoren. Von diesen Konstruktoren sind zwei als **deprecated** (engl. missbilligt) gekennzeichnet, werden also in späteren Versionen von Java entfallen und sollten daher schon jetzt nicht mehr verwendet werden. Alle Konstruktoren müssen natürlich den Namen String tragen, da Konstruktoren nach Definition die Methoden einer Klasse sind, die den Klassennamen tragen. Der Unterschied zwischen den Konstruktoren liegt vielmehr in den unterschiedlichen Daten, die als **Argumente** - also zwischen den runden Klammern - eingegeben werden können. Die wichtigsten Konstruktoren nennt die folgende Tabelle:

Konstruktor	Wirkung
String()	Anlegen eines leeren neuen String-Objektes.
String(String value)	Das neue String-Objekt wird mit der Zeichenkette gefüllt, die im String-Objekt mit Namen value gespeichert ist
String(char value[ ])	Die Schriftzeichen im char Array mit Namen value werden zusammengefügt zu einer Zeichenkette und im neuen String-Objekt gespeichert.
String(char value[ ], int offset, int length)	Beginnend bei dem Element mit Index offset werden length viele Zeichen aus dem char Array value[] herauskopiert, zu einer Zeichenkette zusammengefügt und im neuen String-Objekt gespeichert.
String(byte bytes[ ])	Die Elemente des byte Array byte[] werden kopiert und gemäß der Codierungstabelle des Betriebssystems in char Zeichen konvertiert. Anschließend werden diese char Zeichen zu einer Zeichenkette zusammengefügt und im neuen String-Objekt gespeichert.
String(byte bytes[ ], int offset, int length)	Beginnend bei dem Element mit Index offset werden length viele Zeichen aus dem byte Array byte[] herauskopiert und gemäß der Codierungstabelle des Betriebssystems in char Zeichen konvertiert. Anschließend werden diese char Zeichen zu einer Zeichenkette zusammengefügt und im neuen String-Objekt gespeichert.
String(StringBuffer buffer)	Die im StringBuffer-Objekt buffer gespeicherte Zeichenkette wird kopiert und im neuen String-Objekt gespeichert.

Im Programm bsp02053.StringVerarbeitung findet man die folgenden Beispiele hierzu:

```
// 1. Konstruktor aus Tabelle
    String s1 = new String();        // leerer String
// 3. Konstruktor aus Tabelle
    char[] c1 = {'H','a','l','l','o',' ','W','e','l','t','!'};
    String s3 = new String(c1);
// 2. Konstruktor aus Tabelle
    String s2 = new String(s3);      // inhaltlich gleich zu s3, aber
                                     // keine Gleichheit auf Referenzebene
// 4. Konstruktor aus Tabelle
    String s4 = new String(c1,0,5); // die ersten 5 Zeichen aus c1 auswählen
// 5. Konstruktor aus Tabelle
    byte[] b1 = {104,97,108,108,111,32,104,97,108,108,111,33};
                                     // ASCII-Werte im byte-Array
    String s5 = new String(b1);
```

```
// 6. Konstruktor aus Tabelle
    String s6 = new String(b1,6,5); // beginnend bei offset 6 die
                                    // nächsten 5 Werte auswählen
```

2.6.3.3 Vergleich der Inhalte von String-Objekten

Für String-Objekte gelten wie in 2.6.1 und 2.6.3.1 erklärt die Operatoren =, != und == auf Referenzebene und untersuchen damit, ob zwei Objekt-Namen auf *identische* Objekte verweisen. Der Vergleich der Inhalte von Strings muss daher auf andere Weise erfolgen. Da Vergleichsoperatoren für String-Objekte häufig in Programmen benötigt werden, stellt die Klasse java.lang.String entsprechende Methoden bereit. Einige Methoden werden in der folgenden Tabelle zusammengestellt:

Methode	Wirkung
public boolean **equals**(Klasse Objekt1)	Die Zeichenkette im *aktuellen String-Objekt* (s.u.) wird mit der Zeichenkette im Objekt Objekt1 verglichen. Bei Gleichheit erhält man den Wert `true`, ansonsten den Wert `false`.
public boolean **equalsIgnoreCase**(String StringObj1)	Die Zeichenkette im aktuellen String-Objekt wird mit der Zeichenkette im String-Objekt StringObj1 verglichen. Dabei wird nicht zwischen Großschreibung und Kleinschreibung unterschieden. Bei Gleichheit erhält man den Wert `true`, ansonsten den Wert `false`.
public boolean **regionMatches**(int offset1, String StrObj2, int offset2, int length)	Beginnend bei dem (offset1+1)-ten Element im aktuellen String-Objekt sowie beim (offset2+1)-ten Elemente im String-Objekt StrObj2 werden die jeweils length vielen folgenden Zeichen der beiden String-Objekte verglichen.
public boolean **regionMatches**(boolean ignoreCase, int offset1, String StrObj2, int offset2, int length)	Wirkung wie regionMatches(int offset1, String StrObj2, int offset2, int length), jedoch wird nicht zwischen Groß- und Kleinschreibung unterschieden, wenn ignoreCase den Wert `true` hat.
public boolean **startsWith**(String StrObj1)	Methode prüft, ob das aktuelle String-Objekt mit der Zeichenkette im String-Objekt StrObj1 beginnt und gibt entsprechende Wahrheitswerte zurück.
public boolean **endsWith**(String StrObj1)	Methode prüft, ob das aktuelle String-Objekt mit der Zeichenkette im String-Objekt StrObj1 endet und gibt entsprechende Wahrheitswerte zurück.

Alle aufgeführten Methoden sind nicht statisch, werden also unter Zugriff auf ein String-Objekt verwendet, das in der Tabelle als *aktuelles String-Objekt* bezeichnet wird. Die Methode equals(...) wird also wie folgt verwendet:

```
// Methode equals(...)
    if (s2.equals(s3))
      ScreenOut.kPrintln("s2 und s3 sind inhaltlich gleich.");
```

`s2` ist in diesem Beispiel das aktuelle String-Objekt, zu dem die Methode equals(...) aufgerufen wird. equals(...) vergleicht den Inhalt von s2 mit dem Inhalt von s3. Weitere Beispiele zur Verwendung dieser Methoden findet man im Programm bsp02053.StringVerarbeitung:

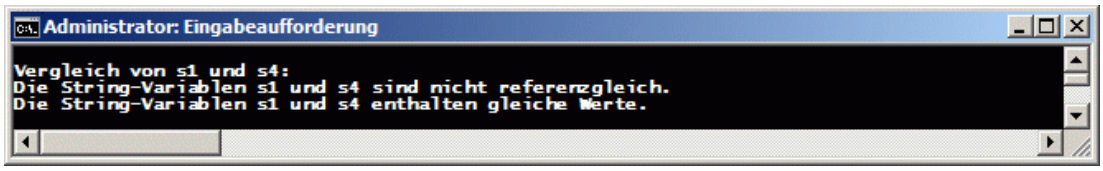
```
    if (s5.equalsIgnoreCase(s3))
      ScreenOut.kPrintln(
        "s3 und s5 sind inhaltlich gleich (Großschreibung ignoriert).");
...
    if (s3.regionMatches(6,s4,0,4))
      ScreenOut.kPrintln(
        "Gleichheit von s3 ab Offset 6 mit s4 ab Offset 0 für die folgenden 4 Zeichen.");
...
    if (s3.regionMatches(true,6,s6,0,4))
      ScreenOut.kPrintln(
        "Gleichheit von s3 ab Offset 6 mit s6 ab Offset 0 für die folgenden 4 Zeichen");
    ScreenOut.kPrintln("(Großschreibung ignoriert).");
...
    if (s3.startsWith("Hallo"))
      ScreenOut.kPrintln("Gleichheit bei Anwendung von startsWith(...) gefunden.");
...
    if (s3.endsWith("Welt!"))
      ScreenOut.kPrintln("Gleichheit bei Anwendung von endsWith(...) gefunden.");
```

Man beachte bei der Formulierung von Abfragen auf Gleichheit, dass der Gleichheitsoperator == String-Objekte wie der Zuweisungsoperator ebenfalls auf Referenzebene wirkt. Dies wird im Folgenden am Beispielprogramm bsp02045.GleichheitString demonstriert. Hierzu betrachte man folgenden Java Code

```
    String s1 = "Hallo Welt";
```

```
...
    String s4 = new String("Hallo Welt");
...
// Vergleich von s1 und s4
    ScreenOut.kPrintln("Vergleich von s1 und s4:");
    if (s1 == s4)
      meldeReferenzGleichheit("s1","s4");
    else
      meldeReferenzUngleichheit("s1","s4");
// Vergleich von Strings geschieht mit der Methode equals()
    if (s1.equals(s4))
      meldeWertGleichheit("s1","s4");
    else
      meldeWertUngleichheit("s1","s4");
```

Ein Durchlauf von bsp02045.GleichheitString ergibt inhaltliche Gleichheit der String-Objekte, jedoch keine Gleichheit der Objekte auf Referenzebene:



2.6.3.4 Zusammenfügen von Strings und Herauslösen aus Strings

Methode / Operator	Wirkung
StrObj1 + StrObj2	**+-Operator**: Zusammenfügen (**Verkettung**) der String-Objekte StrObj1 und StrObj2 zu einem String-Objekt. Assoziativität des Operators ist links. Steht links vom +-Operator ein String, so erfolgt für alle rechts stehenden Ausdrücke eine *automatische Typkonvertierung nach String* .
public String **concat**(String StrObj2)	**Stringverkettung (Konkatenierung)** - fügt den Inhalt des String-Objektes StrObj2 an den Inhalt des aktuellen String-Objektes an. StrObj1.concat(StrObj2) hat also die gleiche Wirkung wie StrObj1 + StrObj2.
public char **charAt**(int indx)	**Zeichenextraktion** - ergibt das Schriftzeichen mit Index indx, also das (indx+1)-te Zeichen des aktuellen String-Objektes.
public String **substring**(int startIndex)	ergibt den Teil der Zeichenkette im aktuellen String-Objekt, der bei Index startIndex beginnt.
public String **substring**(int startIndex, int endIndex)	ergibt den Teil der Zeichenkette im aktuellen String-Objekt, der bei Index startIndex beginnt und *vor* dem Zeichen mit Index endIndex endet.
public String **trim**()	Entfernen von Leerzeichen auf beiden Seiten der Zeichenkette im aktuellen String-Objekt.

Wir betrachten zunächst den +-Operator. Dieser unterscheidet sich syntaktisch nicht von dem arithmetischen +-Operator, so dass der Typ der Operanden entscheidet, welcher Operator verwendet wird. Dies kann zu Fehlersituationen führen. Wir betrachten den folgenden Teil des Programms bsp02053.StringVerarbeitung:

```
    byte a=32,b = 33;                                  // (1)
    if (a+b==65)                                       // (2)
      ScreenOut.kPrintln(a+b);                         // (3)
    if ("3233".equals(""+a+b))   // (4)
      ScreenOut.kPrintln(""+a+b);          // (5)
```

Dieser Programmcode beinhaltet mehrere Besonderheiten:

In Zeile (2) und Zeile (3) wird der arithmetische Additionsoperator verwendet, da beide Operanden Zahlvariablen sind. Der Befehl zur Ausgabe auf dem Konsolbildschirm in Zeile (3) gibt also die Summe der Inhalte der Variablen a und b aus.

In den Zeilen (4) und (5) wird wegen der Assoziativität zunächst der linke +-Operator ausgeführt. Da der *linke* Operand ein String ist, wird für den rechten Operanden eine Typkonversion nach String durchgeführt und der linke (leere) String mit dem rechten String zu einer Zeichenkette "32" zusammengefügt. Anschließend wird rechts noch die Zahl 33 angefügt und es ergibt sich insgesamt die Zeichenkette "3233".

Da "3233" ein String-Objekt ist, kann auf "3233" auch die Methode equals(...) angewendet werden und eine entsprechende Prüfung auf Gleichheit vorgenommen werden.

Zur Methode concat(...) findet man folgendes Beispiel in bsp02045.StringVerarbeitung:

```
ScreenOut.kPrintln(s4.concat(" ").concat(s6));
```

Dieser Java Code ist wie folgt zu verstehen: An den Inhalt des String-Objektes s4 wird zunächst das String-Objekt " " angefügt und es entsteht ein neues String-Objekt ohne Objekt-Namen. Dennoch verfügt dieses Objekt wieder über alle Methoden der String-Klasse, so dass erneut die Methode concat(...) verfügbar ist, die beim zweiten Aufruf jetzt den String s6 anhängt.

Ein Beispiel zur Methode charAt(...) wird in 2.6.5.3 vorgestellt.

Beispiel zu den substring(...)-Methoden sind die folgenden beiden Programmzeilen:

```
ScreenOut.kPrintln(s5.substring(6));       // (6)
ScreenOut.kPrintln(s5.substring(4,8));     // (7)
```

In Zeile (6) wird aus der Zeichenkette "hallo welt" der Teil ab Index 6, also ab dem Buchstaben `w` herausgelöst. In Zeile (7) werden die Zeichen ab Index 4 bis *vor* Index 8, also die Zeichen `o we` herauskopiert.

Ein Beispiel zur Methode trim() kann in 2.6.5.3 gefunden werden.

2.6.3.5 Weitere Methoden der String-Klasse

In diesem Abschnitt werden weitere wichtige Methoden der String-Klasse zusammengestellt. Es handelt sich dabei um Methoden zur **Länge**, zum **Suchen** sowie zum **Ersetzen** von Zeichen in Strings.

Methode	Wirkung
length()	liefert die Länge der Zeichenkette im aktuellen String-Objekt.
public int **indexOf**(String StrObj2)	sucht im aktuellen String-Objekt nach der Zeichenkette StrObj2 und liefert gegebenenfalls den kleinsten Index, an dem StrObj2 gefunden wird. Wird StrObj2 nicht gefunden, so ist der Rückgabewert gleich -1.
public int **indexOf**(String StrObj2, int start)	sucht beginnend bei Index start im aktuellen String-Objekt nach der Zeichenkette StrObj2 und liefert gegebenenfalls den kleinsten Index größer als start, an dem StrObj2 gefunden wird. Wird StrObj2 nicht gefunden, so ist der Rückgabewert gleich -1.
public int **lastIndexOf**(String StrObj2)	sucht im aktuellen String-Objekt nach der Zeichenkette StrObj2 und liefert gegebenenfalls den *größten* Index, an dem StrObj2 gefunden wird. Wird StrObj2 nicht gefunden, so ist der Rückgabewert gleich -1.
public int **lastIndexOf**(String StrObj2, int end)	sucht bis einschließlich Index end im aktuellen String-Objekt nach der Zeichenkette StrObj2 und liefert gegebenenfalls den *größten* Index *kleiner oder gleich* end, an dem StrObj2 gefunden wird. Wird StrObj2 nicht gefunden, so ist der Rückgabewert gleich -1.
public String **toUpperCase**()	ersetzt alle Kleinbuchstaben im aktuellen String-Objekt durch Großbuchstaben.
public String **toLowerCase**()	ersetzt alle Großbuchstaben im aktuelle String-Objekt durch Kleinbuchstaben.
public String **replace**(char oldChar, char newChar)	ersetzt im aktuellen String-Objekt an allen Stellen das Zeichen oldChar durch das Zeichen newChar.

Die Methode length() wird in folgender Anweisung des Programms bsp02053.StringVerarbeitung verwendet:

```
ScreenOut.kPrintln(
  "Die Länge der Zeichenkette \""+s3+"\" ist gleich: "+s3.length()+".");
```

Man beachte bei diesem Beispiel, dass die beiden Zeichen `\"` zusammen innerhalb einer Zeichenkette bei der Ausgabe die doppelten Hochkommata ergeben:

Mit Hilfe der folgenden Anweisungen des Programms bsp02053.StringVerarbeitung wird das String-Objekt s3 bzw. die in ihm gespeicherte Zeichenkette "Hallo Welt" untersucht:

```
    char[] c1 = {'H','a','l','l','o',' ','W','e','l','t','!'};
    String s3 = new String(c1);
...
    int found1, found2, found3, found4;
    found1 = s3.indexOf("Welt");          // (8)
```

```
...
    found2 = s3.indexOf("l",5);           // (9)
...
    found3 = s3.lastIndexOf("l");         // (10)
...
    found4 = s3.lastIndexOf("l",5);       // (11)
...
```

Die Anweisungen erklären sich wie folgt:

- Anweisung (8) sucht das Auftreten der Zeichenkette "Welt" mit dem niedrigsten Index und gibt den Index für den ersten Buchstaben W aus. bsp02053.StringVerarbeitung ermittelt als Index hierzu den Wert 6.
- Anweisung (9) ermittelt das erste Auftreten der nur aus einem einzigen Schriftzeichen bestehenden Zeichenkette "l" mit einen Index größer als 5. bsp02053.StringVerarbeitung liefert hierzu den Wert 8.
- Anweisung (10) bestimmt das Auftreten der Zeichenkette "l" mit dem höchsten Index, an dem die Zeichenkette beginnt. Ergebnis ist hier wieder der Index 8.
- Anweisung (11) ergibt schließlich den höchsten Index kleiner oder gleich 5, an dem die Zeichenkette "l" beginnt.

Die Anweisungen zu den letzten drei Methoden der obigen Tabelle erklären sich von selbst. bsp02053.StringVerarbeitung enthält hierzu den folgenden Programmcode:

```
    char[] c1 = {'H','a','l','l','o',' ','W','e','l','t','!'};
    String s3 = new String(c1);
...
    ScreenOut.kPrintln(
      s3+" in Großbuchstaben: "+s3.toUpperCase());
    ScreenOut.kPrintln(
      s3+" in Kleinbuchstaben: "+s3.toLowerCase());
    ScreenOut.kPrintln(
      s3+" verunstaltet: "+s3.replace('l','s'));
```

Dieser Java Code ergibt folgenden Bildschirmausdruck:

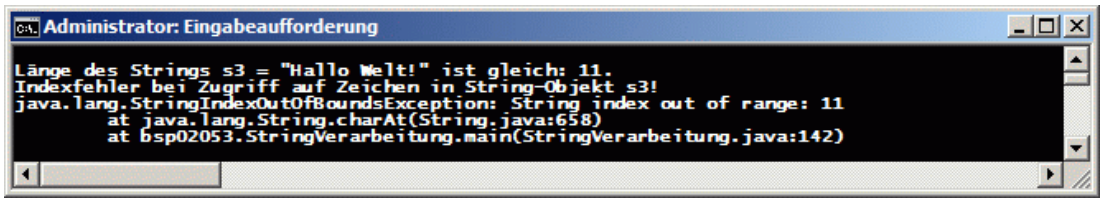
2.6.3.6 Laufzeitfehler

Wie bei jeder anderen Klasse auch erhält man bei der String-Verarbeitung eine **NullPointerException**, wenn man auf ein String-Objekt zugreift, das nicht instantiiert ist. Da String-Objekte nach ihrer Instantiierung nicht mehr verändert werden können und da String-Objekte in der Regel durch Angabe eines String-Literals angegeben werden, treten bei der Verarbeitung von Zeichenketten in der Regel keine weiteren Laufzeitfehler auf.

Die einzige Ausnahme von dieser Regel bildet die **StringIndexOutOfBoundsException**. Diese Ausnahme tritt auf bei der Verwendung der Methode charAt(...) sowie anderer Methoden, die durch Angabe eines Index auf ein einzelnes Schriftzeichen der Zeichenkette zugreifen. Wird dabei ein Index kleiner als Null oder größer gleich der Länge der Zeichenkette angegeben, so wird die StringIndexOutOfBoundsException ausgelöst bzw. 'geworfen'. Besteht die Gefahr einer solchen Situation, kann - wie im Programm bsp02053.StringVerarbeitung codiert - wie folgt verfahren werden:

```
    ScreenOut.kPrintln(
      "Länge des Strings s3 = \""+s3+"\" ist gleich: "+s3.length()+".");
    try
    {
      s3.charAt(s3.length());       // (12) Index um eins zu hoch
    }
    catch (Exception e)             // (13)
    {
      ScreenOut.kPrintln(
        "Indexfehler bei Zugriff auf Zeichen in String-Objekt s3!");
      e.printStackTrace();          // (14)
    }
```

In Zeile (12) wird durch Angabe eines zu hohen Index-Wertes die StringIndexOutOfBoundException geworfen. In Zeile (13) wird diese Ausnahme gefangen. Genauer gesagt wird in dieser Zeile dem von der Fehlerroutine erzeugten StringIndexOutOfBoundException-Objekt der Name e gegeben. In Anweisung (14) wird dann die Methode printStackTrace() verwendet, um die im StringIndexOutOfBoundException-Objekt enthaltene Fehlermeldung in der Konsolbox auszugeben. Der folgende Bildschirmausdruck zeigt die Ausgabe des Programms bsp02053.StringVerarbeitung nach Verarbeitung des obigen Programmteils:



2.6.3.7 Konvertierung in einen String

Oft wird man bei der Codierung von Programmen vor die Aufgabe gestellt, den Inhalt von Variablen einfacher Datentypen oder aber komplexerer Objekte in eine Zeichenkette umzuwandeln. Hierzu gibt es für viele Datentypen in der Klasse String eine jeweils passende *statische* Methode mit dem Namen valueOf(...):

Methode	Wirkung
public static String **valueOf** (boolean b)	Umwandlung des Inhaltes der Variablen b in die Zeichenkette "true" oder "false".
public static String **valueOf** (char c)	Umwandlung eines char Zeichens in einen String aus genau diesem Zeichen.
public static String **valueOf** (char[ ] c)	Umwandlung der Zeichen eines char Arrays in ein String-Objekt.
public static String **valueOf** (char[ ] c, int offset, int length)	Umwandlung von length vielen Zeichen des char Arrays c ab Index offset in ein String-Objekt.
public static String **valueOf** (byte b)	Umwandlung einer byte Zahl in ein String-Objekt, das aus den Ziffer der Zahl besteht.
public static String **valueOf** (short s)	Umwandlung einer short Zahl in ein String-Objekt, das aus den Ziffer der Zahl besteht.
public static String **valueOf** (int i)	Umwandlung einer int Zahl in ein String-Objekt, das aus den Ziffer der Zahl besteht.
public static String **valueOf** (long l)	Umwandlung einer long Zahl in ein String-Objekt, das aus den Ziffer der Zahl besteht.
public static String **valueOf** (float f)	Umwandlung einer float Zahl in ein String-Objekt, das aus den Ziffer und weiteren Zeichen der Darstellung der float Zahl besteht.
public static String **valueOf** (double d)	Umwandlung einer double Zahl in ein String-Objekt, das aus den Ziffer und weiteren Zeichen der Darstellung der double Zahl besteht.

Die Anwendung dieser Methoden zeigt der folgende Teil des Programms bsp02053.StringVerarbeitung:

```
String[] Ausgabe = new String[10];
boolean bo1 = true;
Ausgabe[0] = String.valueOf(bo1);
char ch1 = 'a';
Ausgabe[1] = String.valueOf(ch1);
byte bt1 = 65;
Ausgabe[2] = String.valueOf(bt1);
short sh1 = 65;
Ausgabe[3] = String.valueOf(sh1);
int i1 = 65;
Ausgabe[4] = String.valueOf(i1);
long l1 = 65l;
Ausgabe[5] = String.valueOf(l1);
float f1 = 6.5f;
Ausgabe[6] = String.valueOf(f1);
double d1 = 6.5d;
Ausgabe[7] = String.valueOf(d1);
char ca1[] = {'H','i',' ','W','o','r','l','d','!'};
Ausgabe[8] = String.valueOf(ca1);
Ausgabe[9] = String.valueOf(ca1,3,5);
for (int i=0;i<Ausgabe.length;i++)
  ScreenOut.kPrintln(Ausgabe[i]);
ScreenOut.kPrintln();
```

Die Umwandlung der Inhalte der einzelnen Variablen in den Datentyp String ermöglichte in obigem Programmcode die Ausgabe mit Hilfe einer for-Schleife.

2.6.3.8 Export aus String Objekten

In 2.6.3.7 wurde die Methoden vorgestellt, die verwendet werden können, um Inhalte von Variablen in den Datentyp String umzuwandeln.

Auch der umgekehrte Weg kann erforderlich sein. Hierzu verwendet man stellenweise die sogenannten Hüllklassen:

Hüllklassen sind Klassen, die Datentypen 'umhüllen' und Methoden zur Verarbeitung einfacher Datentypen bereitstellen. Zu jedem einfachen Datentyp gibt es eine Hüllklasse, wie die folgende Tabelle zeigt:

Datentyp	boolean	char	byte	short	int	long	float	double
Hüllklasse	Boolean	Character	Byte	Short	Integer	Long	Float	Double

In der folgenden Tabelle werden bis auf eine Ausnahme stets Hüllklassen bei der Umformungen von String in einen elementaren Datentyp verwendet. Stellenweise sind für die Umformung auch zwei Methoden unterschiedlicher Klassen erforderlich. Daher wird in der folgenden Tabelle jeweils der Ausdruck aufgeführt, der die entsprechende Umwandlung ergibt:

Ausdruck	**Wirkung**
Boolean.valueOf(StrObj1).booleanValue()	Aus dem String-Objekt StrObj1 wird zunächst ein Objekt der Klasse Boolean erzeugt, das jedoch keinen Namen erhält. Mit der Methode booleanValue() wird anschließend der Wahrheitswert aus dem Boolean-Objekt ermittelt.
StrObj1.toCharArray()[0]	Mit der Methode toCharArray wird die Zeichenkette im String-Objekt in ein char Array kopiert, das keinen Namen erhält. Aus diesem char Array wählt man dann das 0-te Element.
Byte.parseByte(StrObj1)	Die statische Methode parseByte(String) der Hüllklasse Byte liefert den byte Zahlwert der Zifferfolge im String-Objekt StrObj1.
Short.parseShort(StrObj1)	Die statische Methode parseShort(String) der Hüllklasse Short liefert den short Zahlwert der Zifferfolge im String-Objekt StrObj1.
Integer.parseInt(StrObj1)	Die statische Methode parseInt(String) der Hüllklasse Integer liefert den int Zahlwert der Zifferfolge im String-Objekt StrObj1.
Long.parseLong(StrObj1)	Die statische Methode parseLong(String) der Hüllklasse Long liefert den long Zahlwert der Zifferfolge im String-Objekt StrObj1.
Float.parseFloat(StrObj1) *bzw.* Float.valueOf(StrObj1).floatValue()	Aus dem String-Objekt StrObj1 wird zunächst ein Objekt der Klasse Float erzeugt, das jedoch keinen Namen erhält. Mit der Methode floatValue() wird anschließend der float Zahlwert aus dem Float-Objekt ermittelt.
Double.parseDouble(StrObj1) *bzw.* Double.valueOf(StrObj1).doubleValue()	Aus dem String-Objekt StrObj1 wird zunächst ein Objekt der Klasse Double erzeugt, das jedoch keinen Namen erhält. Mit der Methode doubleValue() wird anschließend der double Zahlwert aus dem Double-Objekt ermittelt.

Als Beispiel werden die in den Codezeilen des Beispiels zu 2.6.3.6 nach String umgewandelten Daten wieder in ihren ursprünglichen Datentyp zurück verwandelt. Dies geschieht in folgendem Teil des Programms bsp02053.StringVerarbeitung:

```
    boolean bo2 = Boolean.valueOf(Ausgabe[0]).booleanValue();
    ScreenOut.kPrintln(bo1 == bo2);
    char    ch2 = Ausgabe[1].toCharArray()[0];
    ScreenOut.kPrintln(ch1 == ch2);
    byte    bt2 = Byte.parseByte(Ausgabe[2]);
    ScreenOut.kPrintln(bt1 == bt2);
    short   sh2 = Short.parseShort(Ausgabe[3]);
    ScreenOut.kPrintln(sh1 == sh2);
    int      i2 = Integer.parseInt(Ausgabe[4]);
    ScreenOut.kPrintln(i1 == i2);
    long     l2 = Long.parseLong(Ausgabe[5]);
    ScreenOut.kPrintln(l1 == l2);
    float    f2 = Float.valueOf(Ausgabe[6]).floatValue();
    ScreenOut.kPrintln(f1 == f2);
    double   d2 = Double.valueOf(Ausgabe[7]).doubleValue();
    ScreenOut.kPrintln(d1 == d2);
    char  ca2[] = Ausgabe[8].toCharArray();
    boolean cArrayGleichheit = true;
    int index = 0;
    do
    {
// Zuweisung eines Wahrheitswertes zu einer boolean Variablen:
      cArrayGleichheit = ca1[index] == ca2[index];
      index++;
    } while ((cArrayGleichheit == true) & (index < ca1.length));
    ScreenOut.kPrintln(cArrayGleichheit);
```

In diesem Teil von bsp02053.StringVerarbeitung wird zur Kontrolle der Inhalt vor und nach der zweifachen Konvertierung verglichen und der Wahrheitswert des Vergleichs auf dem Konsolbildschirm ausgegeben. Man beachte dabei die Gleichheitsprüfung für die char Arrays ca1 und ca2. Hier wird für jeden Index der Wahrheitswert, der Ergebnis des Vergleichs für den Index ist, auf einer boolean Variablen gespeichert und die Prüfung so lange wiederholt, bis alle Indizes durchlaufen sind oder bis der erste Unterschied gefunden wurde.

2.6.3.9 Exceptions beim Export aus String Objekten

Istbeim Export aus String Objekten die Umwandlung in den Zahldatentyp nicht möglich, so tritt eine Exception vom Typ `NumberFormatException` auf. Hierbei handelt es sich um eine sogenannte *stille Exception*, das heißt der Compiler weist nicht darauf hin, dass an dieser Stelle diese Exception auftreten kann. Dennoch ist es in aller Regel sinnvoll und erforderlich, das Auftreten dieser Exception im Quellcode zu berücksichtigen.

Das generelle Vorgehen bei der Behanldung von Exceptions im Quellcode wird in Kapitel 2.9.5 erklärt. Für den vorliegenden Fall muss ein Programmabschnitt gemäß folgendem Muster erzeugt werden:

```
try
{
  ... Anweisung zum Export aus String Objekt gemäß Kap. 2.6.3.8
}
catch (NumberFormatException e)
{
  ... Anweisungen, die bei Auftreten der Exception ausgeführt werden sollen
}
```

Das konkrete Vorgehen zum Beispiel für die Umwandlung des Inhalts eines String-Objektes in eine double Größe zeigt das Programm `jpr02153`. Das Programm enthält den folgenden Quellcode:

```
// Einlesen einer Benutzereingabe als Text
    ScreenOut.kPrint("\nGeben Sie eine Zahl ein: ");
    String ein;
    ein = KeyboardIn.kReadLine();

// Umwandlung der Eingabe in eine double Zahl - sofern möglich
    double zahl=0;
    boolean ok = true;
    try
    {
      zahl = Double.parseDouble(ein);
    }
    catch (NumberFormatException e)
    {
      ScreenOut.kPrintln("\nFehler - die Eingabe konnte nicht in eine double Zahl umgewandelt
werden!");
// Fehlermeldung zerscheiden bei allen doppelten Hochkommata
      StringTokenizer msg = new StringTokenizer(e.getMessage(),"\"");
// erstes Token wegwerfen, das zweite Token enthält die Benutzereingabe:
      msg.nextToken();
      ScreenOut.kPrintln("Der Fehler ist aufgetreten für folgende Benutzereingabe:
\""+msg.nextToken()+"\"");
      ScreenOut.kPrintln("\nDer Java StackTrace lautet: ");
      e.printStackTrace();
      ok = false;
    }

// Ausgabe des Ergebnisses, sofern kein Fehler aufgetreten ist.
    if (ok)
      ScreenOut.kPrintln("\nDie Benutzereingabe ergibt die Zahl: "+zahl);
    else
      ScreenOut.kPrintln("\nDas Programm ergibt wegen Fehler kein Ergebnis.");
```

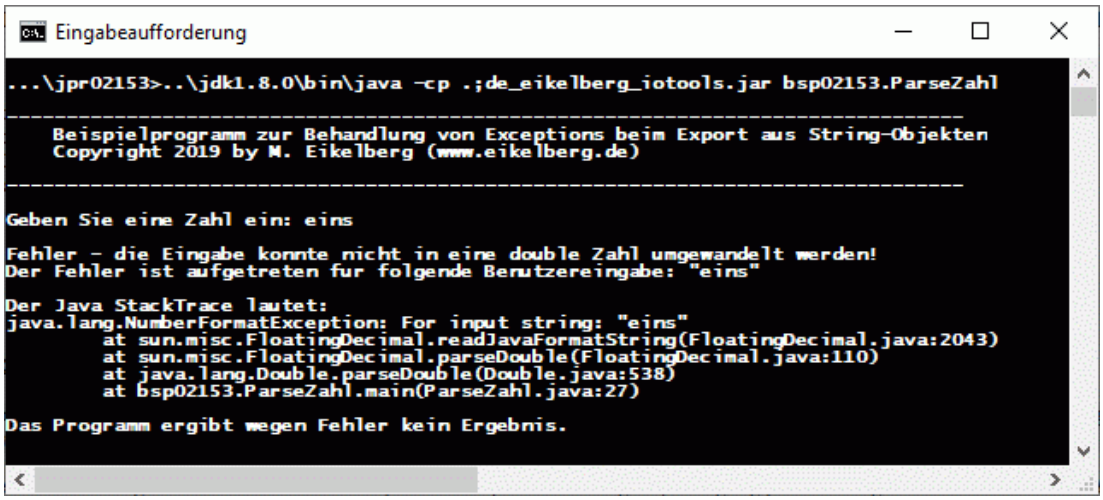
Hinweis: Im obigen Quelltext wird die Klasse `StringTokenizer` verwendet, um die von der Methode `e.getMessage()` gelieferte Fehlermeldung in mehrere Teile zu zerlegen. Die Klasse `StringTokenizer` wird in Kapitel 2.6.6.2 erklärt.

In obigem Abschnitt aus dem Beispielprogramm im Projekt `jpr02153` wird zur Umwandlung einer Zeichenkette in eine double Zahl wie folgt vorgegangen:

- Zunächst wird eine double-Variable mit dem Namen `zahl` für das Ergebnis der Umwandlung angelegt. Diese muss mit einem Wert initialisiert werden.
- Der Bereich, in dem die Exception auftreten kann, wird im try-Block codiert. Dieser enthält also die Anweisung für die Umwandlung der Zeichenkette in eine double Zahl.
- Auf den try-Block folgt ein Block, der beim Auftreten der Exception vom Typ `NumberFormatException` ausgeführt werden soll. Dieser Block folgt auf die Klausel `catch (NumberFormatException e)`.

- Für die Steuerung des Programms wird eine boolesche Variable mit dem Namen `ok` verwendet. Unter Verwendung dieser Variablen wird dann die Schlussmeldung des Programms festgelegt.

Beim Programmlauf erscheint sowohl bei einer erfolgreichen Umwandlung nach double als auch beim Auftreten eines Fehlers eine Schlussmeldung. Dies zeigt, dass auch im Fehlerfall das Programm nicht abstürzt, sondern kontrolliert beendet wird, wie der folgende Bildschirmausdruck zeigt:



2.6.4 StringBuffer-Objekte

2.6.4.1 Verwendungszweck von StringBuffer-Objekten

Wie in 2.6.3 bereits erwähnt kann der Inhalt von String-Objekten nach ihrer Instantiierung nicht mehr verändert werden. Oft wäre es jedoch wünschenswert, derartige Veränderungen vornehmen zu können. Eine solche **Veränderbarkeit von Zeichenketten** wird mit der Klasse **StringBuffer** bereitgestellt.

Zweckmäßig ist eine solche Funktionalität zum Beispiel beim Einlesen aus einer Datei unter Verwendung von Java Standardmethoden (vgl. Kapitel 2.6.6.4). Hierbei werden die Schriftzeichen aus der Eingabedatei einzeln eingelesen, analysiert und soweit sinnvoll an eine Zeichenkette angehängt, die sich in einem StringBuffer-Objekt befindet (vgl. Beispielprogramm bsp02060.DateiEinAus).

Das Ziel der Java-Entwickler bei der Codierung der Klasse StringBuffer-Objekten war also die Implementierung von veränderbaren Zeichenketten. Dementsprechend stellt die Klasse StringBuffer in erster Linie Methoden

- zur **Veränderung** von Zeichenketten,
- zum **Anhängen** an Zeichenketten,
- zum **Einfügen** in Zeichenketten sowie
- zur **Konvertierung** zwischen den Klassen String und StringBuffer

bereit. In Bezug auf die Funktionalität der Methoden ist die Schnittmenge der Klassen String und StringBuffer gering, das heisst die meisten Funktionalitäten findet man entweder in der Klasse String oder aber in der Klasse StringBuffer implementiert.

2.6.4.2 Konstruktoren der StringBuffer-Klasse

In der Klasse StringBuffer sind die folgenden drei Konstruktoren definiert:

Konstruktor	Wirkung
public StringBuffer()	Anlegen eines neuen StringBuffer-Objektes, das keine Zeichen enthält und das zunächst 16 Zeichen aufnehmen kann
public StringBuffer(int length)	Anlegen eines neuen StringBuffer-Objektes, das keine Zeichen enthält und zunächst length viele Zeichen aufnehmen kann
public StringBuffer(String StrObj1)	Anlegen eines neuen StringBuffer-Objektes, das mit den Zeichen des String-Objektes StrObj1 gefüllt ist und weitere 16 Zeichen aufnehmen kann.

Die Angabe der Anzahl von Zeichen, die ein StringBuffer-Objekt zunächst aufnehmen kann, ist in Bezug auf die Funktionalität ohne Interesse, denn der Speicherplatz für ein StringBuffer-Objekt wird automatisch geeignet erhöht, wenn der vorhandene Speicherplatz nicht mehr ausreicht. Lediglich die Verarbeitungsgeschwindigkeit eines Programms kann erhöht werden, wenn bei einer Verarbeitung einer großen Anzahl von StringBuffer-Objekten das nachträgliche Vergrößern der Speicherbereiche vermieden wird.

Die Konstruktoren der StringBuffer-Klasse werden wie die Konstruktoren anderer Klassen verwendet. Im folgenden Java Code aus dem Programm bsp02054.StringBufferVerarbeitung wird zum Beispiel ein StringBuffer-Objekt mit dem Namen `StrBuf1` (und mit zunächst 16 Zeichen Kapazität) wie folgt angelegt:

```
StringBuffer StrBuf1;
StrBuf1 = new StringBuffer();
```

2.6.4.3 Methoden der Klasse StringBuffer zur Bearbeitung des Inhaltes der Objekte

Die Klasse StringBuffer stellt in erster Linie Methoden

- zum Anhängen an die Zeichenkette in einem StringBuffer-Objekt sowie
- zum Einfügen in die Zeichenkette in einem StringBuffer-Objekt

bereit. Diese Methoden sind **überladen**, das heisst es gibt diese Methoden unter gleichem Namen zur Anwendung auf viele verschiedene Datentypen. Die folgende Tabelle stellt diese Methoden zusammen:

Methode	Wirkung
public StringBuffer **append**(boolean b) public synchronized StringBuffer **append**(char c) public synchronized StringBuffer **append**(char[] cArr) public synchronized StringBuffer **append**(char[] cArr,int off,int len)	Konvertieren der als Argumente angegebenen Daten nach String und Anfügen des String-Objektes an den Inhalt des StringBuffer-Objektes, auf das die Methode angewendet wird. append(char[] cArr,int off,int len) fügt den Inhalt des char Arrays cArr beginnend

public StringBuffer **append**(int i) public StringBuffer **append**(long l) public StringBuffer **append**(float f) public StringBuffer **append**(double d) public synchronized StringBuffer **append**(Class Obj1) public synchronized StringBuffer **append**(String StrObj1)	mit Offset off und über eine Länge von len vielen Zeichen an den StringBuffer an. append(Class Obj1) konvertiert den Inhalt des Objektes Obj1 nach String und fügt diesen String an.
public StringBuffer **insert**(int indx,boolean b) public synchronized StringBuffer **insert**(int indx,char c) public synchronized StringBuffer **insert**(int indx,char[] cArr) public StringBuffer **insert**(int indx,int i) public StringBuffer **insert**(int indx,long l) public StringBuffer **insert**(int indx,float f) public StringBuffer **insert**(int indx,double d) public synchronized StringBuffer **insert**(int indx,Class Obj1) public synchronized StringBuffer **insert**(int indx,String StrObj1)	Konvertieren der als Argumente angegebenen Daten nach String und Einfügen des String-Objektes bei Offset indx in den Inhalt des StringBuffer-Objektes, auf das die Methode angewendet wird. insert(int indx,Class Obj1) konvertiert den Inhalt des Objektes Obj1 nach String und fügt diesen String bei Offset indx - also nach dem indx-ten Zeichen bzw. *nach* dem Zeichen mit Index (indx-1) und *vor* dem Zeichen mit Index indx - ein.

Der Rückgabewert all dieser Methoden ist vom Typ StringBuffer. Daher kann man einen Aufruf der Methoden wie folgt codieren:

```
...
    StrBuf1 = StrBuf1.append(c);  // (1)
```

Die Methoden append(...) verändern jedoch bereits das StringBuffer-Objekt, zu dem sie aufgerufen werden. Daher ist auch folgende Codierung möglich, die die gleiche Wirkung hat wie die Codezeile (1):

```
    StrBuf1.append(c);            // (2)
```

Das Schlüsselwort `synchronized` bewirkt, dass die jeweilige Methode auch bei Implementierung paralleler Abläufe (Threads) stets nur ein einziges mal gleichzeitig ausgeführt ist. Für diesen Kurs ist das Schlüsselwort synchronized somit nicht von Bedeutung, da alle Programme des Kurses aus einem einzigen Thread bestehen.

Die folgende Tabelle stellt die übrigen Methoden der Klasse StringBuffer zusammen:

Methode	Wirkung
public int **length**()	gibt die Anzahl der gespeicherten Schriftzeichen im StringBuffer-Objekt zurück. Diese ist kleiner oder gleich der aktuellen Gesamtkapazität des StringBuffer-Objektes.
public synchronized void **setLength**(int newLength)	setzt die Anzahl der gespeicherten Schriftzeichen auf den Wert newLength. Die gespeicherte Zeichenkette wird dabei gekürzt oder mit Unicode-Zeichen x0000 (null character) aufgefüllt.
public int **capacity**()	gibt die Gesamtkapazität des StringBuffer-Objektes zurück. Bei Überschreiten der Kapazität wird das StringBuffer-Objekt vergrößert.
public synchronized void **ensureCapacity**(int minCapacity)	vergrößert die Gesamtkapazität des StringBuffer Objektes mindestens auf minCapacity viele Zeichen.
public synchronized char **charAt**(int indx)	liefert das Schriftzeichen mit Index indx, also das (indx+1)-te Zeichen des StringBuffer-Objektes.
public synchronized void **getChars**(int scrBegin,int scrEnd,char[] cArr,int dstBegin)	kopiert Schriftzeichen aus dem StringBuffer-Objekt in den char Array cArr. Dabei werden aus dem StringBuffer-Objekt die Zeichen ab Index scrBegin bis Index (srcEnd-1) herauskopiert und in cArr beginnend mit Index dstBegin gespeichert.
public String **toString**()	konvertiert den Inhalt des StringBuffer-Objektes in ein String-Objekt.
public synchronized void **setCharAt**(int indx,char c)	ersetzt das Zeichen des StringBuffer-Objektes mit Index indx durch das auf der Variablen c gespeicherte Zeichen.
public synchronized StringBuffer **reverse**()	kehrt die Reihenfolge der Schriftzeichen im StringBuffer-Objekt um.

Bei automatischer Vergrößerung eines StringBuffer-Objektes hat das StringBuffer-Objekt nach Vergrößerung eine neue Größe, die gleich dem "Doppelten der alten Größe plus 2" ist. Man kann dies leicht an folgendem Bildschirmausdruck zum Programm bsp02054.StringBufferVerarbeitung sehen:

```
Administrator: Eingabeaufforderung
...\jpr02054>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02054.StringBufferVerarbeitung
--------------------------------------------------------------------------------
    Beispielprogramm zur StringBuffer Verarbeitung
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Im Programm wird die Funktionalität 'Einlesen einer ganzen Zeile' (nicht im
Standard Java Sprachumfang enth.) fur das Einlesen aus Dateien implementiert.

Anzahl gespeicherter Zeichen: 0 / Kapazität des StringBuffer-Objektes: 16

Eingelesene Zeile: Dies ist ein Test-Text mit den
Anzahl gespeicherter Zeichen: 30 / Kapazität des StringBuffer-Objektes: 34

Eingelesene Zeile: Zahlen 12 3.4 5e-10 6.7e8, einem Zeilenwechsel
Anzahl gespeicherter Zeichen: 46 / Kapazität des StringBuffer-Objektes: 70

Eingelesene Zeile: sowie Sonderzeichen @ $%& §#*?
Anzahl gespeicherter Zeichen: 30 / Kapazität des StringBuffer-Objektes: 70

Dateiende erreicht !
```

Angelegt wurde das StringBuffer-Objekt im Programm bsp02054.StringBufferVerarbeitung mit dem in Kapitel 2.6.4.2 angeführten Programmcode, also unter Verwendung des Konstruktors `StringBuffer()` und somit mit einer ursprünglichen Kapazität von 16 Zeichen. bsp02054.StringBufferVerarbeitung liest die Zeilen einer Eingabedatei ein und speichert diese Zeilen in dem StringBuffer zwischen. Da die erste Zeile der Eingabedatei aus 30 Zeichen besteht, muss hier der StringBuffer erstmalig vergrößert werden auf die neue Größe 2*16+2 = 34.

In 2.6.4.4 wird das Programm bsp02054.StringBufferVerarbeitung und die in ihm enthaltenen Beispiele genauer erläutert:

2.6.4.4 Beispiel zur Anwendung der Klasse StringBuffer

Ausgangspunkt für die Programmierung des Programms bsp02054.StringBufferVerarbeitung war der folgende Sachverhalt: Bei der Verwendung von Standard Java Sprachmitteln ist es nicht möglich, mit Hilfe eines einzigen Befehls aus einer Eingabedatei eine ganze Zeile einzulesen. Vielmehr können lediglich einzelne Zeichen gelesen werden. Diese Zeichen müssen beim **Einlesen aus einer Eingabedatei mit Standard Sprachmitteln** wie folgt verarbeitet werden:

1. **Einlesen einzelner Zeichen**
2. Untersuchung der Zeichen und **Nachbildung der Zeilen der Eingabedatei** im Programm (u.a. unter Verwendung der Klasse StringBuffer)
3. **Zerlegen der Zeilen** in zusammenhängende Teile - die sogenannten **Token** (Worte, Zahlen etc.) - mit Hilfe der Klasse StringTokenizer (vgl. Kapitel 2.6.6.2)
4. **Umwandlung der Token** in den jeweiligen Datentyp zur Weiterverarbeitung im Programm.

Möchte man also das Einlesen aus einer Datei mit Standard Sprachmitteln realisieren, so benötigt man unter anderem einen Programmteil zum Schritt 2, also **zur Untersuchung der eingelesenen Schriftzeichen und zum Zusammenfügen dieser Schriftzeichen zu einer Zeile**. Bei diesem Verarbeitungsschritt wird die Klasse StringBuffer verwendet. Der entsprechende Programmcode ist im Beispielprogramm bsp02054.StringBufferVerarbeitung enthalten.

Im Programm bsp02054.StringBufferVerarbeitung erfolgt das Einlesen der einzelnen Schriftzeichen unter Verwendung der Methoden des Paketes de.eikelberg.iotools. Die Durchführung dieses 1. Verarbeitungsschrittes sowie die weitere Verarbeitung der eingelesenen Zeilen in Schritt 3 und 4 mit Standard Java Sprachmitteln wird in Kapitel 2.6.6 erklärt.

In Bezug auf StringBuffer-Verarbeitung im Rahmen der Realisierung von Schritt 2 enthält das Programm bsp02054.StringBufferVerarbeitung folgenden interessanten Programmcode:

```
...
// StringBuffer-Objekt anlegen
    StringBuffer StrBuf1;                                          // (1)
    StrBuf1 = new StringBuffer();                                  // (2)

// StringBuffer hat Platz für 16 Zeichen, enthält aktuell 0 Zeichen.
    ScreenOut.kPrint(
      "Anzahl gespeicherter Zeichen: "+StrBuf1.length()+" / ");    // (3)
    ScreenOut.kPrintln(
      "Kapazität des StringBuffer-Objektes: "+StrBuf1.capacity());  // (4)

// Verarbeitung bis Dateiende oder Fehler
    int status = 0;         // status-Variable für den Einlesevorgang
    char c = ' ';           // eingelesenes Zeichen als char Zeichen speichern
    StrBuf1.setLength(0);   // Länge StringBuffer gleich 0 setzen     // (5)
    while (status == 0)     // bei Dateiende wird status gleich 4 gesetzt
    {
// Einlesen eines einzelnen Zeichens
      c = dsIn1.dReadChar();
      status = dsIn1.dGetStatus();

// Anhängen des Zeichens, wenn nicht Dateiende, Return oder Newline
      if ((status == 0) & (c != '\r') & (c != '\n'))
```

```
      {
        StrBuf1.append(c);                                          // (6)
//        StrBuf1 = StrBuf1.append(c);   ist auch möglich           // (7)
      }

// Ist c gleich '\r' oder ist Dateiende erreicht, so ist die Zeile
// beendet. Ausgabe des StringBuffer-Objektes und Zurücksetzen der
// Länge des StringBuffer-Objektes auf 0:
      if ((c == '\r') || (status == 4))
      {
// Inhalt des StringBuffer-Objektes in ein String-Objekt kopieren
        String AusString1;
        AusString1 = new String(StrBuf1);                           // (8)
//        AusString1 = StrBuf1.toString(); ist auch möglich         // (9)

// Ausgabe auf dem Bildschirm
        ScreenOut.kPrintln();
        ScreenOut.kPrintln("Eingelesene Zeile: " + AusString1);     //(10)
// Ausgabe in die Ausgabedatei
        dsOut1.dPrintln("Eingelesene Zeile: " + AusString1);        //(10)

// Bufferlänge mit Methode setLength auf Null setzen:
        ScreenOut.kPrint(
          "Anzahl gespeicherter Zeichen: "+StrBuf1.length()+" / ");   // (3)
        ScreenOut.kPrintln(
          "Kapazität des StringBuffer-Objektes: "+StrBuf1.capacity());// (4)
        StrBuf1.setLength(0);
      }
    }
```

Im dem obigen Teil des Programms bsp02054.StringBufferVerarbeitung findet man folgende Beispiele zur StringBuffer-Verarbeitung:

1. Deklaration eines StringBuffer-Objektes
2. Instantiieren eines StringBuffer-Objektes mit einer Standard-Kapazität von 16 Zeichen.
3. Abfrage der Anzahl der im StringBuffer-Objekt gespeicherten Zeichen (Länge)
4. Abfrage der aktuellen Gesamtkapazität des StringBuffer-Objektes
5. Anzahl der im StringBuffer-Objekt gespeicherten Zeichen (auf 0) setzen
6. Anfügen eines Zeichens (gespeichert in der char Variablen c) an ein StringBuffer-Objekt (mit Namen StrBuf1)
7. Alternativer Programmcode zu (5) mit gleicher Wirkung. Man beachte dass die StringBuffer-Methoden auf den Inhalt des (aktuellen) StringBuffer-Objektes wirken, zu dem sie aufgerufen werden.
8. Kopieren des Inhaltes eines StringBuffer-Objektes in ein String-Objekt unter Verwendung eines Konstruktors der Klasse String.
9. Kopieren des Inhaltes eines StringBuffer-Objektes in ein String-Objekt unter Verwendung der Methode toString() der Klasse StringBuffer.
10. Automatische Typkonversion von StringBuffer nach String bei Verwendung der Verkettung von Strings mit dem +-Operator.

2.6.4.5 Konvertierung von und nach String

Bislang offen ist die Frage, wie Inhalte zweier StringBuffer-Objekte verglichen werden. Dieser Vergleich ist nicht mit Hilfe von Methoden der Klasse StingBuffer möglich, sondern basiert auf der Konvertierung der Inhalte von StringBuffer-Objekten in String-Objekte. Für die Ein- und Ausgabeverarbeitung benötigt man ebenfalls Konvertierungsfunktionen zwischen StringBuffer- und String-Objekten. Die folgende Tabelle listet einige Konvertierungsmöglichkeiten auf:

Klasse	Methode	Wirkung
StringBuffer	public **StringBuffer**(String StrObj1)	Konstruktor legt ein neues StringBuffer-Objekt an, das mit den Zeichen des String-Objektes StrObj1 gefüllt ist und weitere 16 Zeichen aufnehmen kann.
StringBuffer	public String **toString**()	konvertiert den Inhalt des StringBuffer-Objektes in ein String-Objekt.
String	public **String**(StringBuffer buffer)	Konstruktor legt ein neues String-Objekt an und füllt dieses mit der im StringBuffer-Objekt buffer gespeicherten Zeichenkette.
String	public static String **valueOf**(Class Obj1)	Wendet man diese Methode auf StringBuffer-Objekte an, so wird die Zeichenkette im StringBuffer-Objekt in ein String-Objekt kopiert. Dieses String-Objekt ist Rückgabewert der Methode.

Drei der vier genannten Methoden werden im Beispielprogramm bsp02045.GleichheitStringBuffer verwendet:

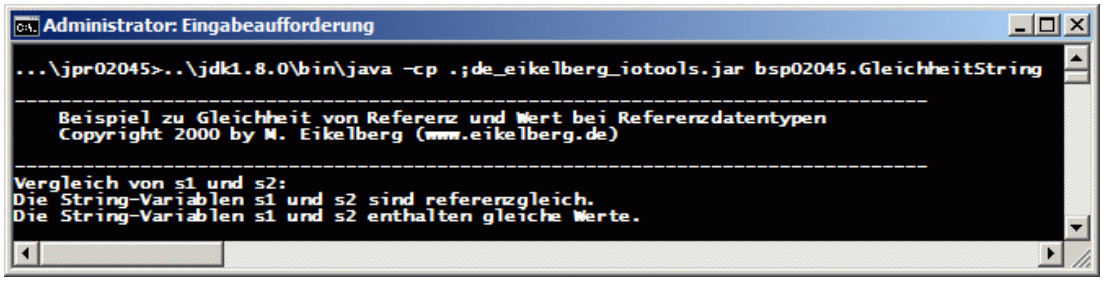
```
...
public class GleichheitStringBuffer
{
```

```
  public static void main(String[] args)
  {
...
    String s1 = new String("Hallo Welt");
...
    StringBuffer sb1 = new StringBuffer(s1);     // (11)
    StringBuffer sb2 = sb1;                      // (12)
...
// Vergleich der Inhalte der StringBuffer Variablen
    vergleicheStringBuffer(sb1,sb2,"sb1","sb2");
...
  }

  private static void vergleicheStringBuffer(
    StringBuffer sb1, StringBuffer sb2, String name1, String name2)
  {
...
    String s1 = new String();
    s1 = sb1.toString();                         // (13)
    String s2 = new String();
// s2 = sb2.toString(); - hierzu äquivalent ist die Anweisung
    s2 = String.valueOf(sb2);                    // (14)
// Anschließend erfolgt im Programm der Vergleich
// der Inhalte von s1 und s2
...
  }
...
```

In Bezug auf die StringBuffer-Verarbeitung sind folgende Anweisungen des Programms bsp02045.StringBufferVerarbeitung interessant:

11. Deklaration und Instantiierung eines StringBuffer-Objektes mit Namen sb1, das mit der Zeichenkette im String-Objekt s1 gefüllt wird.
12. Deklaration eines StringBuffer-Objektes sb2 und Referenzzuweisung des Objektes zu sb1. Die StringBuffer-Objekte sb1 und sb2 sind also gleich auf Referenzebene.
13. Kopie des Inhaltes des StringBuffer-Objektes sb1 in ein String-Objekt s1 unter Verwendung der Methode toString() der Klasse StringBuffer.
14. Kopie des Inhaltes des StringBuffer-Objektes sb1 in ein String-Objekt s1 unter Verwendung der Methode valueOf() der Klasse String.

Die Anweisungen (13) und (14) sind also äquivalent. Ferner sind die StringBuffer-Objekte sb1 und sb2 auf Referenzebene gleich, wie auch die Ausgabe des Programms in der Konsolbox zeigt:



2.6.4.6 Vergleich der Inhalte von StringBuffer-Objekten

Wir betrachten jetzt die Frage, wie StringBuffer-Objekte inhaltlich verglichen werden können. Wie bereits in Kapitel 2.6.1 erklärt gehören alle Klassen und damit auch die StringBuffer-Klasse zu den Referenzdatentypen. Die Operatoren "=", "==" und "!=" beziehen sich somit jeweils auf die Referenzen. Spezielle Methoden zum Vergleich der Inhalte von StringBuffer-Objekten gibt es nicht. Ein Vergleich von StringBuffer-Objekten kann jedoch wie folgt durchgeführt werden:

1. Kopieren der Inhalte der zu vergleichenden StringBuffer-Objekte in String-Objekte (vgl. 2.6.4.5)
2. Vergleich der Inhalte der String-Objekte (vgl. 2.6.3.3)

Ein solches Vorgehen ist im Programm bsp02045.GleichheitStringBuffer codiert:

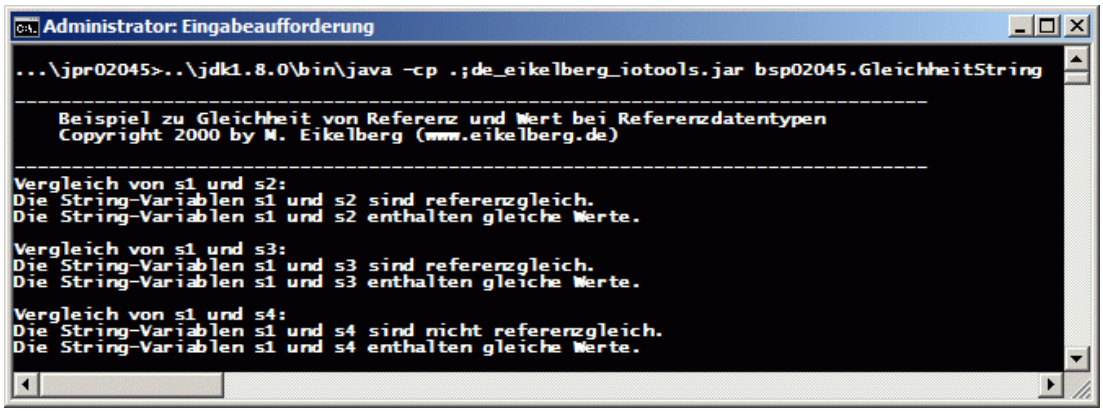
```
...
public class GleichheitStringBuffer
{

  public static void main(String[] args)
  {
...
    String s1 = new String("Hallo Welt");        // (15)
...
    StringBuffer sb1 = new StringBuffer(s1);     // (16)
```

```
...
    StringBuffer sb3 = new StringBuffer(s1);      // (16)
...
// Vergleich von sb1 und sb3
    ScreenOut.kPrintln("Vergleich von sb1 und sb3 mit Methode equals():");
    if (sb1 == sb3)                               // (17)
      meldeReferenzGleichheit("sb1","sb3");
    else
      meldeReferenzUngleichheit("sb1","sb3");
//  Vergleich der Inhalte der StringBuffer Variablen
    vergleicheStringBuffer(sb1,sb3,"sb1","sb3"); // (18)
...
  }

  private static void vergleicheStringBuffer(
    StringBuffer sb1, StringBuffer sb3,
    String name1, String name2)                   // (19)
  {
// Zum Vergleich konvertiert man die Inhalte von StringBuffer Variablen
// mit der toString() nach String und vergleicht die String-Variablen
    String s1 = new String();
    s1 = sb1.toString();                          // (20)
    String s3 = new String();
//  s3 = sb3.toString(); - hierzu äquivalent ist die Anweisung
    s3 = String.valueOf(sb3);                     // (20)
    if (s1.equals(s3))                            // (21)
      meldeWertGleichheit(name1,name2);
    else
      meldeWertUngleichheit(name1,name2);
    ScreenOut.kPrintln();
  }
...
```

Die markierten Anweisungen bewirken folgende Verarbeitung:

15. Deklaration und Instantiierung eines String-Objektes s1, das die Zeichenkette "Hallo Welt" enthält.
16. Deklaration und Instantiierung zweier StringBuffer-Objekte sb1 und sb3, die mit der Zeichenkette im String-Objekt s1 gefüllt werden.
17. Vergleich von sb1 und sb3 auf Referenzgleichheit.
18. Aufruf der Methode vergleicheStringBuffer(...) zum Vergleich der StringBuffer sb1 und sb3
19. Deklaration der Methode vergleicheStringBuffer(...).
20. Kopieren der Inhalte von sb1 und sb3 in String-Objekte s1 und s3 wie in 2.6.4.5 erklärt.
21. Vergleich der Inhalte der String-Objekte s1 und s3

Da die StringBuffer-Objekte sb1 und sb3 jeweils ausdrücklich instantiiert wurden, sind sie ungleich auf Referenzebene, jedoch inhaltlich gleich, wie der folgende Bildschirmausdruck zeigt:



2.6.4.7 Laufzeitfehler

Bei der Verarbeitung von StringBuffer-Objekten treten im Prinzip ähnliche Fehler auf wie bei der Verarbeitung von Arrays oder String-Objekten.

Legt man dem StringBuffer-Objekte mit dem Konstruktor `StringBuffer(n)` an und gibt man dabei eine Länge `n<0` an, so erhält man eine **NegativeArraySizeException**. Eine **StringIndexOutOfBoundsException** erhält man bei falscher Anwendung der insert(..)-Methoden sowie bei falscher Anwendung der Methoden setLength(..), charAt(..), setCharAt(..) und getChars(..).

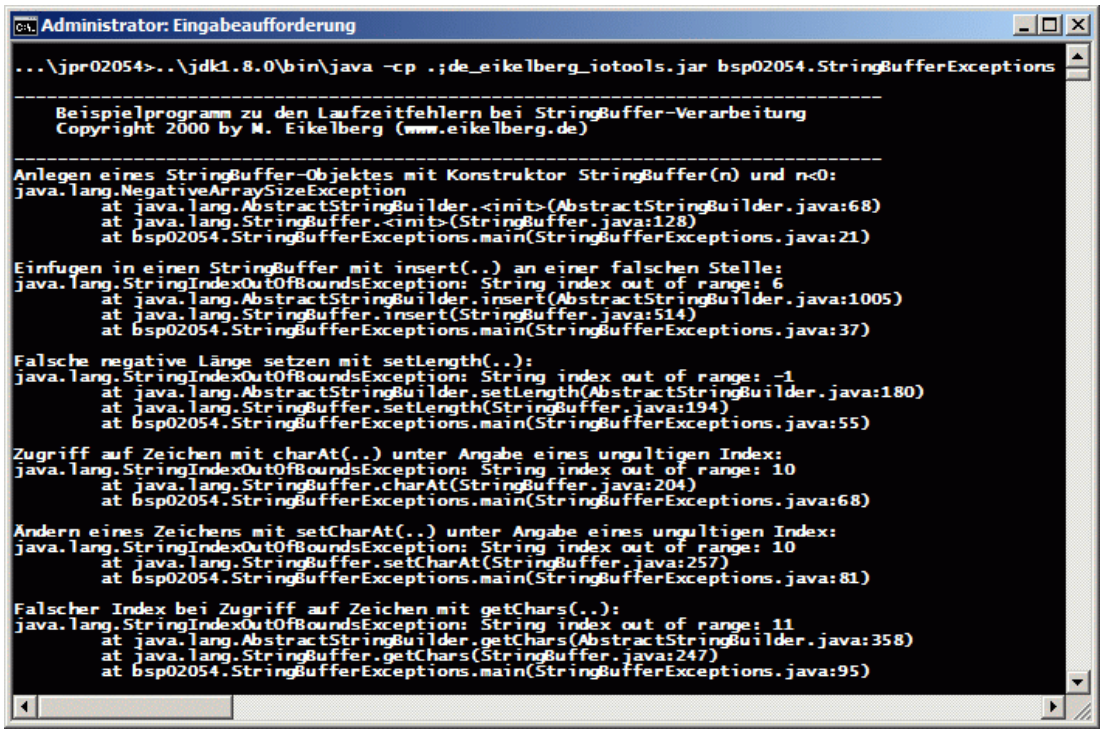
Im Beispielprogramm bsp02054.StringBufferExceptions werden entsprechende falsche Aufrufe von StringBuffer-Methoden codiert, wobei die Laufzeitfehler (Exceptions) mit Hilfe von `try ... catch` abgefangen werden. Zu jedem Laufzeitfehler wird die entsprechende Fehlermeldung in der Konsolbox ausgegeben und somit der jeweilige Fehler sichtbar gemacht. Das Programm

bsp02054.StringBufferExceptions enthält folgenden Programmcode:

```
...
public class StringBufferExceptions
{

  public static void main(String[] args)
  {
...
// Anlegen eines StringBuffer-Objektes mit Konstruktor StringBuffer(n) und n<0
    ScreenOut.kPrintln(
      "Anlegen eines StringBuffer-Objektes mit Konstruktor StringBuffer(n) und n<0:");
    StringBuffer StrBuf1;
    try
    {
      StrBuf1 = new StringBuffer(-1);
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

// Korrektes Anlegen von StrBuf1
    StrBuf1 = new StringBuffer("Hallo");

// Einfügen in einen StringBuffer mit insert(..) an einer falschen Stelle
    ScreenOut.kPrintln();
    ScreenOut.kPrintln(
      "Einfügen in einen StringBuffer mit insert(..) an einer falschen Stelle:");
    try
    {
      StrBuf1.insert(6,"ihall"); // Länge ist 5, maximaler Offset somit gleich 5
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

// Korrektes Einfügen in StrBuf1 - ergibt Zeichenkette "Hallihallo"
   StrBuf1.insert(4,"ihall");    // Einfügen bei Offset 5, also nach Index 5-1=4
//   String s1 = new String(StrBuf1);
//   ScreenOut.kPrintln(s1);

// Falsche negative Länge setzen mit setLength(..)
    ScreenOut.kPrintln();
    ScreenOut.kPrintln(
      "Falsche negative Länge setzen mit setLength(..):");
    try
    {
      StrBuf1.setLength(-1);// Länge -1 ist ungültig
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

// Zugriff auf Zeichen mit charAt(..) unter Angabe eines ungültigen Index
    ScreenOut.kPrintln();
    ScreenOut.kPrintln(
      "Zugriff auf Zeichen mit charAt(..) unter Angabe eines ungültigen Index:");
    try
    {
      StrBuf1.charAt(10);   // Index geht von 0 bis 9
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

// Ändern eines Zeichens mit setCharAt(..) unter Angabe eines ungültigen Index
    ScreenOut.kPrintln();
    ScreenOut.kPrintln(
      "Ändern eines Zeichens mit setCharAt(..) unter Angabe eines ungültigen Index:");
```

```
    try
    {
      StrBuf1.setCharAt(10,'*');    // Index geht von 0 bis 9
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

// Falscher Index bei Zugriff auf Zeichen mit getChars(..)
    ScreenOut.kPrintln();
    ScreenOut.kPrintln(
      "Falscher Index bei Zugriff auf Zeichen mit getChars(..):");
    char[] cArr = new char[6];
    try
    {
      StrBuf1.getChars(5,11,cArr,0);    // Zugriff von 0 bis 10, Index jedoch <10 stets
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }

...
  }
}
```

Ein Durchlauf des Programms bsp02054.StringBufferExceptions ergibt folgende Bildschirmausgabe:



2.6.5 Ein- und Ausgabe von Arrays und String-Objekten

2.6.5.1 Einlesen und Ausgeben von Arrays mit de.eikelberg.iotools

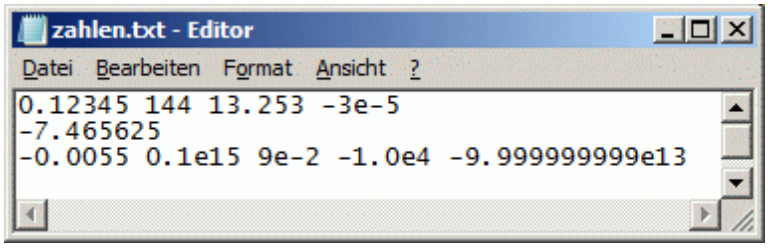
Wie bei dem Vergleich der Inhalte zweier Arrays erfolgt auch die Ein- und Ausgabe von Array Elementen durch Zugriff auf die Elemente des Arrays. Abgesehen von char Arrays gibt es weder im Standard Sprachumfang von Java noch im Paket de.eikelberg.iotools Sprachelemente für die Ein- und Ausgabe von Arrays.

Lediglich zu char Arrays gibt es Methoden, die einen ganzen Array verarbeiten können. Diese Methoden werden allerdings selten benötigt. Die folgende Tabelle stellt die Methoden des Paketes de.eikelberg.iotools zu char Arrays zusammen:

Methode	Klasse	Wirkung
public void **kPrint**(char x[ ])	ScreenOut	Schreiben der Elemente eines Arrays aus character Variablen.
public void **kPrintln**(char x[ ])	ScreenOut	Schreiben der Elemente eines Arrays aus character Variablen. Ferner Schreiben eines Zeilenwechsels am Ende des Schreibvorganges.
public void **kPrint**(char x[ ], int off, int len)	ScreenOut	Schreiben der Elemente eines Arrays aus character Variablen. Es werden beginnend bei Offset off die folgenden len Zeichen geschrieben.
public void **kPrintln**(char x[ ], int off, int len)	ScreenOut	Schreiben der Elemente eines Arrays aus character Variablen. Es werden beginnend bei Offset off die folgenden len Zeichen geschrieben. Ferner Schreiben eines Zeilenwechsels am Ende des Schreibvorganges.
public int **dReadChar**(char[ ] cbuf,int off,int len)	DatasetIn	Einlesen von len vielen Zeichen und Speichern dieser Zeichen in einem Array von character Zeichen, beginnend bei Offset off.
public void **dPrint**(char x[ ])	DatasetOut	Schreiben der Elemente eines Arrays aus character Variablen.
public void **dPrintln**(char x[ ])	DatasetOut	Schreiben der Elemente eines Arrays aus character Variablen. Ferner Schreiben eines Zeilenwechsels am Ende des Schreibvorganges.
public void **dPrint**(char x[ ], int off, int len)	DatasetOut	Schreiben der Elemente eines Arrays aus character Variablen. Es werden beginnend bei Offset off die folgenden len Zeichen geschrieben.
public void **dPrintln**(char x[ ], int off, int len)	DatasetOut	Schreiben der Elemente eines Arrays aus character Variablen. Es werden beginnend bei Offset off die folgenden len Zeichen geschrieben. Ferner Schreiben eines Zeilenwechsels am Ende des Schreibvorganges.

Im Allgemeinen erfolgt eine Ausgabe von Arrays eher für Zahlwerte. Für diesen Fall gibt es keine besonderen Methoden, was allerdings nicht problematisch ist, da die Ein- und Ausgabe leicht mit Hilfe einer Schleifenkonstruktion durchgeführt werden kann. Das entsprechende Vorgehen wird am Programm bsp02055.ArrayIO erläutert.

Das Einlesen der Daten in einen Array erfolgt im Programm bsp02055.ArrayIO zunächst wie im Beispielprogramm bsp02051.ArithmMittel (vgl. Kapitel 2.6.2.6). In der Eingabedatei steht also zuerst (in der ersten Zeile) die Anzahl der einzugebenden Zahlen, anschließend (beginnend mit der 2. Zeile) sind die Daten für die Elemente des Arrays eingetragen:



Diese Zahlen werden wie folgt eingelesen:

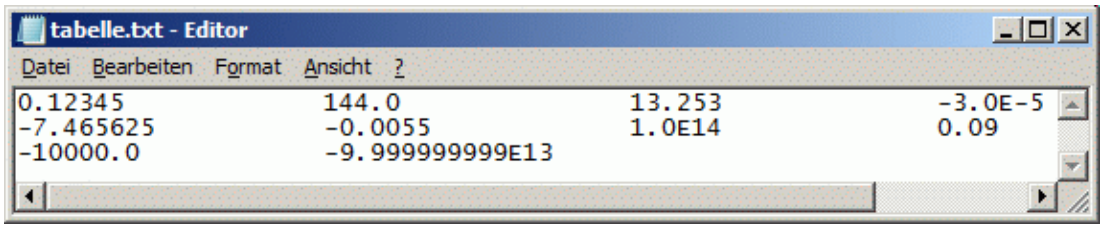
```
// 1. Eingabedatei öffnen
    DatasetIn Eingabe;
    Eingabe = new DatasetIn("daten\\zahlen.txt");

// 2. Anzahl der zu verarbeitenden Zahlen einlesen
    int anzahl = Eingabe.dReadInt();

// 3. Array für die einzulesenden Zahlen anlegen
    double werte[ ];
    werte = new double[anzahl];
```

```
// 4. Einlesen der zu verarbeitenden Zahlen
    int status = 0;
    for (int i=0; i<anzahl; i++)
    {
      werte[i] = Eingabe.dReadDouble();
      status = Eingabe.dGetStatus();
      if (status != 0)
      {
        ScreenOut.kPrintln(
          "Zu wenig Werte in der Eingabedatei - Einlesevorgang wird abgebrochen!");
        i=anzahl;   // Verarbeitung abbrechen
      }
    }

// 5. Eingabedatei schließen
    Eingabe.dClose();
```

Die Ausgabe der Elemente im Array `werte[ ]` ist im Prinzip sehr einfach möglich. Allerdings besteht oft die Notwendigkeit, die Elemente des Arrays bei der Ausgabe geeignet in Spalten und Zeilen anzuordnen. Daher wird in bsp02055.ArrayIO eine solche Anordnung vorgenommen - die Elemente des Arrays werden zu je vier Zahlen je Zeile ausgegeben und es werden dabei 4 Spalten mit je 20 Schriftzeichen je Spalte gebildet. Eine solche Anordnung kann wie folgt erreicht werden:

```
// 6. Ausgabedatei öffnen
    DatasetOut Ausgabe;
    Ausgabe = new DatasetOut("daten\\tabelle.txt");

// 7. formatierte Ausgabe der Zahlen im Array werte[ ], 4 Werte je Zeile
    for (int i=0; i<werte.length; i++)
    {
//    Ausgabe eines Elementes des Arrays
      Ausgabe.dPrint(werte[i]);
//    mit Leerzeichen so auffüllen, dass eine Tabelle entsteht
      String Zahl = String.valueOf(werte[i]);   // Zahl in Zeichenkette umwandeln
      int sp = 20-Zahl.length();                 // Anzahl der einzufügenden
                                                 // Leerzeichen berechnen
      StringBuffer Space = new StringBuffer(0); // StringBuffer Space anlegen mit
                                                 // einer Länge gleich 0
      for (int j=0; j<sp; j++)                   // StringBuffer geeignet auf Länge
        Space.append(" ");                       // sp vergrößern und mit
                                                 // Leerzeichen füllen
      Ausgabe.dPrint(Space.toString());          // StringBuffer nach String
                                                 // konvertieren und hinter Zahl
                                                 // ausgeben
//    nach jedem 4. Zeichen einen Zeilenwechsel einfügen
      if (i % 4 == 3) Ausgabe.dPrintln();
    }

// 8. Ausgabedatei schließen
    Ausgabe.dClose();
```

Die formatierte Ausgabe ordnet die aus der Eingabedatei eingelesenen Zahlen wie folgt an:



2.6.5.2 Einlesen und Ausgeben von String-Objekten mit de.eikelberg.iotools

Die Ein- und Ausgabe von String-Objekten ist ein verhältnismäßig häufiger Vorgang. Daher stellt das Paket de.eikelberg.iotools entsprechende Methoden bereit. Im Standard Java Sprachumfang gibt es ebenfalls wenige - jedoch wesentliche - Sprachelemente zur Verarbeitung von String-Objekten. Diese Elemente werden in Kapitel 2.6.6 vorgestellt.

Methode	Klasse	Wirkung
public static String **kReadLine()**	KeyboardIn	Einlesen einer Zeichenkette und Weitergabe der Zeichenkette als Rückgabewert der Methode. Ende der Eingabe / F6 und Zeilenwechsel werden als Ende der Zeichenkette interpretiert.

public static String **kReadString()**	KeyboardIn	Einlesen einer Zeichenkette und Weitergabe an das Programm mit Datentyp String. Leerzeichen, Ende der Eingabe, Zeilenwechsel, Carriage-Return und Return werden als Ende des Strings interpretiert.
public static String **kReadString**(char trennzeichen)	KeyboardIn	Einlesen einer Zeichenkette und Weitergabe an das Programm mit Datentyp String. Das beim Aufruf der Methode als Parameter mitgegebene Trennzeichen sowie Dateiende und Zeilenwechsel werden als Ende des Strings interpretiert.
public static void **kPrint**(String aus)	ScreenOut	Schreiben eines String-Literals oder des Inhaltes eines String-Objektes bei Bildschirmausgabe in eine Konsolbox.
public static void **kPrintln**(String aus)	ScreenOut	Schreiben eines String-Literals oder des Inhaltes eines String-Objektes bei Bildschirmausgabe in eine Konsolbox. Zum Abschluss des Schreibvorganges wird zusätzlich ein Zeilenwechsel geschrieben.
public static void **kPrint**(String aus, int off, int len)	ScreenOut	Schreiben einer Zeichenkette oder des Inhaltes einer String Variablen bei Bildschirmausgabe in eine Konsolbox. Es werden beginnend bei Offset off die folgenden len vielen Zeichen ausgegeben.
public static void **kPrintln**(String aus, int off, int len)	ScreenOut	Schreiben einer Zeichenkette oder des Inhaltes einer String Variablen bei Bildschirmausgabe in eine Konsolbox. Es werden beginnend bei Offset off die folgenden len vielen Zeichen ausgegeben. Zum Abschluss des Schreibvorganges wird zusätzlich ein Zeilenwechsel geschrieben.
public String **dReadLine()**	DatasetIn	Einlesen einer ganzen Zeile aus der Eingabedatei und Weitergabe an das Programm mit Datentyp String.
public String **dReadString()**	DatasetIn	Einlesen einer Zeichenkette und Weitergabe an das Programm mit Datentyp String. Leerzeichen, Dateiende, Zeilenwechsel, Carriage-Return und Return werden als Ende des Strings interpretiert.
public String **dReadString**(char trennzeichen)	DatasetIn	Einlesen einer Zeichenkette und Weitergabe an das Programm mit Datentyp String. Das beim Aufruf der Methode als Parameter mitgegebene Trennzeichen sowie Dateiende und Zeilenwechsel werden als Ende des Strings interpretiert.
public void **dPrint**(String x)	DatasetOut	Schreiben des Inhaltes eines String-Objektes x.
public void **dPrintln**(String x)	DatasetOut	Schreiben des Inhaltes eines String-Objektes x. Zum Abschluss des Schreibvorganges wird zusätzlich ein Zeilenwechsel geschrieben.
public void **dPrint**(String x, int off, int len)	DatasetOut	Schreiben des Inhaltes eines String-Objektes x. Es werden beginnend bei Offset off die folgenden len Zeichen geschrieben.
public void **dPrintln**(String x, int off, int len)	DatasetOut	Schreiben des Inhaltes eines String-Objektes x. Es werden beginnend bei Offset off die folgenden len Zeichen geschrieben. Zum Abschluss des Schreibvorganges wird zusätzlich ein Zeilenwechsel geschrieben.

Man beachte, dass in der Tabelle genau genommen zwei verschiedene Typen von Methoden zu finden sind:

- Methoden zur Verarbeitung des Inhaltes von String-Objekten.
- Methoden zur zeilenweisen Ein- und Ausgabe.

Diese beiden Methoden unterscheiden sich dadurch, dass das Ende der Zeilenkette jeweils unterschiedlich definiert ist:

- Bei der Verarbeitung von String-Objekten wird das **Ende des Strings** durch die Zeichen **Leerzeichen, Dateiende, Zeilenwechsel, Carriage-Return, Return oder ein besonders angegebenes Trennzeichen** angegeben.
- Bei der **zeilenweise**n Ein- und Ausgabe gelten dagegen nur **Dateiende, Zeilenwechsel, Carriage-Return oder Return** als Ende des einzugebenden Bereichs, die eingelesene Zeichenkette kann in diesem Fall also auch Leerzeichen enthalten. Derart eingelesene ganze Zeilen können mit Hilfe der Klasse StringTokenizer in ihre Bestandteile zerlegt werden. Anschließend können die einzelnen Bestandteile unter Verwendung der Methoden aus Kapitel 2.6.3.8 in die jeweiligen Datentypen umgeformt werden.

Die Verarbeitung ganzer Zeilen bei der Ein- und Ausgabe sowie ihre weitere Bearbeitung wird in Kapitel 2.6.6.1 erklärt. An dieser Stelle sind also zunächst die Methoden zur Ein- und Ausgabe von Zeichenketten zu erläutern. Beispiele zu diesem Thema findet man in den Programmen bsp02056.StringObjectIOKonsole und bsp02056.StringObjectIODataset:

Das Einlesen von Zeichenketten von der Tastatur und die Ausgabe von Zeichenketten auf dem Konsolbildschirm kann wie im folgenden Ausschnitt des Programms bsp02056.StringObjectIOKonsole codiert werden:

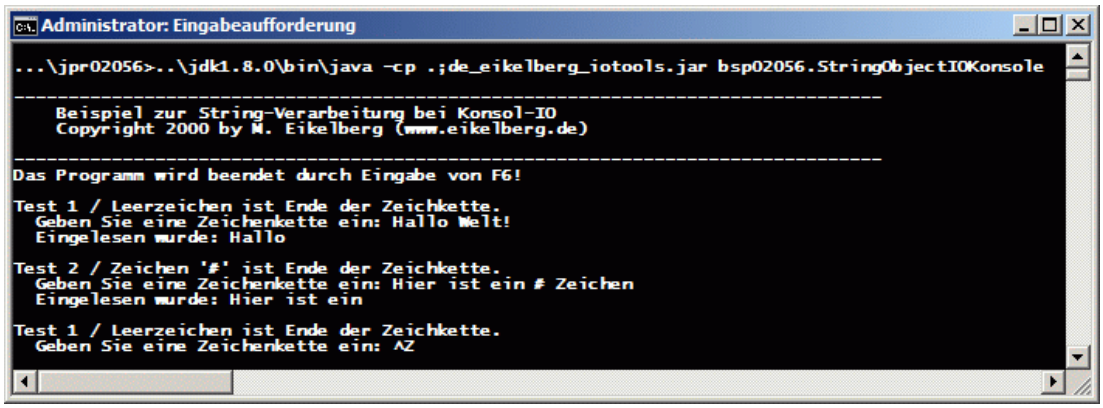
```
    int status = 0;
    do
    {
// 1   Einlesen einer Zeichenkette - Leerzeichen ist Ende der Zeichenkette
// 1.1 Eingabeaufforderung
```

```
        ScreenOut.kPrintln("Test 1 / Leerzeichen ist Ende der Zeichkette.");
        ScreenOut.kPrint("  Geben Sie eine Zeichenkette ein: ");
// 1.2 Einlesen der Zeichenkette mit Statusabfrage
        String St1;
        St1 = KeyboardIn.kReadString();
        status = KeyboardIn.kGetStatus();
// 1.3 Keine Fortsetzung der Verarbeitung bei Fehler oder Eingabe von F6
        if (status == 0)
        {
// 1.4 Rest der Eingabe verarbeiten
          KeyboardIn.kReadLine();
// 1.5 Ausgabe der eingelesenen Zeichenkette
          ScreenOut.kPrint("  Eingelesen wurde: ");
          ScreenOut.kPrintln(St1);
          ScreenOut.kPrintln();
...
    }
    while (status == 0);
```

Soll statt eines Leerzeichens ein anderes Zeichen das Ende einer Zeichenkette markieren, so kann die Programmierung der Ein- und Ausgabe wie in folgendem Programmcode geschehen. Dieser Java Code ist ebenfalls Teil des Programms bsp02056.StringObjectIOKonsole:

```
    int status = 0;
    do
    {
...
        String St1;
...
// 2   Einlesen einer Zeichenkette - '#' ist Ende der Zeichenkette
// 2.1 Eingabeaufforderung
          ScreenOut.kPrintln("Test 2 / Zeichen '#' ist Ende der Zeichkette.");
          ScreenOut.kPrint("  Geben Sie eine Zeichenkette ein: ");
// 2.2 Einlesen der Zeichenkette mit Statusabfrage
          St1 = KeyboardIn.kReadString('#');
          status = KeyboardIn.kGetStatus();
        }
// 2.3 Keine Fortsetzung der Verarbeitung  bei Fehler oder Eingabe von F6
        if (status == 0)
        {
// 2.4 Rest der Eingabe verarbeiten
          KeyboardIn.kReadLine();
// 2.5 Ausgabe der eingelesenen Zeichenkette
          ScreenOut.kPrint("  Eingelesen wurde: ");
          ScreenOut.kPrintln(St1);
          ScreenOut.kPrintln();
        }
    }
    while (status == 0);
```

Dieser Programmcode führt zu folgender Verarbeitung:

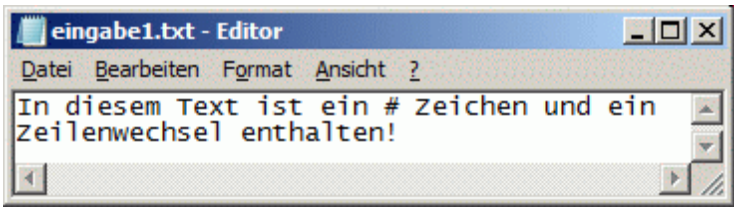


Bem Einlesen aus einer Datei entfallen einige der obigen Schritte, wie die folgenden Beispiele zeigen. Sie zeigen die den beiden obigen Programmstücken aus bsp02056.StringObjectIOKonsole entsprechende Dateiverarbeitung und sind Bestandteil des Programms bsp02056.StringObjectIODataset. Wir betrachten zunächst die Verarbeitung für den Fall, dass ein Leerzeichen das Ende einer Zeichenkette anzeigen soll:

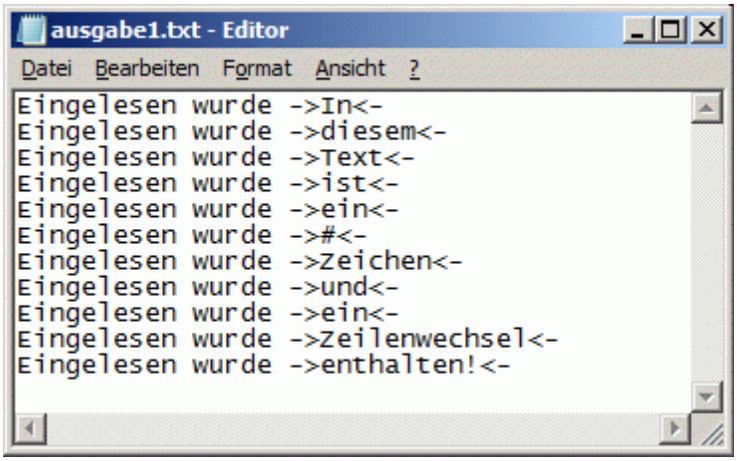
```
// Dateien öffnen
```

```
    DatasetIn Ein1;
    Ein1 = new DatasetIn("daten\\eingabe1.txt");
...
    DatasetOut Aus1;
    Aus1 = new DatasetOut("daten\\ausgabe1.txt");
...
// Schleife zur Verarbeitung von eingabe1.txt
    int status = 0;
    do
    {
// 1   Einlesen einer Zeichenkette - Leerzeichen ist Ende der Zeichenkette
// 1.1 Einlesen der Zeichenkette mit Statusabfrage
        String St1;
        St1 = Ein1.dReadString();
        status = Ein1.dGetStatus();
// 1.2 Keine Fortsetzung der Verarbeitung bei Fehler oder Dateiende
        if (status == 0)
        {
// 1.3 Ausgabe der eingelesenen Zeichenkette
          Aus1.dPrint("Eingelesen wurde ->");
          Aus1.dPrint(St1);
          Aus1.dPrintln("<-");
        }
    }
    while (status == 0);
```

Die folgenden beiden Bildschirmausdrucke zeigen den Inhalt der Dateien eingabe1.txt und ausgabe1.txt nach einem Programmlauf:



Die Leerzeichen sowie der Zeilenwechsel in der Eingabedatei führen zur Bildung der folgenden Zeichenketten:

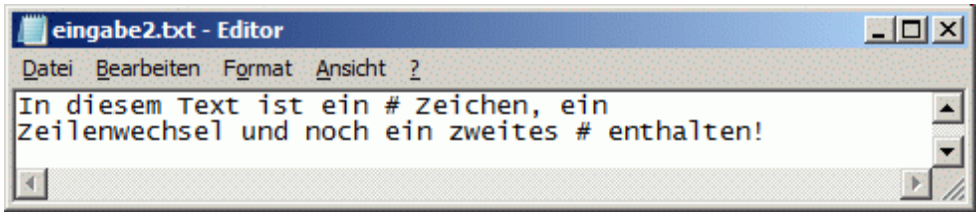


Wird statt eines Leerzeichens für die Bildung der Zeichenketten das Schriftzeichen '#' als Trennzeichen gewählt, so kann die Ein- und Ausgabe der Zeichenketten wie folgt codiert werden:

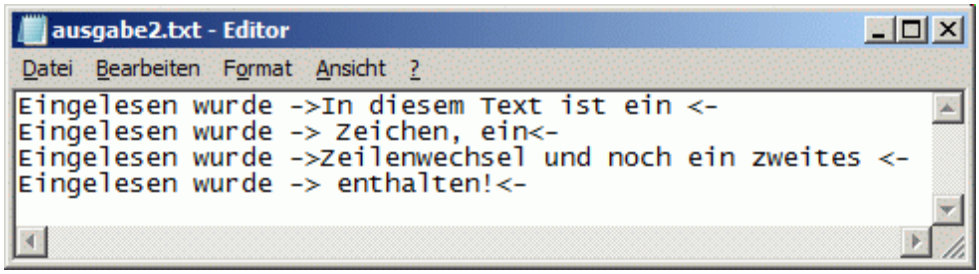
```
// Dateien öffnen
...
    DatasetIn Ein2;
    Ein2 = new DatasetIn("daten\\eingabe2.txt");
...
    DatasetOut Aus2;
    Aus2 = new DatasetOut("daten\\ausgabe2.txt");
...
// Schleife zur Verarbeitung von eingabe2.txt
    status = 0;
    do
    {
// 2   Einlesen einer Zeichenkette - '#' ist Ende der Zeichenkette
// 2.1 Einlesen der Zeichenkette mit Statusabfrage
        String St1;
```

```
        St1 = Ein2.dReadString('#');
        status = Ein2.dGetStatus();
// 2.2 Keine Fortsetzung der Verarbeitung bei Fehler oder Dateiende
        if (status == 0)
        {
// 2.3 Ausgabe der eingelesenen Zeichenkette
          Aus2.dPrint("Eingelesen wurde ->");
          Aus2.dPrint(St1);
          Aus2.dPrintln("<-");
        }
    }
    while (status == 0);
```

In der Datei eingabe2.txt stehen die folgenden Schriftzeichen:



Die beiden Zeichen '#' sowie der Zeilenwechsel in der Eingabedatei führen zur Bildung der folgenden Zeichenketten:



2.6.5.3 Einlesen einer Menüauswahl mit de.eikelberg.iotools

Häufig besteht ein Programm aus einer äußeren Schleife, in der die Verarbeitung des Programms wiederholt abläuft. Bietet das Programm verschiedene Verarbeitungsmöglichkeiten, so wird zu Beginn jedes Schleifendurchlaufs ein Menü ausgegeben, anhand dessen der Anwender die gewünschte Verarbeitungsmöglichkeit auswählen kann. Eine solche Menüsteuerung muss robust sein, damit das Programm nicht so leicht abstürzt. Das folgende Programm bsp02057.MenueAuswahl zeigt, wie eine solche robuste Menüsteuerung programmiert werden kann.

Die beiden wesentlichen Verarbeitungsschritte 'Menüauswahl' und 'Verarbeitung' sind - wie in Kapitel 2.7 noch genauer erklärt wird - im Programm bsp02057.MenueAuswahl in separate Methoden ausgelagert worden. Diese Aufteilung wird in Kapitel 2.7 genauer erklärt. Für uns zunächst interessant ist die Codierung der Methode zur Anzeige der Menüauswahl und zum Einlesen des Auswahlpunktes.

Die Methode zur Menüauswahl enthält den folgenden Java Programmcode:

```
// Menübildschirm aufbauen
    ScreenOut.kPrintln("Sie haben die Auswahl zwischen folgenden MenÃ¼punkten:");
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("  Diese Verarbeitung ............................ [a]");
    ScreenOut.kPrintln("  Jene Verarbeitung ............................ [b]");
    ScreenOut.kPrintln("  Eine andere Verarbeitung ...................... [c]");
    ScreenOut.kPrintln("  Noch eine andere Verarbeitung.................. [d]");
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("  Programm verlassen (exit) ..................... [x]");
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Geben Sie Ihre Auswahl ein :");
// Auswahl einlesen
    String s = KeyboardIn.kReadLine();  // (1)
    char auswahl;
    if (s.length() == 0)
      auswahl = ' ';                    // (2)
    else
    {
      s = s.trim();                     // (3)
      auswahl = s.charAt(0);            // (4)
    }
```

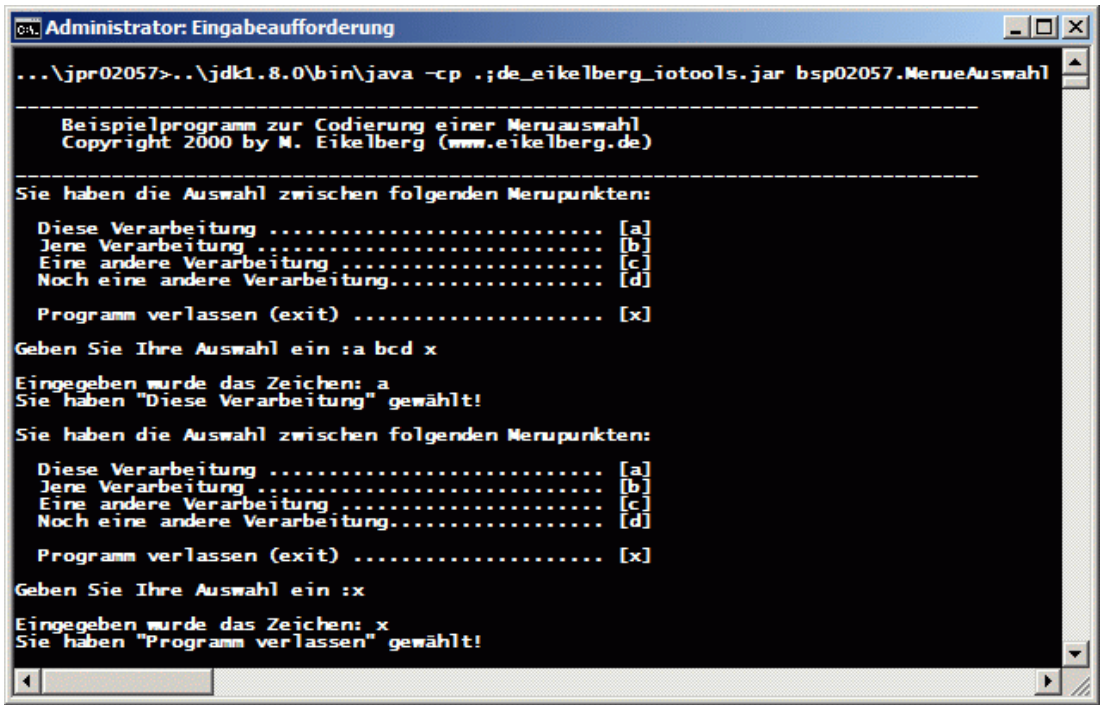
Der Einlesevorgang geschieht im Einzelnen wie folgt:

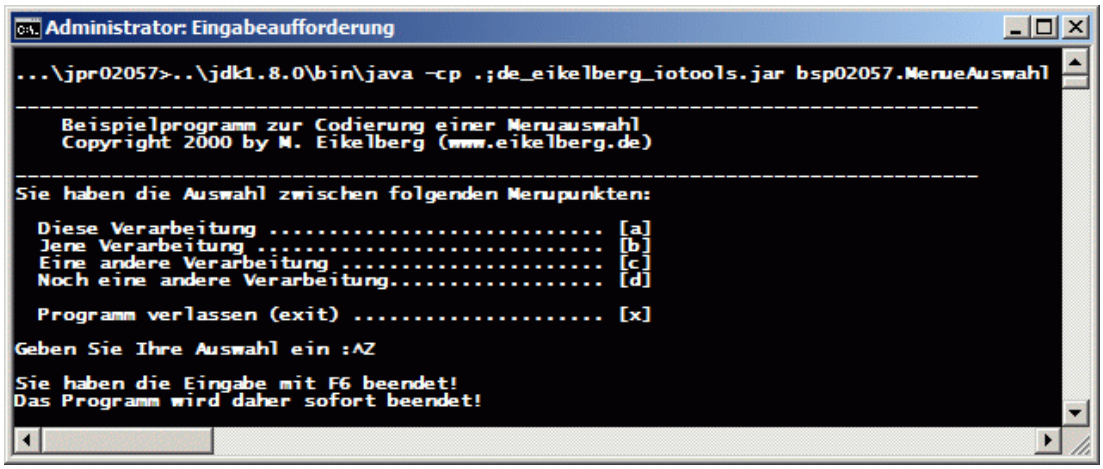
1. Nach Aufbau des Menübildschirmes wird im Schritt (1) zunächst die gesamte Tastatureingabe bis einschließlich Return eingelesen. Alle Zeichen *vor* Return werden in dem String-Objekt s gespeichert.
2. Falls genau nur die Taste F6 eingegeben wurde, enthält s eine leere Zeichenkette. In diesem Fall wird der Inhalt der steuernden char Variablen auswahl gleich einem Leerzeichen gesetzt. Dieses Leerzeichen wird im weiteren Programm als Kennzeichen für einen sofortigen Programmabbruch gewertet.
3. Wenn eine andere Eingabe als genau nur F6 erfolgt, werden im Schritt (3) alle Leerzeichen am linken und rechten Rand der Zeichenkette im String-Objekt s entfernt. Das so entstehende String-Objekt wird (per Referenzzuweisung) wieder der Variablen s zugewiesen. Damit ist anschließend das linke äußere Zeichen im String-Objekt s ein vom Leerzeichen verschiedenes Zeichen.
4. Im Schritt (4) wird jetzt das linke äußere char Zeichen aus dem String-Objekt s herausgelöst.

Die gesamte Verarbeitung wird so sehr robust: Es wird unabhängig von der Reaktion des Anwenders zunächst die Eingabe bis einschließlich Return eingelesen. Auch die irrtümliche bzw. ungeeignete Betätigung der Taste F6 ohne vorherige weitere Schriftzeichen wird noch nicht abgefangen. In der Annahme, das das erste vom Leerzeichen verschiedene Zeichen die Auswahl anzeigt, wird dieses gesucht, als Auswahl gewertet und der steuernden char Variablen auswahl als Wert zugewiesen. Im Rahmen der Verarbeitung kann dann der Wert von auswahl wie folgt verwendet werden:

```
    switch (auswahl)
    {
      case 'a':
        ScreenOut.kPrintln("Eingegeben wurde das Zeichen: "+auswahl);
        ScreenOut.kPrintln("Sie haben \"Diese Verarbeitung\" gewählt!");
        break;
....
      case 'x':
        ScreenOut.kPrintln("Eingegeben wurde das Zeichen: "+auswahl);
        ScreenOut.kPrintln("Sie haben \"Programm verlassen\" gewählt!");
        break;
      case ' ':
        ScreenOut.kPrintln("Sie haben die Eingabe mit F6 beendet!");
        ScreenOut.kPrintln("Das Programm wird daher sofort beendet!");
        break;
      default :
        ScreenOut.kPrintln("Eingegeben wurde das Zeichen: "+auswahl);
        ScreenOut.kPrintln("Sie haben keine gültige Auswahl eingegeben!");
    }
```

Dass eine derart aufgebaute Steuerung sehr robust ist, zeigen die folgenden beiden Bildschirmausdrucke:




Administrator: Eingabeaufforderung
...\jpr02057>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02057.MenueAuswahl
Beispielprogramm zur Codierung einer Menuauswahl
Copyright 2000 by M. Eikelberg (www.eikelberg.de)
Sie haben die Auswahl zwischen folgenden Menupunkten:
Diese Verarbeitung ..... [a]
Jene Verarbeitung ..... [b]
Eine andere Verarbeitung ..... [c]
Noch eine andere Verarbeitung..... [d]
Programm verlassen (exit) ..... [x]
Geben Sie Ihre Auswahl ein :^Z
Sie haben die Eingabe mit F6 beendet!
Das Programm wird daher sofort beendet!

2.6.6 Ein- und Ausgabe mit Standard Java Sprachmitteln

2.6.6.1 Generelles Vorgehen bei der Ein- und Ausgabe mit Standard Java Sprachmitteln

Mit den in Kapitel 2.6 bis hier erklärten Mitteln ist es jetzt möglich, dass Sie - sofern Sie dies wirklich wollen - sich von den Methoden des Paketes de.eikelberg.iotools lösen und auf Standard Java Methoden umsteigen. Man beachte dabei, dass die Java Standard Sprachmittel die Problematik der unterschiedlichen Codierungen unter Windows nicht berücksichtigen und daher deutsche Sonderzeichen usw. nicht immer korrekt verarbeitet werden (vgl. Kapitel 2.2.3.7)

Die Ein- und Ausgabe mit Standard Java Sprachmitteln basiert auf der Verwendung von String-Objekten. Wir betrachten zunächst die Ausgabe.

- Bei der **Ausgabe** werden dabei die verschiedenen Daten in ein String-Objekt verwandelt und es wird das String-Objekt ausgegeben.
- Bei Bedarf werden hierbei die Zeichenketten verschiedener String-Objekte mit dem +-Operator verkettet (vgl. Kapitel 2.6.3.4).

Das Einlesen von Daten mit Standard Java Sprachmitteln ist aufwändig. Das einfachste mit Standard Java Sprachmitteln zu codierende Verfahren hierzu läuft wie folgt ab:

1. Bei der **Eingabe** werden zunächst stets ganze Eingabezeilen eingelesen (vgl. hierzu auch 2.6.5.3).
2. Die eingelesenen Zeilen werden im nächsten Schritt in ihre Bestandteile zerlegt. Diese Bestandteile sind vom Datentyp String. Die Zerlegung geschieht mit Hilfe der Methoden der Klasse **StringTokenizer** (vgl. Kapitel 2.6.6.2).
3. Die einzelnen Bestandteile der eingelesenen Zeile sind vom Datentyp String und müssen mit den in Kapitel 2.6.3.8 erklärten Methoden in passende andere Datentypen umgewandelt werden.

In letzten Schritt können leicht Ausnahmen bzw. Laufzeitfehler auftreten. Dies geschieht insbesondere, wenn der jeweilige Bestandteil nicht in den gewünschten Datentyp umgewandelt werden kann. Bei der Codierung des beschriebenen Verfahrens zur Eingabe von Daten müssen also viele mögliche Laufzeitfehler berücksichtigt werden.

Die genannten Verfahren werden im Folgenden sowohl für die Arbeit mit Tastatur und Konsolbox (Kapitel 2.6.6.3) als auch für die Dateiverarbeitung (Kapitel 2.6.6.4) an Beispielen erklärt. Hierzu müssen jedoch zunächst die an dieser Stelle wichtigen Methoden der Klasse StringTokenizer erklärt werden.

2.6.6.2 Die Klasse StringTokenizer

Die Klasse **StringTokenizer** ist eine Klasse des Paketes **java.util** und dient dazu, ein String-Objekt in Teile zu zerlegen. Diese Teile heissen **Token**. Als Kriterium für die Einteilung eines String-Objektes in Token dient ein Trennzeichen, **Delimiter** genannt. Wird zum Beispiel ein Leerzeichen als Delimiter gewählt, so ist jede Zeichenkette, die kein Leerzeichen enthält und von Leerzeichen berandet wird, ein Token. Man betrachte hierzu folgende Codezeilen des Programms bsp02058.StringObjektZerlegung:

```
...
import java.util.*;

public class StringObjektZerlegung
{

  public static void main(String[] args)
  {
...
// String-Objekt anlegen
    String Str1 =
      "Wir lernen immer besser Java zu programmieren!";

// String zerlegen
    StringTokenizer StrTk1;
    StrTk1 = new StringTokenizer(Str1);
```

Das StringTokenizer-Objekt StrTk1 enthält 7 Token, nämlich die 7 Worte des im String-Objekt Str1 gespeicherten Satzes. Bei dieser Token-Bildung wirkten die Leerzeichen zwischen den einzelnen Worten als Delimiter und die Delimiter wurden nicht in die Token übernommen.

Standardmäßig gelten Leerzeichen, Tabulator-Zeichen, Return und Newline-Zeichen als Delimiter. Bei der Instantiierung kann dies jedoch beeinflusst werden, indem man als Argument des StringTokenizer-Konstuktors mit angibt, welche Zeichen als Delimiter gelten sollen:

Konstruktor	Wirkung
public **StringTokenizer**(String StrObj1)	Konstruktor instantiiert ein neues StringTokenizer-Objekt, indem die Zeichenkette des String-Objektes StrObj1 zerlegt wird. Dabei gelten die Zeichen Leerzeichen, Tabulator (\t), Return (\r) und Newline (\n) als Delimiter.

public **StringTokenizer**(String StrObj1, String Delim)	Konstruktor instantiiert ein neues StringTokenizer-Objekt, indem die Zeichenkette des String-Objektes StrObj1 zerlegt wird. Dabei gelten alle Zeichen in der Zeichenkette Delim als Delimiter.

Die wichtigsten Methode zur Verarbeitung von StringTokenizer-Objekten sind in der folgenden Tabelle zusammengestellt:

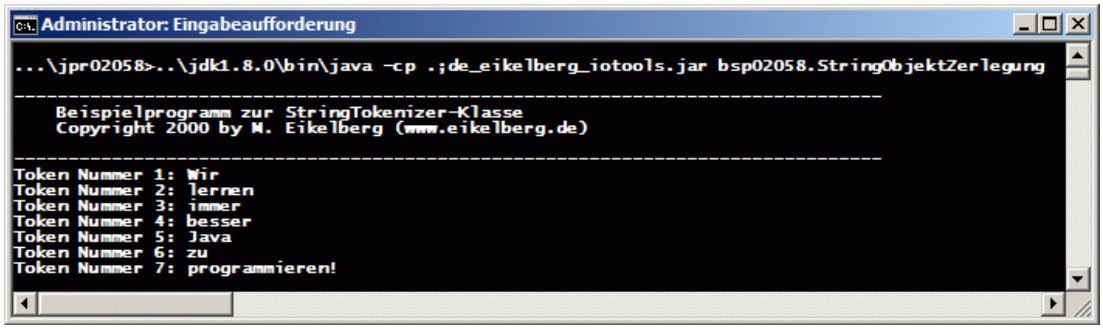
Methode	Wirkung
public int **countTokens**()	Methode gibt die Anzahl noch nicht mit nextToken() gelesener Token im StringTokenizer-Objekt zurück. Es wird also nicht stets die Gesamtzahl der Token ermittelt, sondern die Zahl der noch verbleibenden Token.
public String **nextToken**()	Methode gibt das nächst noch nicht verarbeitete Token des StringTokenizer-Objektes zurück. Dabei wird das Token in ein String-Objekt umgewandelt.
public boolean **hasMoreTokens**()	Methode testet, ob im StringTokenizer-Objekt außer den bereits mit nextToken() verarbeiteten Token noch ein weiteres Token vorhanden ist.

Die Token eines StringTokenizer-Objektes müssen also sukzessiv aus dem StringTokenizer-Objekt herausgeholt werden. Hierzu wird die Methode nextToken() verwendet. Die Methode nextToken kann stets so oft aufgerufen werden, wie countTokens() angibt. Die Anwendung der genannten Methoden der Klasse StringTokenizer zeigt der folgende Java Code des Programms bsp02058.StringObjektZerlegung:

```
// Ausgeben der einzelnen Token
    int uebrigeElemente = StrTk1.countTokens();                  // (1)
    int tokenZahl;
    while (StrTk1.hasMoreTokens())                               // (2)
    {
      tokenZahl = uebrigeElemente - StrTk1.countTokens() + 1;    // (3)
      ScreenOut.kPrint("Token Nummer "+tokenZahl+": ");          // (4)
      ScreenOut.kPrintln(StrTk1.nextToken());                    // (5)
    }
```

Da vor der obigen Verarbeitung noch kein Zugriff auf die Token des StringTokenizer-Objektes StrTk1 erfolgte, wird in Anweisung (1) die Gesamtzahl der Token in StrTk1 festgestellt. Der Ausdruck `StrTk1.hasMoreTokens()` ergibt einen boolean Wert und stellt fest, ob es noch weitere nicht verarbeitete Zeichen in StrTk1 gibt. Die while-Schleife bewirkt also das sukzessive Lesen aller Token in StrTk1. In Zeile (3) wird das beim Schleifendurchlauf zu lesende Zeichen nummeriert. Dieses Zeichen wird in Zeile (5) gelesen und ausgegeben. Ein Durchlauf des Programms bsp02058.StringObjektZerlegung führt also zur Ausgabe aller Token nach Zerlegung der im String-Objekt Str1 gespeicherten Zeichenkette

"Wir lernen immer besser Java zu programmieren!";



2.6.6.3 Einlesen von Tastatureingaben und Ausgabe in der Konsolbox mit Java Standardmethoden

In diesem Kapitel wird die Tastatureingabe und Ausgabe in der Konsolbox mit Java Standardmethoden an einem Beispiel demonstriert.

Die **Ausgabe** in der Konsolbox geschieht unter Verwendung der Methoden **print()** bzw. **println()** des Objektes **System.out**, das bereits in Kapitel 2.3.3.5 angesprochen wurde.

Die **Eingabe** über die Tastatur geschieht für Konsolprogramme unter Bezug auf das Objekt **System.in**. Da jedoch das Einlesen prinzipiell aufwendiger zu behandeln ist als die Ausgabe, sollten - anders als bei der Ausgabe - bei der Eingabe nicht direkt die Methoden von System.in verwendet werden, sondern die Klassen **InputStreamReader** sowie **BufferedReader** des Paketes **java.io** eingesetzt werden. Dies geschieht wie in folgendem Programmstück des Beispielprogramms bsp02059.KonsolEinAus:

```
import java.io.*;

public class KonsolEinAus
{
```

```
  public static void main(String[] args)
  {
...
// Eingabe vorbereiten
    InputStreamReader InStrReader1;
    InStrReader1 = new InputStreamReader(System.in);
    BufferedReader BufReader1;
    BufReader1 = new BufferedReader(InStrReader1);
```

Das Einlesen geschieht jetzt unter Verwendung der Methode **readLine()** des BufferedReader-Objektes BufReader1. Dieses Objekt bezieht seine Daten seinerseits über das InputStreamReader-Objekt InStrReader1 von dem Objekt System.in.

Die Ein- und Ausgabe geschieht jetzt wie in folgendem Programmteil:

```
// Eingabeaufforderung
    System.out.println(
      "\nGeben Sie in einer Zeile mindestens drei Worte und eine long Ganzzahl ein:");
...
// Einlesen
    String Ein1 = "";
    try
    {
      Ein1 = BufReader1.readLine();   // (6)
    } catch (IOException e)
    {
      e.printStackTrace();            // (7)
    }
```

Das Einlesen der eingegebenen Zeile erfolgt in Zeile (6). Diese Anweisung muss in eine **try ... catch** Anweisung eingefügt werden, da readLine() eine Ausnahme vom Typ java.io.IOException 'werfen' kann. Eine solche Ausnahme kann - wie in Kapitel 2.3.2.4 im Zusammenhang mit arithmetischen Ausnahmen erklärt - mit `try...catch` abgefangen werden. Allerdings sollte im Allgemeinen wesentlich mehr Aufwand getrieben werden als im Beispiel in Anweisung (7) codiert wurde.

Die eingelesene Zeile befindet sich nach erfolgreicher Abarbeitung von Anweisung (6) im String-Objekt Ein1 und sollte in der Regel wie in 2.6.6.2 erklärt unter Verwendung der StringTokenizer-Klasse in ihre Bestandteile zerlegt werden. In bsp02059.KonsolEinAus geschieht dies wie folgt:

```
// Zerlegung in die Token wie in bsp02058.StringObjektZerlegung
    StringTokenizer StrTk1;
    StrTk1 = new StringTokenizer(Ein1);
```

Die einzelnen Bestandteile der Eingabezeile sind jetzt als Token in dem StringTokenizer-Objekt gespeichert. Wendet man jetzt die Methode `nextToken()` an, so erhält man diese Token umgewandelt in den Datentyp String. Dies geschieht im Programm bsp02059.KonsolEinAus für die eingegebenen drei Worte wie folgt:

```
// Ausgeben der ersten drei Worte jeweils in einer Zeile
    String Aus1;
    System.out.println();
    for (int i=0;i<3;i++)
    {
      Aus1 = StrTk1.nextToken();
      System.out.println(Aus1);
    }
```

Das vierte Token enthält eine Zahl. Auch diese Zahl liegt nach Anwendung von `nextToken()` zunächst wieder als String-Objekt vor. Daher muss die Zeichenkette in diesem String-Objekt wie in Kapitel 2.5.3.8 beschrieben in eine Zahl umgewandelt werden:

```
// Umwandlung der Zahl nach float und Ausgabe der umgewandelten Zahl
// -> Zahl aus StringTokenizer-Objekt holen
    Aus1 = StrTk1.nextToken();
// -> Umwandlung in Datentyp String (vgl. Kapitel 2.6.3.8)
    float f1;
    f1 = Float.valueOf(Aus1).floatValue();
// -> Ausgabe der float Zahl
    System.out.println(f1);
```

Wie das Beispielprogramm bsp02059.KonsolEinAus zeigt, ist das Codieren der Ein- und Ausgabe von Daten mit Hilfe von Java Standardmethoden wesentlich aufwändiger als die Programmierung unter Verwendung der Methoden des Paketes de.eikelberg.iotools. Zudem erfordern die Java-Methoden die Kenntnis objektbasierter Programmierung sowie die Kenntnis der Ausnahmebehandlung mit `try ... catch`. Einen korrekten Ablauf von bsp02059.KonsolEinAus zeigt der folgende Bildschirmausdruck:

```
Administrator: Eingabeaufforderung
...\jpr02059>..\jdk1.8.0\bin\java bsp02059.KonsolEinAus

Beispielprogramm zu Java Standardmethoden bei Konsol-I/O
Copyright 2000 by M. Eikelberg (www.eikelberg.de)

Geben Sie in einer Zeile mindestens drei Worte und eine long Ganzzahl ein:
Die Zahl lautet 444 !

Die
Zahl
lautet
444.0

Programm beenden mit RETURN!
```

In obigem Programmlauf wurde als 5. Token eine Zeichenkette erzeugt, die aus einem Ausrufungszeichen besteht. Diese Zeichenkette wurde im Programm jedoch nicht weiter verarbeitet. Gibt man zwischen der Zahl und der Zeichenkette anders als oben *kein* Leerzeichen ein, so erhält man einen Laufzeitfehler, da das vierte Token das String-Objekt "444!" ergibt und somit nicht in eine Zahl umgewandelt werden kann:

```
Administrator: Eingabeaufforderung
...\jpr02059>..\jdk1.8.0\bin\java bsp02059.KonsolEinAus

Beispielprogramm zu Java Standardmethoden bei Konsol-I/O
Copyright 2000 by M. Eikelberg (www.eikelberg.de)

Geben Sie in einer Zeile mindestens drei Worte und eine long Ganzzahl ein:
Die Zahl lautet 444!

Die
Zahl
lautet
Exception in thread "main" java.lang.NumberFormatException: For input string: "444!"
        at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
        at sun.misc.FloatingDecimal.parseFloat(FloatingDecimal.java:122)
        at java.lang.Float.parseFloat(Float.java:451)
        at java.lang.Float.valueOf(Float.java:416)
        at bsp02059.KonsolEinAus.main(KonsolEinAus.java:55)
```

Die unterschiedliche Behandlung des 4. Tokens kann man auch an folgendem Bildschirmausdruck erkennen:

```
Administrator: Eingabeaufforderung
...\jpr02059>..\jdk1.8.0\bin\java bsp02059.KonsolEinAus

Beispielprogramm zu Java Standardmethoden bei Konsol-I/O
Copyright 2000 by M. Eikelberg (www.eikelberg.de)

Geben Sie in einer Zeile mindestens drei Worte und eine long Ganzzahl ein:
111 222 333 444

111
222
333
444.0

Programm beenden mit RETURN!
```

2.6.6.4 Einlesen aus eine Datei und Ausgabe in eine Datei mit Java Standardmethoden

Das Einlesen und Ausgeben von Daten bei Dateiverarbeitung geschieht nach ähnlichen Prinzipien wie die Verarbeitung bei Konsol-I/O. Grundsätzlich ist es hierbei möglich, jedes Zeichen auf der Festplatte einzeln einzulesen und zu verarbeiten. Das entsprechende Vorgehen zeigt das Programm bsp02060a.DateiEinAus, das Sie auf der DVD oder auf dem FTP-Server im Projektverzeichnis jpr02060 finden können.

Grundsätzlich vorteilhafter ist jedoch wieder ein zeilenweises Lesen bzw. Schreiben von Daten. Diese zeilenweise Verarbeitung der Datei kann auf zwei verschiedenen Wegen erreicht werden, wie auch der Programmcode von bsp02060.DateiEinAus weiter unten zeigt. Im Folgenden werden diese beiden Wege für eine Eingabedatei aufgelistet - für eine Ausgabedatei erfolgt die Verarbeitung analog:

<u>1. Weg</u>

1. **FileInputStream** anlegen, der auf die Datei zugreift.
2. **InputStreamReader** anlegen, der die Daten des FileInputStreams verarbeitet.
3. **BufferedReader** anlegen, der die Daten des InputStreamReaders puffert.

<u>2. Weg</u>

1. **FileReader** anlegen, der auf die Datei zugreift.
2. **BufferedReader** anlegen, der die Daten des FileReaders puffert.

Das Ende der Eingabedatei muss auf jeden Fall vom Programmierer selbst erkannt werden. Die in diesem Fall notwendigen Verarbeitungsschritte zeigt das Programm bsp02060.DateiEinAus:

```
package bsp02060;

import java.io.*;

/**
 * Title:        Beispielprogramm zur Dateiverarbeitung
 * Description:  Verwendung von Streams, Readern und Writern
 * Copyright:    Copyright (c) 2003
 * Company:
 * @author Prof. Dr. M. Eikelberg
 * @version 1.0
 */

public class DateiEinAus
{

  public static void main(String[] args)
  throws java.io.IOException
// Das Programm behandelt die Exceptions nicht, die beim Öffnen der
// Ein- und Ausgabedatei auftreten können
  {
// Startmeldung in Konsolbox ausgeben
    System.out.println(
      "\nBeispielprogramm zu Java Standardmethoden bei Datei-I/O");
    System.out.println(
      "Copyright 2003 by M. Eikelberg (www.eikelberg.de)");

// Auf den Eingabestrom von der Tastatur zugreifen
    InputStreamReader InStrReader1;
    InStrReader1 = new InputStreamReader(System.in);
    BufferedReader BufReader1;
    BufReader1 = new BufferedReader(InStrReader1);

// Eingabedatei öffnen - wirft IOException!
// Der folgende auskommentierte Block zeigt die Verarbeitung, wenn auf
// den Eingabestrom (Stream) Bezug genommen wird:
/*
    FileInputStream InDatei1;
    InDatei1 = new FileInputStream("daten\\eingabe.txt");
    InputStreamReader FileIn1;
    FileIn1 = new InputStreamReader(InDatei1);
*/
// Statt über den Eingabestrom kann man auch direkt über einen Reader auf die
// Datei zugreifen
    FileReader FileIn1;
    FileIn1 = new FileReader("daten\\eingabe.txt");
    BufferedReader BufReaderFileIn1;
    BufReaderFileIn1 = new BufferedReader(FileIn1);

// Ausgabedatei öffnen - wirft IOException!
// Der folgende auskommentierte Block zeigt die Verarbeitung, wenn auf
// den Ausgabstrom (Stream) Bezug genommen wird:
/*
    FileOutputStream OutDatei1;
    OutDatei1 = new FileOutputStream("daten\\ausgabe.txt");
    OutputStreamWriter FileOut1;
    FileOut1 = new OutputStreamWriter(OutDatei1);
*/
// Statt über den Ausgabestrom kann man auch direkt über einen Writer auf die
// Datei zugreifen
    FileWriter FileOut1;
    FileOut1 = new FileWriter("daten\\ausgabe.txt");
    BufferedWriter BufWriterFileOut1;
    BufWriterFileOut1 = new BufferedWriter(FileOut1);

// Einlesen bis Dateiende
    String zeile = " ";
// Das Dateiende, also das Ende des Streams ergibt beim Einlesen null
    while (zeile != null)
    {
```

```
// Lesen einer Zeile aus der Eingabedatei
      try
      {
        zeile = BufReaderFileIn1.readLine();
      } catch (IOException e)
      {
        e.printStackTrace();
        zeile = null;         // Abbruch der Verarbeitung
      }

// Ausgabe der Zeile auf Bildschirm und in Datei, sofern nicht Ende der
// Eingabedatei erreicht wurde
      if (zeile != null)
      {
// Ausgabe auf dem Bildschirm
        System.out.print("\nEingelesene Zeichenkette: " + zeile);
// Ausgabe in die Ausgabedatei
        BufWriterFileOut1.write(zeile);
// Zeilenwechsel in die Ausgabedatei schreiben
        BufWriterFileOut1.write("\r\n");
      }
    }

// Meldung bei Dateiende
    System.out.println("\n\nDateiende erreicht !");

// Inhalt des Puffers zur Ausgabedatei in die Datei schreiben und
// Dateien schließen
    BufWriterFileOut1.flush();
    FileIn1.close();
    FileOut1.close();

// Bildschirmausgabe anhalten
    System.out.println("\nProgramm beenden mit RETURN!");
    try
    {
      BufReader1.readLine();
    } catch (IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

2.6.6.5 Exkurs: Verwendung von nicht standard Klassen zur Vereinfachung des Einlesevorgangs aus einer Datei und der Ausgabe in eine Datei

Die oben gezeigte Verarbeitung mit Java Standardmethoden ist stellenweise unkomfortabel. Daher ist eine Vereinfachung der Verarbeitung von Ein- und Ausgabedateien durch Verwendung von Klassen, die - wie die Klassen in de.eikelberg.iotools - nicht zum Standard Java Sprachumfang gehören, nicht ungewöhnlich. Das folgende Programm zeigt - als Alternative zu de.eikelberg.iotools - als Beispiel die Klasse **borland.jbcl.io.EncodedInputStream**. Diese Klasse ist Bestandteil der **Borland Java Beans Component Library**, also der **Borland Java Beans Komponentenbibliothek** bzw. kurz **JBCL-Paket**. Das JBCL-Paket ist in einigen (inzwischen veralteten) Versionen der Entwicklungsumgebung Borland JBuilder Standard enthalten und stellt Java Klassen für die Dateiverarbeitung, für die Datenbankanbindung sowie für graphische Benutzeroberflächen bereit. *Eclipse enthält nicht das JBCL-Paket!*

Bei Verwendung der Klasse **borland.jbcl.io.EncodedInputStream** wird die Codierung des Lesevorgangs aus einer Datei wie bei der Verwendung von Java Standardmethoden mit einer Methode **readLine()** realisiert, die analog zur Methode readLine() der Klasse BufferedReader wirkt und eine ganze Zeile aus einer Datei liest. Diese Methode wird im Beispielprogramm bsp02061.JBCLLeseDatei verwendet. Die Verarbeitung in bsp02061.JBCLLeseDatei erfolgt analog zu der Verarbeitung in den Progrmmen bsp02059.KonsolEinAus und bsp02060.DateiEinAus. Der Lesevorgang ist in bsp02061.JBCLLeseDatei folgt codiert:

```
...
import java.io.*;
...
import borland.jbcl.io.EncodedInputStream;

public class JBCLLeseDatei
{

  public static void main(String[] args)
  throws java.io.IOException
```

```
  {
...
// Eingabedatei öffnen - wirft IOException!
    FileInputStream InDatei1;
    InDatei1 = new FileInputStream("daten\\eingabe.txt");
    EncodedInputStream InStream1;
    InStream1 = new EncodedInputStream(InDatei1);

// Einlesen einer Zeile aus der Datei
    String Ein1 = "";
    try
    {
      Ein1 = InStream1.readLine();
    } catch (IOException e)
    {
      e.printStackTrace();
    }
...
// Eingabedatei schließen
    InStream1.close();
...
```

Man sieht, dass der Einlesevorgang bei Verwendung des JBCL-Paketes wie bei einer Verwendung der Standard-Methoden erfolgt. Auch das Paket de.eikelberg.iotools stellt eine Methode readLine() zur Verfügung. Ersetzt man die JBCL-Klasse EncodedInputStream durch die Klasse DatasetIn, so erhält man das Programm bsp02062.EikLeseDatei, das folgenden Java Quellcode enthält:

```
...
import de.eikelberg.iotools.*;

public class EikLeseDatei
{

  public static void main(String[] args)
  {
...
// Eingabedatei öffnen - wirft keine IOException!
    DatasetIn InDatei1;
    InDatei1 = new DatasetIn("daten\\eingabe.txt");

// Einlesen einer Zeile aus der Datei
    String Ein1 = InDatei1.dReadLine();

...
// Eingabedatei schließen
    InDatei1.dClose();
...
```

2.7 Methoden

2.7.1 Deklaration von Methoden

2.7.1.1 Sinn und Zweck von Methoden

Wie im Kapitel zu Struktogrammen (Kap. 1.5.2) bereits angesprochen ist es ein Ziel effektiver Programmentwicklung, **wiederverwendbare Blöcke** zu bilden. Diese Blöcke werden ein einziges Mal codiert und getestet, können jedoch anschließend mehrfach in einem Programm oder aber auch in mehreren Programmen verwendet werden. Allerdings sind bis jetzt in diesem Kurs die entsprechenden Sprachmittel hierzu noch nicht erklärt worden.

Eine Wiederverwendbarkeit von Blöcken kann man in einfacher Weise durch Codierung von **Methoden** erreichen. Früher bezeichnete man diese Methoden auch als Prozeduren. Soll die codierte Methode lediglich *innerhalb eines Programms* verwendet werden, so kann die Methode oft in der gleichen Klasse wie die Methode main() angeordnet werden. Möchte man Methoden in mehreren Programmen verwenden, so sollte man diese Methoden geeignet in **Klassen** anordnen. In diesem Kapitel wird zunächst die Bildung von Methoden erläutert, in Kapitel 2.8 anschließend die Anordnung von Methoden in Klassen.

Eine **Methode** ist eine Folge von Anweisungen, die unter einem gemeinsamen Namen aufgerufen werden können. Dabei können für die Ausführung der Anweisungen benötigte Werte an die Methode übergeben werden. Ferner kann eine Methode (höchstens) ein Ergebnis-Wert liefern. Methoden sind also eine praktische Zusammenfassung mehrerer Anweisungen, die bei der Programmentwicklung zuvor in einem **Modulblock** zusammengefasst worden sind.

Der einzige scheinbare Nachteil von Methoden ist, dass sie maximal einen Wert als Ergebnis liefern können. Dieser Punkt ist jedoch de facto kein Nachteil, da eine Bildung von Klassen mit Klassenvariablen gemäß Kapitel 2.8 für Methoden, die mehrere Werte verändern, die geeignetere Vorgehensweise ist.

Methoden werden **deklariert** und **aufgerufen**. In der **Methodendeklaration** (kurz **Deklaration**) fasst man die Ausführungen zusammen, die in der Methode als Block stets gemeinsam auszuführender Anweisungen gruppiert werden sollen. Die Deklaration einer Methode bewirkt zunächst keinerlei Ausführungsschritte beim Ablauf eines Programms. Möchte man die in der Deklaration zusammengefassten Anweisungen an einer Stelle des Programms ausführen, so codiert man an dieser Stelle einen **Methodenaufruf** (auch kurz **Aufruf**).

2.7.1.2 Methodendeklaration

Die Deklaration einer Methode hat im Allgemeinen folgende Form:

```
Modifizierer  DatentypRückgabewert  MethodenName(Typ1 formalerParm1,
                                                 Typ2 formalerParm2,
                                                 ...,
                                                 TypN formalerParmN)
{
  MethodenRumpf
}
```

Die Bestandteile der Methodendeklaration haben folgende Bedeutung:

- Den Bereich `Modifizierer ... TynN formalerParmN` nennt man auch **Methodenkopf**.
- Der Bereich `MethodenRumpf` heisst **Methodenrumpf** und enthält die in der Methode zusammengefassten Anweisungen der Methode.
- Als `MethodenName` kann ein beliebiger nach den Regeln für **Bezeichner** gewählter **Methodenname** eingesetzt werden. Es ist unter Java-Programmierern üblich, Methodennamen stets mit einem *Kleinbuchstaben* beginnen zu lassen.
- Als **Modifizierer** können die bereits in Kapitel 2.3.3 erklärten Schlüsselworte eingesetzt werden. Sie regeln die Sichtbarkeit sowie die Frage, ob eine Methode statisch oder nicht statisch sein soll (s.u. und Kapitel 2.8).
- Eine Methode kann maximal einen Wert beim Aufruf zurückgeben. Der `DatentypRückgabewert` legt diesen Datentyp fest (s.u.).
- Über die **formalen Parameter** `formalerParm1` bis `formalerParmN` geschieht die Übergabe von Werten an die Methode (vgl Kapitel 2.8).

Im Folgenden muss also die Bedeutung der Modifizierer und des Eintrags zu `DatentypRückgabewert` erklärt werden.

2.7.1.3 Modifizierer

Als **Modifizierer** können die bereits in Kapitel 2.3.3 erklärten Schlüsselworte private, protected, public und static eingesetzt werden. Dies wird in Kapitel 2.8 genauer besprochen. In diesem Kapitel wird zunächst *nur die Methodendeklaration für den Fall erläutert, dass die Methodendeklaration in der Java-Quelldatei codiert wird, die die Methode main(...) enthält*. In diesem Fall gilt:

1. Codiert man eine Methodendeklaration in einer Klasse, die auch die Methode `main(...)` der jeweiligen Anwendung enthält, so muss die Methode als **static** gekennzeichnet sein.

2. Die Modifizierer zur Sichtbarkeit können beliebig gewählt werden, sinnvoll ist im vorliegenden Fall jedoch eigentlich nur der Modifizierer **private**.

Als Beispiel betrachten wir die Methode `menueAuswahl()` aus dem Programm bsp02057.MenueAuswahl. Der Kopf dieser Methode ist wie folgt codiert:

```
  private static char menueAuswahl()
```

Man sieht die beiden Modifizierer `private` und `static`. Der Eintrag `char` legt in diesem Beispiel fest, dass die Methode menueAuswahl() einen Wert vom Datentyp char ergibt. menueAuswahl() ergibt nach Ausführung aller Anweisungen also einen Wert vom Datentyp char. Diesen Wert nennt man Rückgabewert der Methode.

2.7.1.4 Rückgabewert von Methoden

Der **Rückgabewert** einer Methode ist der Wert des Ausdrucks, der die Methode menueAuswahl (s.u.) aufruft.

- Der Rückgabewert ist also der Datentyp des Ergebniswertes der Anweisungen der Methode.
- Als **Rückgabewert** einer Methode kann prinzipiell jeder Datentyp von Java auftreten. Auch ein Klassenname kann an dieser Stelle stehen.
- Es kann aber auch sein, dass eine Methode keine Rückgabewert ergibt. In diesem Fall codiert man das Schlüsselwort **void** (engl. Wort für "leer") als Rückgabewert.

Inerhalb der Methode legt man den Rückgabewert wie folgt fest:

```
    return VarName;
```

`VarName` ist dabei eine Variable, deren Datentyp dem Datentyp *`DatentypRückgabe`* in der Methodendeklaration entsprechen muss. In der Methode menueAuswahl() sieht dies wie folgt aus:

```
  private static char menueAuswahl()
  {
    char auswahl;
...
    return auswahl;
  }
```

Natürlich macht eine solche Verarbeitung nur dann Sinn, wenn der Inhalt der Variablen in der Methode verändert wird. Insbesondere muss die Variable `auswahl` im vorliegenden Fall in der Methode überhaupt erst einen Wert zugewiesen bekommen. Dies geschieht auch in der Methode menueAuswahl() in bsp02057.MenueAuswahl:

```
  private static char menueAuswahl()
  {
// Menübildschirm aufbauen
    ScreenOut.kPrintln("Sie haben die Auswahl zwischen folgenden Menüpunkten:");
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("  Diese Verarbeitung ............................ [a]");
    ScreenOut.kPrintln("  Jene Verarbeitung ............................. [b]");
    ScreenOut.kPrintln("  Eine andere Verarbeitung ...................... [c]");
    ScreenOut.kPrintln("  Noch eine andere Verarbeitung.................. [d]");
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("  Programm verlassen (exit) ..................... [x]");
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Geben Sie Ihre Auswahl ein :");
// Auswahl einlesen
    String s = KeyboardIn.kReadLine();  // Zeichenkette einlesen
    char auswahl;
    if (s.length() == 0)                // falls keine Eingabe erfolgte, wird ein
      auswahl = ' ';                    // Leerzeichen zurückgegeben
    else
    {
      s = s.trim();                     // Abschneiden von Leerzeichen an beiden Enden von s
      auswahl = s.charAt(0);            // linkes äußeres Zeichen aus s herauskopieren
    }
    ScreenOut.kPrintln();
// gelesene Auswahl an rufende Methode zurückgeben
    return auswahl;
  }
```

2.7.1.5 Formale Parameter

Noch nicht erläutert und auch nicht im Beispiel vorhanden sind die Einträge `Typ1 formalerParm1,...,TypN formalerParmN` zwischen den runden Klammern in der Methodendeklaration. Als Einträge `formalerParm1` bis `formalerParmN` trägt man Namen ein, die ähnlich wie Variablennamen wirken, jedoch eine etwas andere Bedeutung haben und die **formale Parameter** genannt werden. Die Liste aller Parameter einer Methode heißt **Parameterliste**. Die Einträge `Typ1` bis `TypN` legen wie bei einer Variablendeklaration die Datentypen der formalen Parameter fest. Wie im Beispielprogramm bsp02057.MenueAuswahl können die formalen Parameter auch ganz fehlen.

Die formalen Parameter werden in Kapitel 2.7.2 erklärt.

2.7.1.6 Der Methodenaufruf

Wie bereits erwähnt, führt die Methodendeklaration zu keiner Ausführung der Anweisungen der Methode. Vielmehr codiert man an der Stelle des Programms, an der die Anweisungen der Methode ausgeführt werden sollen, einen **Methodenaufruf**. Die Methode, in der der Methodenaufruf codiert ist, wird in diesem Zusammenhang dann als **rufende Methode** bezeichnet.

Die Syntax des Methodenaufrufs lautet im Allgemeinen wie folgt:

```
MethodenName(aktuellerParm1,aktuellerParm2,...,aktuellerParmN)
```

Die Einträge `aktuellerParm1,aktuellerParm2,...,aktuellerParmN` heißen **aktuelle Parameter** und müssen von dem Datentyp sein, der in der Deklaration der Methode zu den entsprechenden positionsgleichen formalen Parametern `formalerParm1,formalerParm2,..., formalerParmN` angegeben ist (genaueres hierzu in Kapitel 2.7.2). Werden in der Deklararation einer Methode wie bei der Deklaration der Methode menueAuswahl() *keine* formalen Parameter angegeben, so dürfen beim Aufruf der Methode auch keine aktuellen Parameter eingetragen werden. Zwischen den runden Klammern steht beim Aufruf der Methode in diesm Fall also *kein Eintrag*.

Ist eine Methode als void deklariert, so ergibt sie keinen Rückgabewert und der oben beschriebene Methodenaufruf ist zugleich eine geeignete Anweisung des Programms. Hat die Methode dagegen einen **Rückgabewert** vom Datentyp `DatentypRückgabewert`, so kann der Rückgabewert in der rufenden Methode mit dem Anweisungsoperator einer Variablen mit Datentyp `DatentypRückgabewert` zugewiesen werden:

```
DatentypRückgabewert VarName;
...

VarName = MethodenName(aktuellerParm1,
                       aktuellerParm2,
                       ...,
                       aktuellerParmN)
```

Die Beispielmethode menueAuswahl() im Programm bsp02057.MenueAuswahl ist ohne Angabe einer Parameterliste deklariert worden, so dass beim Aufruf auch keine Parameterliste eingetragen werden darf. Der Rückgabewert der Methode ist vom Datentyp char. Somit lautet ein Aufruf der Methode menueAuswahl() wie folgt:

```
// Variable für Rückgabewert deklarieren
  char auswahl;
...
// Methode zur Menüauswahl aufrufen
  auswahl = menueAuswahl();
```

2.7.1.7 Beispiel

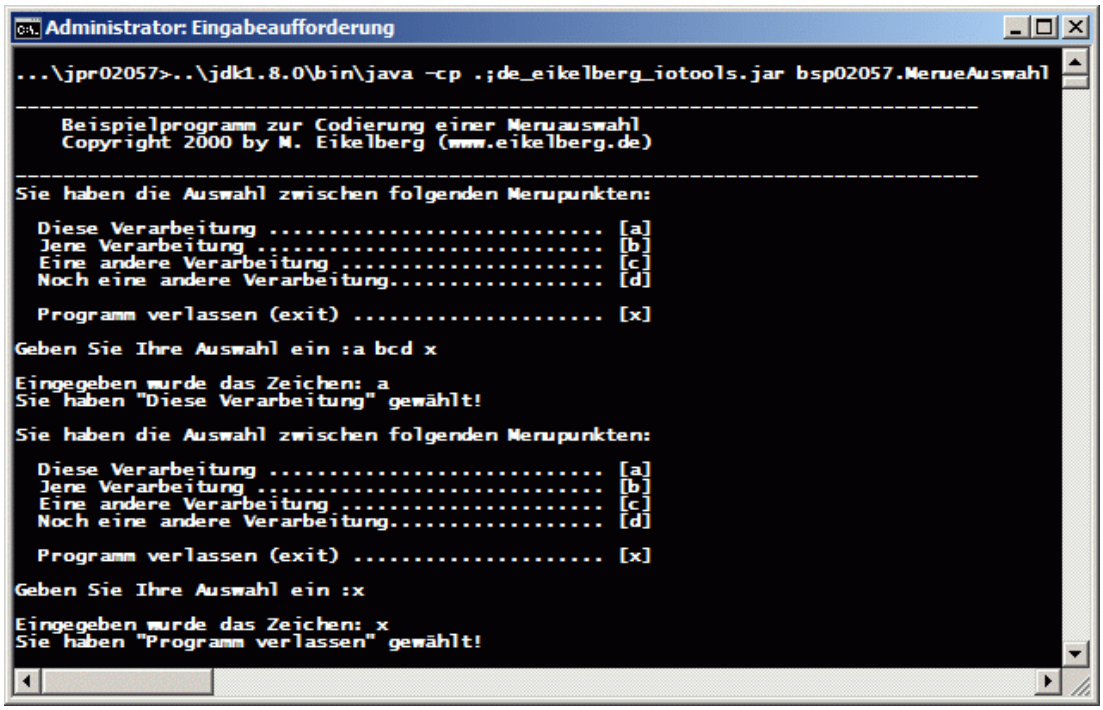
Im Programm bsp02057.MenueAuswahl sind wie gezeigt alle Schritte zur Arbeit mit einer Methode ohne formale Parameter enthalten. Sie stehen insgesamt in folgendem Zusammenhang:

```
package bsp02057;

import de.eikelberg.iotools.*;

public class MenueAuswahl
{

  public static void main(String[] args)
  {
...
// 1. Variable für Rückgabewert der Methode menueAuswahl() deklarieren
    char auswahl;

// Schleife für mehrfache Verarbeitung
    do
    {
```

```
// 2. Methode zur Menüauswahl aufrufen
      auswahl = menueAuswahl();
...
// Ende der Schleife
    } while (auswahl != 'x' && auswahl != ' ');
...
  }  // Ende von main()

// 3. Deklaration der Methode menueAuswahl()
  private static char menueAuswahl()
  {
// Menübildschirm aufbauen
...

// Auswahl einlesen
    String s = KeyboardIn.kReadLine();
// 4. Variable für Rückgabewert deklarieren
    char auswahl;
    if (s.length() == 0)
      auswahl = ' ';
    else
    {
      s = s.trim();
      auswahl = s.charAt(0);
    }
    ScreenOut.kPrintln();
// 5. Rückgabewert der Methode festlegen
    return auswahl;
  }  // Ende von menueAuswahl()
...
}
```

Die Variable **auswahl** wird im Programm bsp02057.MenueAuswahl an den mit 1. und 4. gekennzeichneten Stellen deklariert. Diese beiden Variablen sind jeweils nur in der Methode main() und in der Methode menueAuswahl() gültig, somit unabhängig und könnten auch verschiedene Namen tragen. Der Übertrag des Wertes der Variablen `auswahl` der Methode menueAuswahl() in die Variable `auswahl` der Methode main() geschieht mit der return-Anweisung (5.) und dem Anweisungsoperator beim Aufruf der Methode (2). Der bei dieser Übergabe eines Wertes zulässige Datentyp wird in der Methodendeklaration (3) als Datentyp char festgelegt.

Der folgende Bildschirmausdruck zeigt die Wirkung der Methode menueAuswahl():



2.7.2 Formale und aktuelle Parameter

2.7.2.1 formale Parameter

Bei der Codierung und beim Aufruf von Methoden ist es sehr häufig erforderlich, dass der Methode für die Ausführung der Anweisungen bestimmte Werte übergeben werden. Dies geschieht mit Hilfe der formalen und aktuellen Parameter.

Formale Parameter sind Variablen, die in der Deklaration der Methode eingetragen werden. Die Deklaration formaler Parameter erfolgt also nicht wie bei Variablen sonst üblich im Methodenrumpf oder direkt in der Klasse, sondern bei der Deklaration der Methode im Methodenkopf. Dabei wird zu jeder Variablen der Datentyp mit angegeben. Als Folge der Deklaration der formalen Parameter werden beim Aufruf der Methode Speicherplätze angelegt, in die die von der Methode zu verwendenden Werte *kopiert* werden (vgl. 2.7.2.3).

Eine Methodendeklaration mit formalen Parametern lautet wie bereit in Kapitel 2.7.1 angesprochen wie folgt:

```
Modifizierer  DatentypRückgabewert  MethodenName(Typ1 formalerParm1,
                                                 Typ2 formalerParm2,
                                                 ...,
                                                 TypN formalerParmN)
{
  MethodenRumpf
}
```

Dabei werden für die formalen Parameter `formalerParm1` bis `formalerParmN` nach den Regeln von Java zu bildende Variablennamen eingesetzt. Vor den Variablennamen werden wie bei einer Variablendeklaration die Datentypen `Typ1` bis `TypN` der formalen Parameter eingetragen.

Als formale Parameter können sowohl einfache Datentypen als auch Referenzdatentypen eingetragen werden. Beispiele für die Deklaration von Methoden mit formalen Parametern findet man in den Programmen bsp02064.MethodenMitParametern und bsp02066.MethodenMitParametern. Beide Programme führen die gleiche Verarbeitung durch, nämlich die Berechnung des arithmetischen Mittelwertes für drei Gleitpunktzahlen. In bsp02064.MethodenMitParametern werden hierzu einfache Datentypen verwendet, in bsp02066.MethodenMitParametern Referenzdatentypen.

Die Berechnung des Mittelwertes geschieht in bsp02064.MethodenMitParametern in der folgenden Methode:

```
  private static double arithmetMittel(double s1,
                                       double s2,
                                       double s3)
  {
    double s;
    s = 1.0/3.0*(s1 + s2 + s3);
    return s;
  }
```

Im Programm erfolgt die Berechnung des Mittelwertes unter Verwendung eines Arrays `sd` aus double Komponenten. Die Deklaration dieser Methode lautet wie folgt:

```
  private static double arithmetMittel(double[] sd)
  {
    double s = 0.0;
    for (int i=0; i<3; i++)
      s = s + sd[i];
    s = 1.0/3.0*s;
    return s;
  }
```

Wir sehen, dass die beiden Methoden den gleichen Namen haben. Dies ist bei der Java-Programmierung auch dann möglich, wenn die Methoden in einer gemeinsamen Klasse deklariert sind. Der Unterschied zwischen den Methoden liegt in der Parameterliste. Das Codieren gleichnamiger Methoden mit unterschiedlichen Parameterlisten nennt man **überladen der Methoden**, die jeweiligen Methoden nennt man auch **überladene Methoden**.

2.7.2.2 aktuelle Parameter

Die formalen Parameter der Methodendeklaration sind nicht mit einem Wert gefüllt, sondern müssen bei dem Aufruf der Methode mit Werten gefüllt werden. Diese Werte werden bei dem Aufruf der Methode angegeben. Der Methodenaufruf lautet in der allgemeinen Form wie folgt:

```
DatentypRückgabewert VarName;
...
```

Die Einträge *aktuellerParm1* bis *aktuellerParmN* müssen Werte bereitstellen, die den Datentypen der formalen Parameter *formalerParm1* bis *formalerParmN* der Methodendeklaration entsprechen. Als aktuelle Parameter können Variablennamen der rufenden Methode oder Konstanten eingetragen werden. Die Zuweisung der Werte der aktuellen Parameter zu den formalen Parametern erfolgt von links nach rechts.

Da Methoden überladen werden können und somit mehrere gleichnamige Methoden mit unterschiedlichen Parameterliste existieren können, ist eine Methode durch ihren Namen und ihre Parameterliste eindeutig gekennzeichnet. Daher müssen die aktuellen Parameter des Methodenaufrufs sowohl in Bezug auf die Anzahl als auch in Bezug auf den Datentyp genau zu den formalen Parametern der Methodendeklaration passen. Ferner ist es streng genommen notwendig, Methoden nicht nur mit ihrem Namen sondern auch mit ihren Parametern zu nennen.

Die Methode arithmetMittel(..) des Programms bsp02064.MethodenMitParametern sollte also genau mit **arithmetMittel(double s1,double s2,double s3)** oder kurz mit arithmetMittel(double,double,double) bezeichnet werden. Die Methode arithmetMittel(..) des Programms bsp02066.MethodenMitParametern wird also besser durch **arithmetMittel(double[] sd)** bzw. arithmetMittel(double[]) bezeichnet.

Unter Verwendung passender aktueller Parameter werden die Methoden arithmetMittel(double,double,double) und arithmetMittel(double[] sd) wie folgt in den Programmen bsp02064.MethodenMitParametern und bsp02066.MethodenMitParametern aufgerufen:

```
package bsp02064;
...
public class MethodenMitParametern
{

  public static void main(String[] args)
  {
...
// Summanden einlesen
    double summand1;
    double summand2;
    double summand3;
...
// Mittelwert berechnen;
    double arMittel = arithmetMittel(summand1,summand2,summand3);
...
  }
...
  private static double arithmetMittel(double s1,
                                       double s2,
                                       double s3)
  {
    double s;
    s = 1.0/3.0*(s1 + s2 + s3);
    return s;
  }
...
}
```

Die entsprechende Verarbeitung im Programm bsp02066.MethodenMitParametern lautet wie folgt:

```
package bsp02066;
...
public class MethodenMitParametern
{

  public static void main(String[] args)
  {
...
// Array anlegen für die Summanden
    double[] summand;
    summand = new double[3];
...
// Mittelwert berechnen;
    double arMittel = arithmetMittel(summand);
...
  }
...
  private static double arithmetMittel(double[] sd)
  {
    double s = 0.0;
    for (int i=0; i<3; i++)
      s = s + sd[i];
```

```
    s = 1.0/3.0*s;
    return s;
  }
...
}
```

Äußerlich ist die Wirkung der beiden Methoden arithmetMittel(...) gleich, jedoch gibt es in Bezug auf die Verarbeitung einen wichtigen Unterschied, der im folgenden Abschnitt besprochen werden soll:

2.7.2.3 call by value

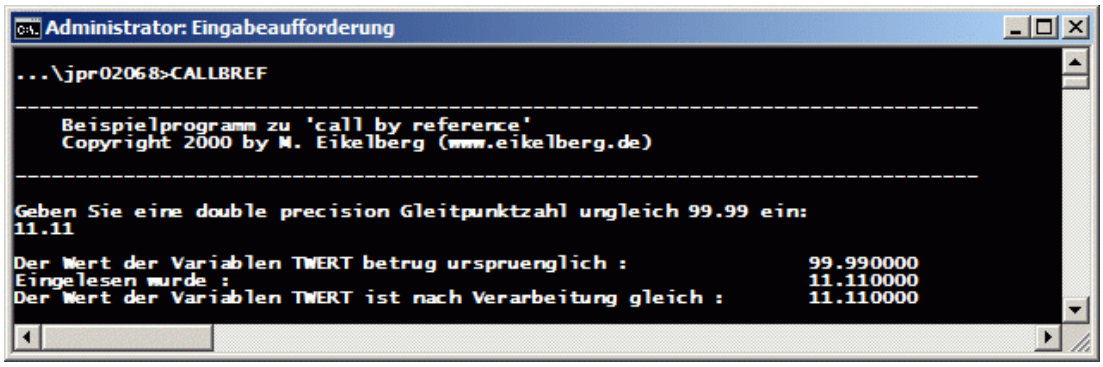
In 2.7.2.1 bereits kurz angesprochen werden beim Aufruf einer Methode die Werte der aktuellen Parameter in die Speicherplätze zu den formalen Parametern **kopiert**. Man spricht daher auch von **"call by value"** und meint damit, dass beim Aufruf (call) der Methode die Werte (value) übergeben werden.

In einigen Programmiersprachen ist stellenweise ein anderes Verhalten als call by value implementiert: In diesen Fällen wird für die aktuellen Parameter und die formalen Parameter der selbe Speicherplatz verwendet. Man spricht dann von **"call by reference"** und meint damit, das beim Aufruf (call) der Methode der Verweis (reference) auf den Speicherplatz der Variablen der rufenden Methode übergeben wird. Dies hat zur Folge, dass nach Beendigung der Methode als aktuelle Parameter eingetragene Variablen veränderte Werte haben können. Die Programmiersprache FORTRAN verwendet im Zusammenhang mit Methoden - die FORTRAN Sprachelemente hierzu sind FUNCTION-Unterprogramme bzw. SUBROUTINE-Unterprogramme - die Übergabe von Parametern mit call by reference.

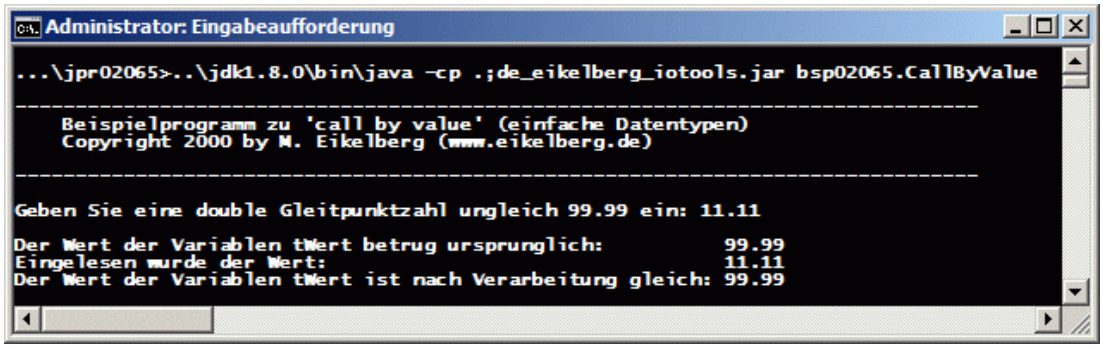
Das folgende Beispiel illustriert call by value und call by reference. Hierzu werden ein Java- und ein FORTRAN-Programm in ihrer Wirkung verglichen. Das Fortran-Programm

- entspricht der FORTRAN 77 Sprachspezifikation,
- wurde mit Microsoft Fortran PowerStation 4.0 entwickelt,
- befindet sich im Verzeichnis bsp02068\CallBRef,
- heisst CALLBREF.FOR

und ergibt folgende Bildschirmausgabe:



Das entsprechende Java-Programm bsp02065.CallByValue ist in Bezug auf die Syntax gleich aufgebaut wie das Programm CALLBREF. Dennoch erhält man ein anderes Ergebnis:



Der Unterschied zwischen den beiden Programmen liegt also darin, dass die Variable TWERT verändert wurde, die Variable tWert jedoch nicht. Wir betrachten zunächst den in diesem Zusammenhang interessanten Teil des FORTRAN Programmcode:

```
      PROGRAM CALLBREF
C     Beispielprogramm zu call by reference

C     Vereinbarung der Variablen des Hauptprogramms
      DOUBLE PRECISION TWERT,ZWERT,LIESZAHL
```

```
...

C     (1) Variable TWERT mit Wert initialisieren
      TWERT = .9999D2

C     ZWERT einlesen und dabei Test-Variable TWERT an FUNCTION übergeben
C     (2) TWERT ist aktueller Parameter, wird verändert wegen call by reference
      ZWERT = LIESZAHL(TWERT)

C     Werte ausgeben
      WRITE(*,110) 99.99
...
      WRITE(*,111) ZWERT
...
      WRITE(*,112) TWERT
...
      END

      DOUBLE PRECISION FUNCTION LIESZAHL(T)
C     (3) T ist formaler Parameter
C     Z ist lokale Variable
      DOUBLE PRECISION T,Z
...
      READ (*,202) Z
202     FORMAT(F16.0)
C     (4) Dem formalen Parameter wird der eingelesenen Wert zugewiesen
C     (4) Dies führt zur Veränderung des Wertes des aktuellen Parameters
      T = Z
C     Rückgabewert der Funktion festlegen
      LIESZAHL = Z
      END
```

Die Veränderung des Wertes der Variablen TWERT lässt sich jetzt leicht wie folgt erklären:

1. Zunächst wird TWERT mit dem Wert 99.99 initialisiert.
2. TWERT wird beim Aufruf des FUNCTION Unterprogramms LIESZAHL als aktueller Parameter eingetragen. Da FORTRAN call by reference durchführt, bewirken Veränderungen des entsprechenden formalen Parameters Änderungen des Wertes des aktuellen Parameters.
3. T ist der formale Parameter des FUNCTION Unterprogramms LIESZAHL. *T wird im Unterprogramm wie eine lokale Variable deklariert, erhält jedoch keinen eigenen Speicherplatz zugewiesen!*
4. Dem formalen Parameter wird der in die Variable Z eingelesene Wert 11.11 zugewiesen. Damit wird auch der Wert des aktuellen Parameters von 99.99 nach 11.11 verändert!

Das dem FORTRAN-Programm CALLBREF.FOR entsprechende Java-Programm bsp02065.CallByValue enthält folgenden Programmcode:

```
package bsp02065;
...
public class CallByValue
{

  public static void main(String[] args)
  {
...
// (1) Variable für Test anlegen
    double tWert = 99.99;

// zWert einlesen und dabei Test-Variable tWert an Methode liesZahl(...) übergeben
// (2) tWert ist aktueller Parameter, wird nicht verändert wegen call by value
    double zWert = liesZahl(tWert);

// Werte ausgeben
...
  }

  private static double liesZahl(double t)
// (3) t ist formaler Parameter
  {
    double z;
...
    z = KeyboardIn.kReadDouble();
// (4) eingelesenen Wert dem formalen Parameter t zu Test-Variable tWert zuweisen
// (4) dies führt nicht zur Veränderung des Wertes des aktuellen Parameters
    t = z;
```

```
// eingelesenen Wert als Rückgabewert der Methode festlegen
    return z;
  }
}
```

Diese Verarbeitung läßt sich wie folgt erklären:

1. Zunächst wird die Variable tWert deklariert und mit dem Wert 99.99 initialisiert.
2. tWert wird beim Aufruf der Methode liesZahl(..) als aktueller Parameter eingetragen. Da Java call by reference vornimmt, wird der formale Parameter t mit dem Wert 99.99 der Variablen tWert gefüllt.
3. t ist formaler Parameter und eine Variable mit eigenem Speicherplatz, obwohl sie in der Liste der Variablen des Unterprogramms nicht auftritt.
4. t wird der eingelesene Wert 11.11 zugewiesen. Da t jedoch über einen eigenen Speicherplatz verfügt, wird der Wert des aktuellen Parameters tWert nicht verändert.

2.7.2.4 call by value für Referenztypen

Auch für Referenztypen gilt die Regel, dass bei Aufruf einer Methode eine Übergabe des Wertes des aktuellen Parameters 'by value' erfolgt. Allerdings ist - wie im Kapitel über Referenztypen bereits angesprochen - der Wert eines Referenztyps die Referenz, also der Verweis auf das zugehörige Objekt im Speicher. Es gelten daher bei der Verwendung von Referenztypen folgende Regeln:

- So wie der Zuweisungsoperator eine Referenzzuweisung vornimmt, erfolgt beim Aufruf einer Methode ebenfalls eine Referenzzuweisung. Die Folge hierzu ist, dass anschließend **aktueller und formaler Parameter auf das selbe Objekt im Speicher verweisen**.
- Eine Veränderung von Variablen des Objektes bei Zugriff über den formalen Parameter ändert den Wert der Variablen im Objekt. Greift man später wieder über den aktuellen Parameter, also über den Objektnamen in der rufenden Methode auf die Variable des Objektes zu, so findet man einen geänderten Wert.

De facto liegt also für Objekte ein call by value vor, dennoch werden die Werte der Variablen verändert, als ob call by reference vorliegen würde. Man spricht daher von einem **simulierten call by reference**.

Die in Bezug auf die Werte in den Variablen unterschiedliche Wirkungsweise des call by value-Aufrufs bei einfachen Datentypen und Referenzvariablen muss unbedingt beachtet werden, da hier erhebliche und aufwändig zu untersuchende Laufzeitfehler entstehen können. Das unterschiedliche Verhalten wird daher an den Beispielprogrammen bsp02065.CallByValue und bsp02067.CallByValue erläutert. Im bereits oben erläuterten Beispiel bsp02065 werden einfache Datentypen verwendet und es ergibt sich keine Veränderung des Wertes des aktuellen Parameters. Ersetzt man die Variable `tWert` im Programm durch einen Array aus einer double Komponente mit Namen tWert, so erhält man das Beispielprogramm bsp02067.CallByValue:

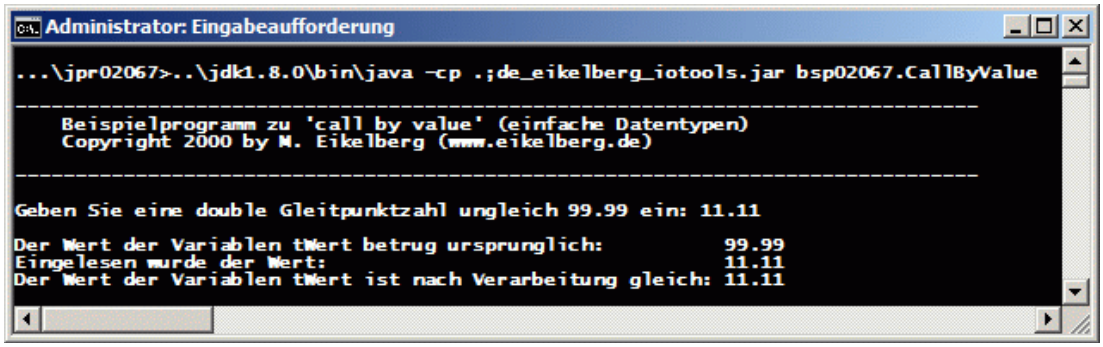
```
package bsp02067;
...
public class CallByValue
{

  public static void main(String[] args)
  {
...
// (1) Variable für Test anlegen - Array ist Referenztyp
    double[] tWert = {99.99};

// zWert einlesen und dabei Test-Variable tWert an Methode liesZahl(...) übergeben
// (2) tWert ist aktueller Parameter,
//     es wird wegen call by value eine Kopie der Referenz übergeben
    double zWert = liesZahl(tWert);

// Werte ausgeben
    ScreenOut.kPrintln();
    ScreenOut.kPrintln(
      "Der Wert der Variablen tWert betrug ursprünglich:          99.99");
    ScreenOut.kPrintln(
      "Eingelesen wurde der Wert:                                 "+zWert);
    ScreenOut.kPrintln(
      "Der Wert der Variablen tWert ist nach Verarbeitung gleich: "+tWert[0]);
  }

  private static double liesZahl(double[] t)
// (3) t ist formaler Parameter,
//     nimmt die Kopie der Referenz auf
  {
    double z;
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Geben Sie eine double Gleitpunktzahl ungleich 99.99 ein: ");
    z = KeyboardIn.kReadDouble();
```

```
// (4) eingelesenen Wert dem formalen Parameter t zu Test-Variable tWert zuweisen
// (4) dies führt zur Veränderung der Variablen des Objektes
    t[0] = z;
// eingelesenen Wert als Rückgabewert der Methode festlegen
    return z;
  }
}
```

Diese Verarbeitung läßt sich wie folgt erklären:

1. Zunächst wird der Array tWert deklariert und mit einem Array aus einem einzelnen Wert 99.99 initialisiert.
2. tWert wird beim Aufruf der Methode liesZahl(..) als aktueller Parameter eingetragen. Da Java call by reference vornimmt, wird der formale Parameter t mit dem Wert der Referenz zum Array tWert gefüllt.
3. t ist formaler Parameter und eine Variable mit eigenem Speicherplatz. Die Variablen tWert und der formale Parameter t verfügen also über jeweils eigene Speicherplätze, Inhalt der Speicherplätze ist jedoch die Referenz auf das selbe Objekt.
4. Der einzigen Komponente t[0] des Arrays wird der eingelesene Wert 11.11 zugewiesen. Da hier kein Zugriff auf eine Referenz, sondern auf einen Speicherplatz des Arrays vorgenommen wird, wird der Inhalt des Arrays verändert!

Das unterschiedliche Verhalten der Programme bsp02065.CallByValue und bsp02067.CallByValue kann man beobachten, wenn man den oben gezeigten Bildschirmausdruck zum Programm bsp02065.CallByValue mit dem folgenden Bildschirmausdruck eines Durchlauf des Programms bsp02067.CallByValue vergleicht. In diesem Fall ist der Wert der Variablen tWert am Ende des Programms verändert:



Wenn man Irrtümer durch die Behandlung von Referenztypen beim Aufruf von Methoden vermeiden will, so kann man zu Beginn der Methode eine Kopie des über die Parameter übergebenen Objektes anlegen und in der Methode nur noch mit dem Objekt arbeiten. Andererseits ist die Verwendung von Objekten ein Ausweg, wenn man Methoden programmieren will, die mehrere Werte verändern. Oft ist die Veränderung des Objektes aber auch beabsichtigt. Es gibt zur Behandlung dieses Phänomens also kein allgemeines Rezept.

2.7.2.5 Ausblick - Anwendung der Bildung von Methoden in der Strukturierten und Klassenbasierten Programmierung

Methoden sind also ein wichtiges Sprachelement, das zur Bildung kleinerer wiederverwendbarer Programmeinheiten eingesetzt wird. Bisher wurde jedoch nur die Bildung von Methoden in der Klasse, die main(...) enthält, erklärt.

- Zweckmäßiger wäre jedoch eine sinnvolle **Gruppierung und Aufbewahrung von Methoden in mehrfach verwendbaren Klassen**, auf die von der Methode main(...) aus Bezug genommen werden kann. Diese Idee wird durch Bildung von Klassen im Rahmen der "**Klassenbasierten Programmierung**" realisiert (vgl. Kapitel 2.8).
- Zuvor soll jedoch die ebenfalls wichtige Umsetzung des Prinzips der **Trennung von Steuerung und Verarbeitung** der **Strukturierten Programmierung** mit Hilfe von Methoden an einem Beispiel erläutert werden (Kapitel 2.7.3).

2.7.3 Beispiel zur Trennung von Steuerung und Verarbeitung

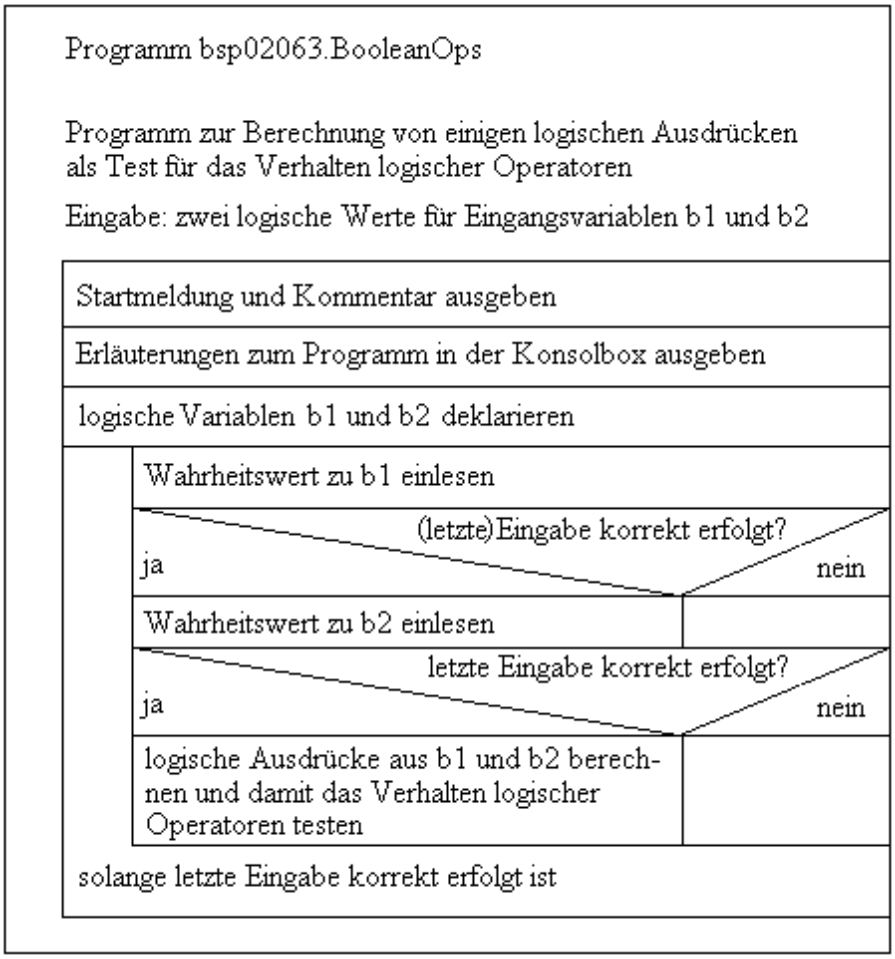
2.7.3.1 Anwendung von Methoden zur Trennung von Steuerung und Verarbeitung

Eine der **Regeln der Strukturierten Programmierung** ist die Vorschrift, Steuerung und Verarbeitung zu trennen. Bei Anwendung von Methoden wird dies jetzt möglich. Insofern bedürfen viele bisher angeführten Beispielprogramme in Bezug auf die Trennung von Steuerung und Verarbeitung der Überarbeitung. Ein Beispiel hierzu ist das Programm bsp02025.BooleanOps, das Wahrheitswerte zu logischen Ausdrücken berechnet. Wir überarbeiten dieses Programm zu einem Programm bsp02063.BooleanOps wie folgt:

2.7.3.2 Die Steuerung des Programms bsp02063.BooleanOps

Das Prinzip der Trennung von Steuerung und Verarbeitung bedeutet, dass man so weit möglich alle Verarbeitungsschritte von Programmen in Methoden auslagert und in der Methode main() möglichst nur steuernde Anweisungen codiert. Ferner sollte man versuchen, in main() lediglich die grobe Programmstruktur zu implementieren. Die Gestaltung geeigneter Modulblöcke zu main() sowie zu den verarbeitenden Methoden erreicht man mit Hilfe von Struktogrammen.

Führt man eine entsprechende Analyse für das Programm bsp02025.BooleanOps durch, so findet man folgenden (oder einen ähnlichen) Modulblock als Modulblock zur Steuerung des Programms:



Ein solches Struktogramm zur Steuerung des Programms ist der Ausgangspunkt einer systematischen Softwareentwicklung. In diesem Struktogramm sind die *verarbeitenden Schritte in Strukturblöcken zusammengefasst* . Zu diesen Strukturblöcken werden im Rahmen der weiteren Programmentwicklung Modulblöcke erarbeitet, die die jeweilige Verarbeitung spezifizieren und die die Basis für die verarbeitenden Methoden sind.

Das obige Struktogramm zur Steuerung kann an dieser Stelle der Softwareentwicklung bereits (von einem zweiten Softwareentwickler) codiert, mit wirkungslosen Methoden bzw. Test-Methoden ergänzt und mit Hilfe eines Debuggers auf Korrektheit der Steuerung überprüft werden. Man erhlät bei der Codierung des Struktogramms folgenden (oder ähnlichen Programmcode):

```
  public static void main(String[] args)
  {
// Startmeldung
```

```
    ScreenOut.kStartScreenOut(
      "Beispielprogramm zu logischen Operatoren",
      "M. Eikelberg (www.eikelberg.de)","2000");

// Erläuterungen auf den Konsolbildschirm schreiben
    schreibeErlaeuterungen();

// Wahrheitswerte für die Verarbeitung deklarieren
// Initialisierung der Wahrheitswerte ist notwendig
    boolean b1=false,b2=false;

// Schleife für die Verarbeitung
    do
    {
// ersten Wahrheitswert einlesen
      b1 = liesWahrheitswert("b1");

// Bei korrekter Eingabe zweiten Wahrheitswert einlesen
      if (status == 0)
        b2 = liesWahrheitswert("b2");

// Tests und Ausgabe durchführen, wenn Eingaben korrekt sind
      if (status == 0)
        performTests(b1,b2);

// Ende der Schleife für die Verarbeitung
    }  while (status == 0);
  }
```

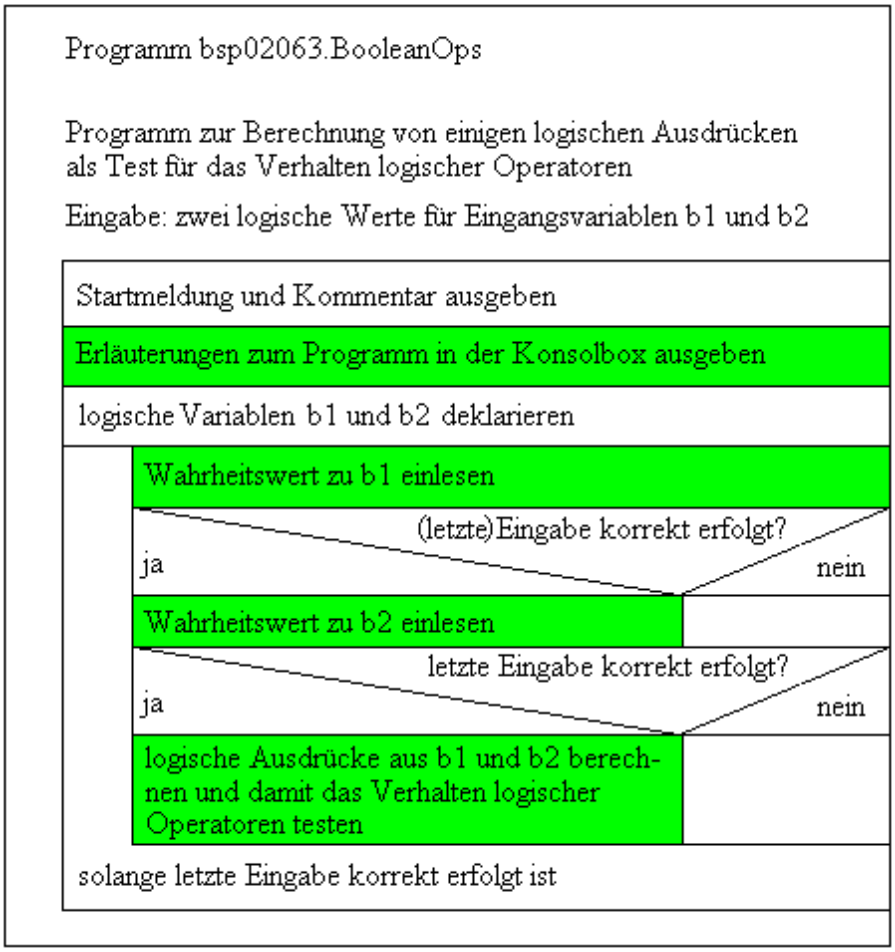
Eine solche Codierung einzelner Modulblücke ist sehr zweckmäig. Die wichtigsten Vorteile bestehen darin, dass man

- für größere Softwareprojekte die einzelnen Aufgaben isoliert, hierdurch
- die Fehlersuche erleichtert,
- die Komplexität des Programmierprojektes reduziert,
- die Bearbeitung von Programmieraufgaben durch mehrere Programmierer ermöglicht und
- so ganz erheblich die Softwareentwicklung beschleunigt.

Die Realisierung dieser Vorteile ist auch deshalb bedeutsam, weil manche Softwareprojekte viele 'Mann-Jahre' (n Personen arbeiten m Jahre = n*m Mann-Jahre) Arbeitszeit erfordern und damit bei einer Bearbeitung von einem Programmierer niemals zu dem Zeitpunkt fertig würden, an dem die Programme benötigt werden.

2.7.3.4 Die Methoden zur Verarbeitung in bsp02063.BooleanOps

Ist die Steuerung des Programms geklärt, so kann mit der Erarbeitung der verarbeitenden Teile des Programms begonnen werden. Hierzu werden jetzt alle verarbeitenden Strukturblöcke analysiert und Modulblock-Struktogramme zu diesen verarbeitenden Teilen des Programms erarbeitet. Die verarbeitenden Strukturblöcke des Programms bsp02063.BooleanOps sind in dem folgenden Struktogramm markiert:



Im Struktogramm des Programms bsp02063.BooleanOps findet man also folgende drei Verarbeitungsschritte:

Verarbeitungsschritt	Methode	Parameter	Rückgabewert
Erläuterungen zum Programm in der Konsolbox ausgeben	private static void schreibeErlaeuterungen()	- keine -	- keiner -
Einlesen eines Wahrheitswertes	private static boolean liesWahrheitswert(String varName)	String-Objekt varName, das den Namen der Variablen zum einzulesenden Wahrheitswert enthält	eingelesener Wahrheitswert als boolean Wert
logische Ausdrücke aus b1 und b2 berechnen und damit das Verhalten logischer Operatoren testen.	private static void performTests(boolean b1, boolean b2)	zwei Wahrheitswerte, die für die Tests verwendet werden	- keiner -

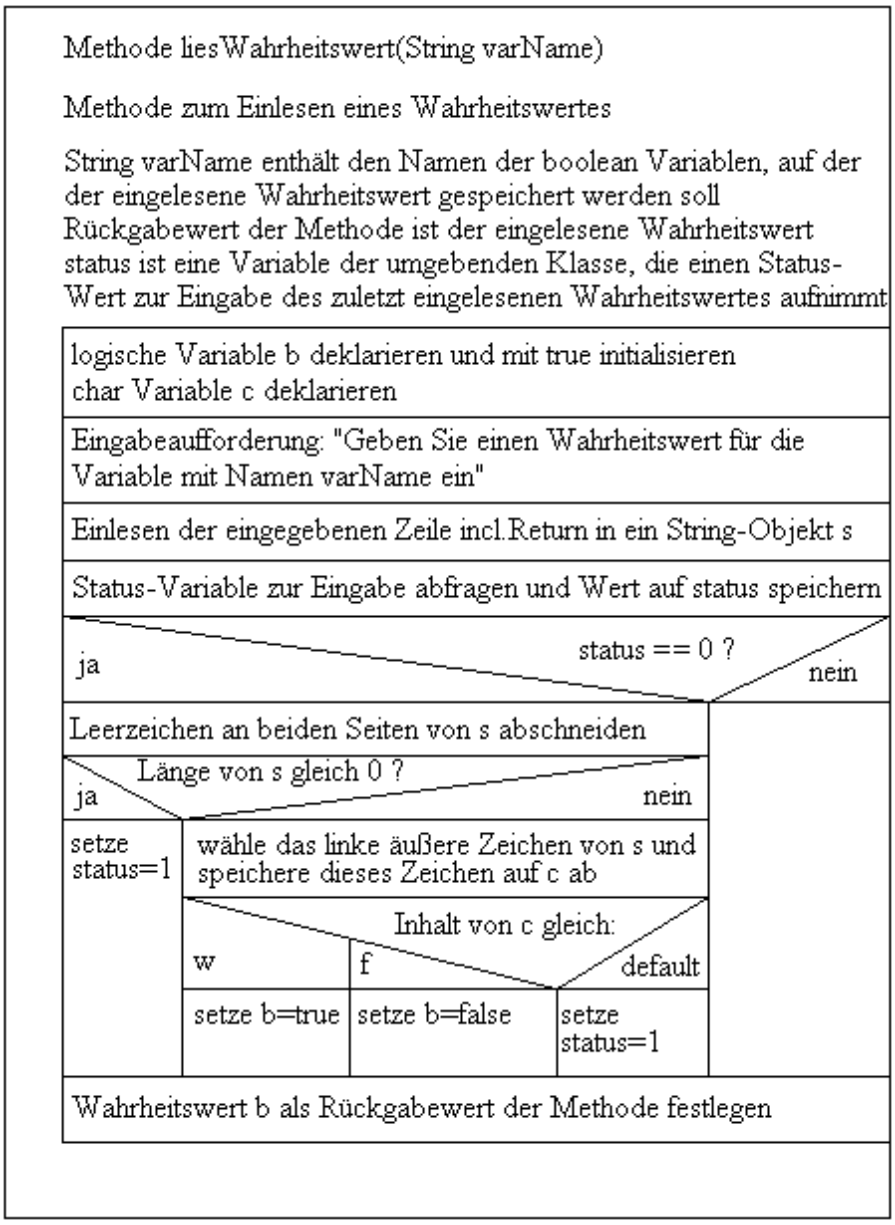
Die Methode liesWahrheitswert(String) wird im Programm bsp02063.BooleanOps zweifach verwendet, die übrigen beiden Methoden werden jeweils nur ein einziges Mal aufgerufen.

Der nächste Schritt der Programmentwicklung ist die Erstellung von Struktogrammen zu diesen drei Methoden. Die Struktogramme zu den Methoden schreibeErläuterungen() und performTests(boolean,boolean) sind denkbar einfach und werden hier nicht angeführt. Der Programmcode zu diesen beiden Methoden lautet wie folgt:

```
  private static void schreibeErlaeuterungen()
  {
// Erläuterung über die Beendigung des Programms
    ScreenOut.kPrintln("Das Programm wird beendet durch eine falsche Eingabe");
    ScreenOut.kPrintln("oder durch Eingabe von (genau nur) F6!");

// Erläuterung der Funktionsweise des Programms
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("Das Programm berechnet folgende Ausdrücke:");
    ScreenOut.kPrintln("  Test 1:   b11 = !b1;");
    ScreenOut.kPrintln("  Test 2:   b12 = b1 | b2;");
    ScreenOut.kPrintln("  Test 3:   b13 = b1 & b2;");
    ScreenOut.kPrintln("  Test 4:   b14 = b1 ^ b2;");
  }
```

```
  private static void performTests(boolean b1, boolean b2)
  {
    boolean b11,b12,b13,b14;
// Tests
    b11 = !b1;
    b12 = b1 | b2;
    b13 = b1 & b2;
    b14 = b1 ^ b2;
// Ergebnisse ausgeben
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("Ergebnisse:");
    ScreenOut.kPrintln("  Test 1:   "+b11+" = !"+b1);
    ScreenOut.kPrintln("  Test 2:   "+b12+" = "+b1+" | "+b2);
    ScreenOut.kPrintln("  Test 3:   "+b13+" = "+b1+" & "+b2);
    ScreenOut.kPrintln("  Test 4:   "+b14+" = "+b1+" ^ "+b2);
  }
```

Etwas mehr Logik enthält das Struktogramm der Methode liesWahrheitswert(String). Das Struktogramm zu dieser Methode sieht wie folgt aus:



Das Struktogramm zeigt, dass es Schwierigkeiten bei der Verarbeitung falscher Eingaben gibt. Die Methode selbst kann nur die Rückgabewerte `true` oder `false` haben und bietet daher nicht die Möglichkeit, über den Rückgabewert der Methode falsche Eingaben anzuzeigen. Um dieses Problem zu lösen, wird in der Klasse bsp02063.BooleanOps eine Klassenvariable

```
  private static int status = 0;
```

deklariert. Sowohl main() als auch die Methode liesWahrheitswert(String) greifen auf diese Klassenvariable zu. Der folgende Java Quellcode

ist eine Umsetzung des Struktogramms zur Methode liesWahrheitswert(String):

```
  private static boolean liesWahrheitswert(String varName)
  {
// Verarbeitung ist gegenüber bsp02025.BooleanOps unter Verwendung der
// Sprachmittel aus Kapitel 2.6 und 2.7 verbessert worden
    boolean b = true;
    char c;
// Werte einlesen
    ScreenOut.kPrintln();
    ScreenOut.kPrint("Geben Sie einen Wahrheitswert für "+varName+" ein [w für true, f für
false] : ");
    String s = KeyboardIn.kReadString();
// Status auf Klassenvariable status abspeichern
    status = KeyboardIn.kGetStatus();
// Eingabe auswerten wenn Eingabe in Ordnung ist
    if (status == 0)
    {
      s = s.trim();
      if (s.length() == 0)
        status = 1;
      else
      {
        c = s.charAt(0);
        switch (c)
        {
          case 'w':
            b = true;
            break;
          case 'f':
            b = false;
            break;
          default:
            status = 1;
        }
      }
    }
// Gelesenen char Wert zurückgeben
    return b;
  }
```

Mit dieser Trennung von Steuerung und Verarbeitung ist jetzt das Programm bsp02063.BooleanOps in kleinere Einheiten (**modularisiert**) zerlegt worden. Was jetzt noch fehlt, ist die Organisation dieser Einheiten in Klassen, so dass diese Einheiten wiederverwendbar werden. Eine solche Anordnung und Gruppierung der Methoden geschieht in der klassenbasierten Programmierung.

2.8 Klassenbasierte Programmierung

2.8.1 Der Begriff der klassenbasierten Programmierung

Bereits nach kurzer Programmiererfahrung entsteht beim Programmierer das Bedürfnis, eigenen oder fremden Programmcode wiederverwenden zu können. Der erste Schritt hierzu ist die in Kapitel 2.7 erklärt Bildung von Methoden. Diese Methoden sollten im zweiten Schritt - möglichst zusammen mit den zugehörigen Variablen - wiederverwendbar gruppiert werden. Bei der Umsetzung dieses zweiten Schritts kommt es jedoch wesentlich darauf an, dass die verwendete Programmiersprache und/oder die verwendete Entwicklungsumgebung eine einfache, elegante und zweckmäßige Technologie hierzu anbieten.

Java bietet für die Gruppierung von Methoden und Daten eine sehr elegante Lösung, die auf der Bildung von **Klassen** und **Paketen** basiert. Die Paketbildung geschieht durch Anordnung von Klassen in Unterverzeichnissen, wobei durch die gewählte Unterverzeichnisstruktur - wie in den Kapiteln 1.2.5 und 2.1 erklärt - der Paketname bestimmt wird. Damit bildet die Paketbildung die 'oberste' Struktur zur Organisation des Programmcode.

Die nächste Struktur unterhalb des Programmpaketes ist die der Klasse. Durch die Bildung von Klassen können Methoden und Daten sehr übersichtlich angeordnet werden. Einen Programmierstil,

- bei dem Methoden und Daten in Klassen angeordnet werden und
- bei dem Klassen als 'Bauplan' für zu instantiierende Objekte verwendet werden

nennt man **klassenbasiert**. Entsprechend spricht man von **klassenbasierter Programmierung**.

Die klassenbasierte Programmierung ist eine der drei Abstufungen der objektorientierten Programmierung **(Vgl. Horn/Kerner, Band 3, S. 270)**:

1. **objektbasierte** Programmierung
2. **klassenbasierte** Programmierung
3. **(volle) objektorientierte** Programmierung

Etwas unglücklich ist bei dieser Begriffsdefinition, dass der Begriff objektorientierte Programmierung sowohl als allgemeiner Oberbegriff auftritt als auch als spezielle Bezeichnung der obersten Stufe. Dies hat sicherlich zu der nicht unerheblichen Begriffsverwirrung um die objektorientierte Programmierung geführt. Zur Klärung wird in diesem Kurs in Zukunft für die dritte Stufe der objektorientierten Programmierung die Bezeichnung *volle* objektorientierte Programmierung verwendet.

Welche Stufe der Objektorientierung im jeweiligen Programm realisiert werden kann, hängt wesentlich davon ab, ob die zu verwendende Programmiersprache geeignete Sprachmittel zur Umsetzung bereitstellt. So kann zum Beispiel bei Verwendung der Standard-Versionen der Programmiersprachen **PASCAL** und **FORTRAN 77** überhaupt keine objektorientierte Programmierung realisiert werden. Die Programmiersprache **JavaScript** erlaubt dagegen eine objektbasierte Programmierung, verfügt aber nicht über alle Funktionalitäten von PASCAL und FORTRAN 77. Die Programmiersprachen **C++** und **Java** ermöglichen die Codierung von voll objektorientierten Programmen.

> Eine genauere Erläuterung der drei Stufen der objektorientierten Programmierung erfolgt in Kapitel 2.9.1. Dort finden Sie auch erste Beispiele, die die Bedeutung der Objektorientierung für die Java-Programmierung aufzeigen.

Bisher haben wir uns in diesem Kurs fast nur im Bereich der objektbasierten Programmierung bewegt. Alle bisherigen Programmbeispiele und Übungsaufgaben dieses Kurses, die lediglich aus einer einzigen Klasse bestehen, sind Beispiele der objektbasierten Programmierung: In diesen Programmierprojekten werden neben der Klasse, die die Methode main(String[] argumente) enthält, keine zusätzlichen Klassen programmiert, sondern höchstens vorhandene fremde Klassen wie zum Beispiel die Klassen des Paketes de.eikelberg.iotools verwendet.

In diesem Kapitel 2.8 wird jetzt der Schritt in die klassenbasierte Programmierung vollzogen. Ziel dabei ist

- eine zweckmäßige Anordnung bzw. Organisation von Programmcode in Klassen
- die Bildung von Klassen als 'Baupläne' für Objekte
- die Verringerung von Codierungsaufwand durch Wiederverwendung von in Klassen vorgehaltenem Programmcode.

Zugleich ist das Erlernen der klassenbasierten Programmierung Voraussetzung für das Erlernen der vollen objektorientierten Programmierung und damit Voraussetzung für die Nutzung von Techniken wie Vererbung, Polymorphie und Überschreiben.

2.8.2 Syntax der Klassendeklaration

Die Syntax der Klassendeklaration ist in ihrer Grundform sehr einfach:

```
class Klassenname
{
// Programmcode zur Klasse
}
```

Bei der Klassendeklaration muss folgendes beachtet werden:

- Als *Klassenname* wird der für die jeweilige Klasse gewählte Klassenname eingetragen.
- Nach allgemeiner Konvention sollten Klassennamen stets mit einem Großbuchstaben beginnen.
- Der Klassenname einer als `public` gekennzeichneten Klasse muss mit dem Namen der Datei identisch sein, in der die Klasse codiert wird. Ist die Klasse nicht `public`, so können Klassenname und Dateinamen verschieden sein.
- Klassen- und Dateinamen sind case-sensitive, das heisst es wird zwischen Großschreibung und Kleinschreibung unterschieden!
- In jeder Klassendatei können mehrere Klassen codiert werden. Davon muss eine Klasse den Namen der Datei haben, in der die Klassen gespeichert sind.
- Der gesamte Java Quellcode zu einer Klasse muss in den Block zur Klasse eingefügt werden. Ausserhalb der Klassendeklaration dürfen nur die package und import Anweisung sowie Kommentare eingefügt werden.

Eine Klasse mit Namen `MeineKlasse` muss also in einer Datei `MeineKlasse.java` codiert werden und unterscheidet sich damit von einer Klasse `Meineklasse`, die in einer Datei `Meineklasse.java` codiert werden muss. Man beachte in diesem Zusammenhang folgende Empfehlungen:

- Von der Verwendung von Klassen innerhalb eines Paketes, deren Namen sich lediglich durch Groß- und Kleinschreibung unterscheiden, wird *dringend* abgeraten. Unter UNIX-Systemen ist dies möglich, unter Windows erhält man jedoch Fehlermeldungen, da Windows-Systeme nicht zwischen den beiden unter UNIX verschiedenen Dateien unterscheiden können! Man erhält bei der Portierung von UNIX nach Windows eine sicher unter Windows nicht lauffähige Anwendung!
- Auch wenn unter Windows-Systemen keine Unterscheidung zwischen Groß- und Kleinschreibung existiert und damit auch nicht zwischen `MeineKlasse.java` und `Meineklasse.java` unterschieden wird, sollte penibel auf die richtige Groß- und Kleinschreibung geachtet werden. Unterschiede in der Schreibung zwischen Dateinamen und Klassennamen werden unter Windows nicht bemerkt, führen jedoch unter UNIX zu Fehlern.
- Beachten Sie ferner, dass Groß- und Kleinschreibung bei der Übertragung von Dateien von Windows nach UNIX geändert werden kann. Keine Probleme gibt es, wenn die Dateinamen aus Kleinbuchstaben bestehen, weshalb man bei der HTML-Programmierung auch alle Dateinamen in Kleinbuchstaben gewählt werden sollten. Unglücklicherweise besagt jedoch die Konvention, dass Java-Klassennamen und damit auch die Dateinamen mit Großbuchstaben beginnen sollen. Prüfen Sie daher bei Problemen nach Übertragung von Java-Anwendungen von Windows nach UNIX die Groß- und Kleinschreibung!

Wie bereits in Kapitel 2.2.1 erklärt können ausserhalb der Klassendeklaration nur noch Kommentare sowie import- und package-Anweisungen in der Java-Quelldatei codiert werden. Eine einfache Java-Klassendatei hat damit einen Aufbau wie die folgende Quelldatei bsp02069.MeineKlasse:

```
// Titel:         Kapitel 2 Beispiel 69
// Version:       1.00
// Copyright:     Copyright (c) 2000
// Autor:         Prof. Dr. Markus Eikelberg
// Beschreibung:  Beispiel zur Klassendeklaration

package bsp02069;

import de.eikelberg.iotools.*;

@SuppressWarnings("unused")  // Unterdrücken der Warnung wg. unbenutztem Import
class MeineKlasse
{
// In dieser Klasse steht kein Java Quellcode
}
```

Die **import-Anweisung** macht alle public Bestandteile (vgl. 2.3.1.4 und 2.8.7) des Paketes de.eikelberg.iotools in der Klasse MeineKlasse verfügbar. Die **package-Anweisung** fügt die Klasse MeineKlasse in das Paket bsp02069 ein, wodurch alle protected und alle public Bestandteile (vgl. 2.3.1.4 und 2.8.7) der übrigen Klassen des Paketes bsp02069 in der Klasse MeineKlasse nutzbar werden (vgl. Kapitel 2.3.3 und 2.8.4). package- und import-Anweisung regeln also zusammen mit den Modifizierern private, protected und public das Verhältnis der Klasse MeineKlasse und ihrer Bestandteile zu anderen Klassen.

Auch die Klasse MeineKlasse kann als public oder protected deklariert werden. Die Deklaration der Klasse MeineKlasse als private oder

static macht jedoch keinen Sinn und ist daher auch nicht erlaubt (vgl. Kapitel 2.8.4).

2.8.3 Klassen- und Instanzvariablen

2.8.3.1 lokale Variablen, Klassenvariablen und Instanzvariablen

Der Begriff der klassenbasierten Programmierung bedeutet wie bereits in 2.8.1 angesprochen, dass Variablen und Methoden (thematisch gegliedert und unter Bezug auf ihre logische und programmtechnische Abhängigkeit) in Klassen gruppiert werden. Wie in 2.3.3 erklärt sind diese Klassen 'Baupläne' für die Objekte, die ausgehend von den Klassen instantiiert werden. Ein im aktuellen Zusammenhang wichtiger zum Objekt-Begriff äquivalenter Begriff ist der der Instanz: Ein Objekt ist eine Instanz einer Klasse und wird daher auch als **Instanz** bezeichnet. Im Zusammenhang mit Variablen und Methoden ist aus historischen Gründen statt des Begriffs Objekt der Begriff Instanz üblich. Daher spricht man zum Beispiel auch von Klassen- und Instanzvariablen. *Entsprechend wird in diesem Kapitel oft statt des Begriffs Objekt der Begriff Instanz verwendet.*

In Kapitel 2.3.3 wurde bereits angesprochen, dass es jedoch nicht immer notwendig ist, Instanzen von Klassen zu bilden. So wurde in 2.3.3.3 und 2.3.3.4 der Unterschied bei der Nutzung von statischen und nicht statischen Bestandteilen einer Klasse erklärt. Die Eigenschaft statisch oder nicht statisch konnte daran erkannt werden, ob die entsprechende Variable oder Methode mit dem Modifizierer **static** als statisch gekennzeichnet wurde. Für den Zugriff auf statische und nicht-statische Bestandteile gelten folgende Regeln:

- Für die Nutzung oder den Zugriff auf eine Instanzvariable muss stets eine Instanz der Klasse gebildet werden. Der Zugriff auf die Variable erfolgt wie in 2.3.3.3 erklärt über den Namen der Instanz, also über den Objektnamen.
- Die Nutzung oder der Zugriff auf eine Klassenvariable kann ohne die Bildung einer Instanz wie in 2.3.3.3 erklärt über den Klassennamen erfolgen. Es ist aber auch möglich, auf die (selbe) Variable über jeden Objektnamen zu Instanzen der Klasse zuzugreifen. In allen Fällen erfolgt der Zugriff auf den selben Speicherplatz.

In diesem Kapitel befassen wir uns zunächst mit den in einer Klasse deklarierten Variablen. Hierzu unterscheiden wir drei Typen von Variablen:

- Eine **lokale Variable** ist eine innerhalb einer Methode deklarierte Variable. Diese ist in ihrer Gültigkeit bzw. Lebensdauer beschränkt auf den Bereich des kleinsten Blocks, der ihre Variablendeklaration umfasst. Damit ist eine lokale Variable niemals ausserhalb einer Methode gültig!
- Eine **Klassenvariable** ist eine ausserhalb von Methoden direkt in der Klasse deklarierte Variable, die zusätzlich als **static** gekennzeichnet ist. Eine solche Klassenvariable existiert für jede Klasse genau ein mal. Insbesondere wird sie bei der Bildung von Instanzen nicht dupliziert. Vielmehr ist in jeder Instanz die selbe einzige Klassenvariable vorhanden.
- Eine **Instanzvariable** ist eine Variable, zu der für jedes Objekt - also für jede Instanz - ein eigenes Exemplar angelegt wird. Von den Instanzvariablen gibt es also genau so viele Exemplare, wie es Instanzen zu der Klasse gibt, in der die Instanzvariable deklariert ist. Eine Instanzvariable erkennt man daran, dass der Modifizierer static *fehlt* .

Wir verdeutlichen den Unterschied von lokalen Variablen, Klassenvariablen und Instanzvariablen an den Beispielprogrammen bsp02070.ClassVar, bsp02071.InstanceVar und bsp02073.LocalVar.

2.8.3.2 Beispiel zu Klassenvariablen

Wir beginnen zunächst mit der folgenden Deklaration einer Klassenvariablen i in einer Klasse bsp02070.VarDeklaration. Die Datei VarDeklaration.java muss wie üblich im Unterverzeichnis bsp02070 des Java-Basisverzeichnisses angeordnet werden. Verwendet man Eclipse, so geschieht das Anlegen der Datei am besten unter Verwendung der Auswahl 'neu' des 'Datei'-Menüs. Im daraufhin erscheinenden Fenster wähle man 'Klasse' aus. Bei dem Anlegen der neuen Klasse wähle man jetzt anders als bisher die Option 'Main-Funktion generieren' *nicht* aus. Zum Abschluss verändern wir den Inhalt der neu angelegten Datei so, dass sie schließlcih folgenden Java Quellcode enthält:

```
package bsp02070;

import de.eikelberg.iotools.*;

@SuppressWarnings("unused")  // Unterdrücken der Warnung wg. unbenutztem Import
public class VarDeklaration
{
// Klassenvariable
  static int i;
}
```

In der Klasse VarDeklaration ist also keine Methode enthalten, sondern lediglich eine int Variable i, die mit dem Modifizierer `static` als statisch gekennzeichnet ist. Diese Klassenvariable i wird in der Methode main(String[] args) der Klasse bsp02070.ClassVar verwendet:

```
package bsp02070;

import de.eikelberg.iotools.*;

public class ClassVar
{
```

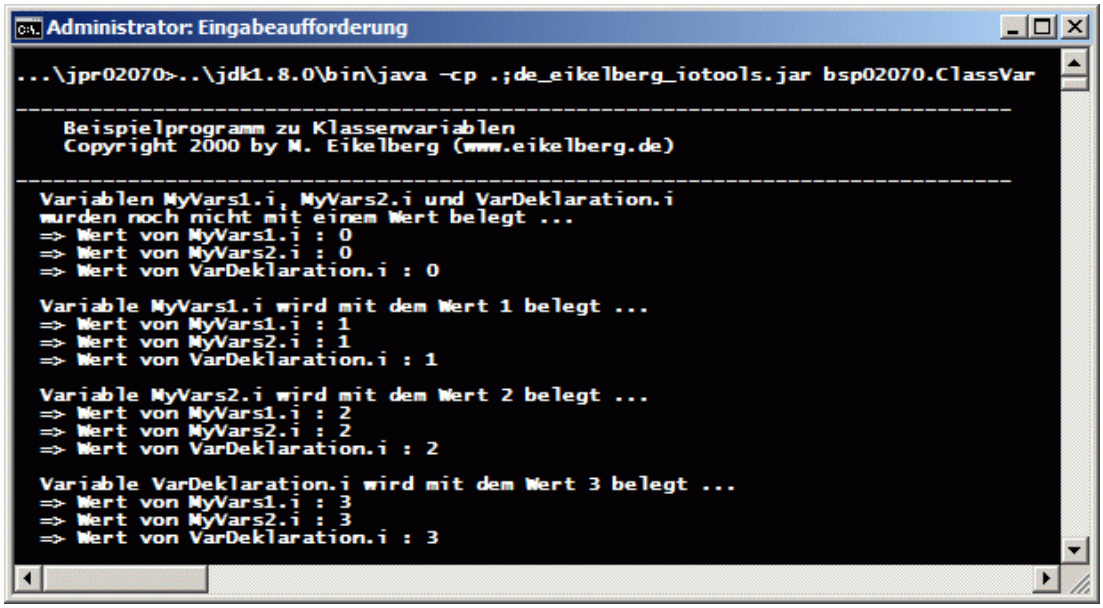
```
// Unterdrücken der Warnung "The static field ... should be accessed in a static way":
  @SuppressWarnings("static-access")
  public static void main(String[] args)
  {
// 0. Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispielprogramm zu Klassenvariablen",
      "M. Eikelberg (www.eikelberg.de)","2000");

// 1. Zwei Instanzen der Klasse VarDeklaration erzeugen (vgl. Kap. 2.3.3)
    VarDeklaration MyVars1;
    MyVars1 = new VarDeklaration();
    VarDeklaration MyVars2;
    MyVars2 = new VarDeklaration();

// 2. Verarbeitung mit nicht initialisierter Klassenvariable
    ScreenOut.kPrintln("  Variablen MyVars1.i, MyVars2.i und VarDeklaration.i");
    ScreenOut.kPrintln("  wurden noch nicht mit einem Wert belegt ...");

// 3. Abfrage der Werte von MyVars1.i, MyVars2.i und VarDeklaration.i
// Hinweis: Statische Variablen können über Objektnamen angesprochen werden.
// Es ist aber guter Programmierstil, auf diese Variablen über den Klassennamen
// zuzugreifen.
    ScreenOut.kPrintln("  => Wert von MyVars1.i : " + MyVars1.i);
    ScreenOut.kPrintln("  => Wert von MyVars2.i : " + MyVars2.i);
    ScreenOut.kPrintln(
      "  => Wert von VarDeklaration.i : " + VarDeklaration.i);
    ScreenOut.kPrintln();

// 4. Zuweisung eines Wertes zu i über die Instanz MyVars1
    ScreenOut.kPrintln("  Variable MyVars1.i wird mit dem Wert 1 belegt ...");
// Siehe Hinweis zu Schritt 3!
    MyVars1.i = 1;

// 5. Abfrage der Werte von MyVars1.i, MyVars2.i und VarDeklaration.i
// Siehe Hinweis zu Schritt 3!
    ScreenOut.kPrintln("  => Wert von MyVars1.i : " + MyVars1.i);
    ScreenOut.kPrintln("  => Wert von MyVars2.i : " + MyVars2.i);
    ScreenOut.kPrintln(
      "  => Wert von VarDeklaration.i : " + VarDeklaration.i);
    ScreenOut.kPrintln();

// 6. Zuweisung eines Wertes zu i über die Instanz MyVars2
    ScreenOut.kPrintln("  Variable MyVars2.i wird mit dem Wert 2 belegt ...");
// Siehe Hinweis zu Schritt 3!
    MyVars2.i = 2;

// 7. Abfrage der Werte von MyVars1.i, MyVars2.i und VarDeklaration.i
// Siehe Hinweis zu Schritt 3!
    ScreenOut.kPrintln("  => Wert von MyVars1.i : " + MyVars1.i);
    ScreenOut.kPrintln("  => Wert von MyVars2.i : " + MyVars2.i);
    ScreenOut.kPrintln(
      "  => Wert von VarDeklaration.i : " + VarDeklaration.i);
    ScreenOut.kPrintln();

// 8. Zuweisung eines Wertes über die Klasse VarDeklaration
    ScreenOut.kPrint  ("  Variable VarDeklaration.i wird mit dem Wert 3");
    ScreenOut.kPrintln(" belegt ...");
    VarDeklaration.i = 3;

// 9. Abfrage der Werte von MyVars1.i, MyVars2.i und VarDeklaration.i
// Siehe Hinweis zu Schritt 3!
    ScreenOut.kPrintln("  => Wert von MyVars1.i : " + MyVars1.i);
    ScreenOut.kPrintln("  => Wert von MyVars2.i : " + MyVars2.i);
    ScreenOut.kPrintln(
      "  => Wert von VarDeklaration.i : " + VarDeklaration.i);
  }
}
```

In diesem Programm werden die folgenden Verarbeitungsschritte durchgeführt:

1. Instantiieren zweier Objekte MyVars1 und MyVars2 der Klasse VarDeklaration.
2. Keine Initialisierung der Klassenvariablen i der Klasse VarDeklaration.
3. Abfrage des Wertes von i über die Instanzen MyVars1, MyVars2 sowie über den Klassennamen VarDeklaration.

4. Zuweisung des Wertes 1 zur Klassenvariablen i über die Instanz MyVars1.
5. Abfrage des Wertes von i über die Instanzen MyVars1, MyVars2 sowie über den Klassennamen VarDeklaration.
6. Zuweisung des Wertes 2 zur Klassenvariablen i über die Instanz MyVars2.
7. Abfrage des Wertes von i über die Instanzen MyVars1, MyVars2 sowie über den Klassennamen VarDeklaration.
8. Zuweisung des Wertes 3 zur Klassenvariablen i über den Klassennamen VarDeklaration.
9. Abfrage des Wertes von i über die Instanzen MyVars1, MyVars2 sowie über den Klassennamen VarDeklaration.

Bevor das Programm bsp02070.ClassVar ausgeführt werden kann, müssen beide Klassen des Paketes bsp02070 in Bytecode compiliert werden. Da dies ohne Fehlermeldung durchgeführt wird, ist zugleich auch klar, dass die Zugriffe in bsp02070.ClassVar über Instanzen und über den Klassennamen gültigen Java-Code darstellen. Dennoch wird stets mit der selben statischen Variablen i der Klasse VarDeklaration gearbeitet, denn die Verarbeitungsschritte 3,5,7 und 9 ergeben bei Änderung über lediglich einen der drei Namen stets für alle drei Namen MyVars1.i, MyVars2.i und VarDeklaration.i stets den selben Wert, wie der folgende Bildschirmausdruck zeigt:



Der Bilschirmausdruck zeigt jedoch auch, dass die Klassenvariable i bereits den Wert 0 hat, ohne dass sie ausdrücklich initialisiert wurde. Dies liegt daran, dass Klassen- und Instanzvariablen stets automatisch initialisert werden (vgl. 2.8.3.6).

2.8.3.3 Beispiel zu Instanzvariablen

Wir führen jetzt im Beispiel bs02071.InstanceVar die Verarbeitung von bsp02070.ClassVar in analoger Weise für eine Instanzvariable j durch. Hierzu wird in einer Klasse bsp02071.VarDeklaration zunächst eine nicht statische Variable i deklariert:

```
package bsp02071;

import de.eikelberg.iotools.*;

@SuppressWarnings("unused")  // Unterdrücken der Warnung wg. unbenutztem Import
public class VarDeklaration
{
// Instanzvariable
  int i;
}
```

Die Verarbeitung von bsp02071.InstanceVar besteht aus allen Anweisungen von bsp02070.ClassVar, die sich auf die Instanzen MyVars1 und MyVars2 beziehen. Die Anweisungen, die sich auf die Klasse VarDeklaration bezogen, müsen entfernt werden, da für Instanzvariablen kein Zugriff über den Klassennamen möglich ist. bsp02071.InstanceVar besteht aus folgendem Programmcode:

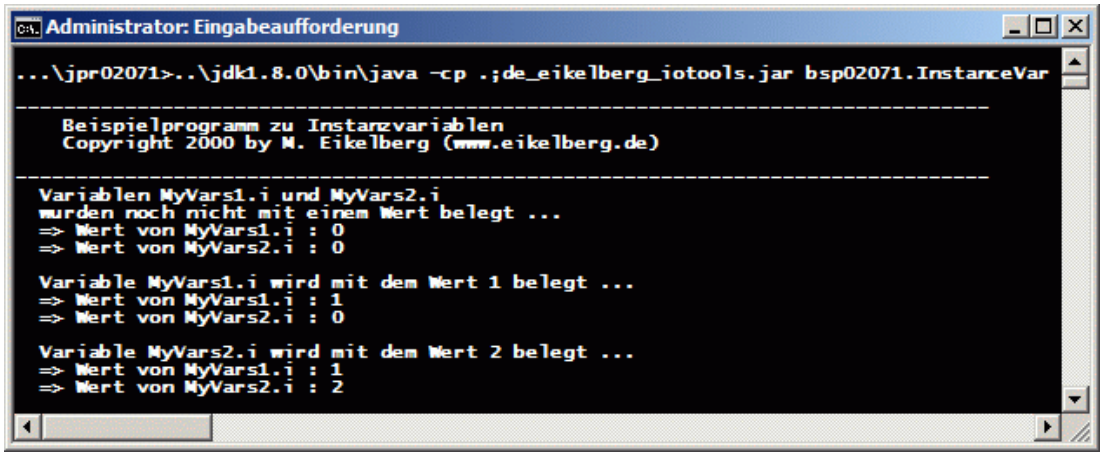
```
package bsp02071;

import de.eikelberg.iotools.*;

public class InstanceVar
{

  public static void main(String[] args)
  {
// 0. Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispielprogramm zu Instanzvariablen",
```

```
      "M. Eikelberg (www.eikelberg.de)","2000");

// 1. Zwei Instanzen der Klasse VarDeklaration erzeugen (vgl. Kap. 2.3.3)
    VarDeklaration MyVars1;
    MyVars1 = new VarDeklaration();
    VarDeklaration MyVars2;
    MyVars2 = new VarDeklaration();

// 2. Verarbeitung mit nicht initialisierter Instanzvariablen
    ScreenOut.kPrintln("  Variablen MyVars1.i und MyVars2.i");
    ScreenOut.kPrintln("  wurden noch nicht mit einem Wert belegt ...");

// 3. Abfrage der Werte von MyVars1.i und MyVars2.i
    ScreenOut.kPrintln("  => Wert von MyVars1.i : " + MyVars1.i);
    ScreenOut.kPrintln("  => Wert von MyVars2.i : " + MyVars2.i);
    ScreenOut.kPrintln();

// 4. Zuweisung eines Wertes zu i über die Instanz MyVars1
    ScreenOut.kPrintln("  Variable MyVars1.i wird mit dem Wert 1 belegt ...");
    MyVars1.i = 1;

// 5. Abfrage der Werte von MyVars1.i und MyVars2.i
    ScreenOut.kPrintln("  => Wert von MyVars1.i : " + MyVars1.i);
    ScreenOut.kPrintln("  => Wert von MyVars2.i : " + MyVars2.i);
    ScreenOut.kPrintln();

// 6. Zuweisung eines Wertes zu i über die Instanz MyVars2
    ScreenOut.kPrintln("  Variable MyVars2.i wird mit dem Wert 2 belegt ...");
    MyVars2.i = 2;

// 7. Abfrage der Werte von MyVars1.i und MyVars2.i
    ScreenOut.kPrintln("  => Wert von MyVars1.i : " + MyVars1.i);
    ScreenOut.kPrintln("  => Wert von MyVars2.i : " + MyVars2.i);
    ScreenOut.kPrintln();
  }
}
```

Es werde also die folgenden Verarbeitungsschritte durchgeführt:

1. Instantiieren zweier Objekte MyVars1 und MyVars2 der Klasse VarDeklaration.
2. Keine Initialisierung der Klassenvariablen i der Klasse VarDeklaration.
3. Abfrage des Wertes von i über die Instanzen MyVars1 und MyVars2.
4. Zuweisung des Wertes 1 zur Klassenvariablen i über die Instanz MyVars1.
5. Abfrage des Wertes von i über die Instanzen MyVars1 und MyVars2.
6. Zuweisung des Wertes 2 zur Klassenvariablen i über die Instanz MyVars2.
7. Abfrage des Wertes von i über die Instanzen MyVars1 und MyVars2.

Ein Durchlauf von bsp02071.InstanceVar zeigt bei der Verarbeitung der Schritte 3, 5 und 7, dass die Variablen MyVars1.i und MyVars2.i voneinander unabhängig sind und unterschiedliche Werte haben können. Schritt 3 zeigt ferner, dass auch Instanzvariablen automatisch mit einem Wert initialisiert werden (vgl. 2.8.3.6). Die Verarbeitung von bsp02071.InstanceVar ergibt folgenden Bildschirmausdruck:



2.8.3.4 Beispiel zu lokalen Variablen

Lokale Variable existieren nach Definition lediglich innerhalb von Methoden und sind damit ausserhalb der Klassen nicht nutzbar. Daher muss ein Beispielprogramm zu lokalen Variablen anders als die Beispielprogramms bsp02070.ClassVar und bsp02071.InstanceVar aufgebaut sein.

Zudem kann es vorkommen, dass in einer Klasse lokale Variablen und Instanz- oder Klassenvariablen codiert sind, die den gleichen Namen haben. In diesem Fall **überdeckt** die lokale Variable die Instanz- oder Klassenvariable und ein Zugriff auf die Instanz- oder Klassenvariable erfordert die Verwendung des Zeigers **this** (vgl. 2.8.3.5).

Lokale Variablen können nur in Methoden deklariert sein. Daher enthält die Beispiel-Klasse bsp02072.VarDeklaration eine Methode drucke(), in der die lokale Variable k deklariert, initialisiert und ausgedruckt wird:

```
package bsp02072;

import de.eikelberg.iotools.*;

public class VarDeklaration
{
// Methode drucke
  void drucke()
  {
// lokale Variable
    int k=1;
    ScreenOut.kPrintln("  => Wert der lokalen Variablen k : " + k);
// Da die Variable das Ende der Methode nicht 'überlebt',
// hat die folgende Anweisung keine Auswirkung:
    k = k+1;
  }
}
```

Durch die letzte Anweisung der Methode drucke() wird der Wert der Variablen k um 1 erhöht. Da mit dem Ende der Methode drucke() auch die Lebensdauer der Variablen k endet, wirkt sich diese Erhöhung jedoch nicht auf den nächsten Durchlauf der Methode drucke() aus. Dieser Sachverhalt wird anhand des Beispielprogramms bsp02072.LocalVar demonstriert:

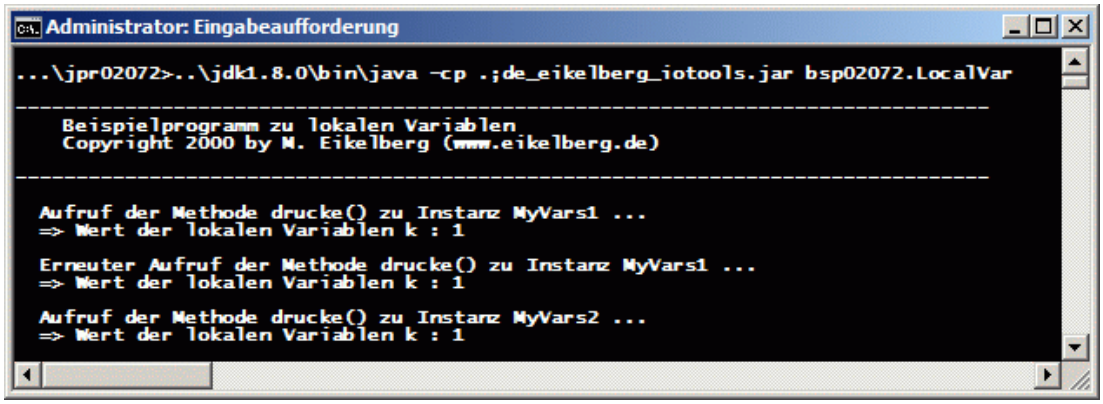
```
package bsp02072;

import de.eikelberg.iotools.*;

public class LocalVar
{

  public static void main(String[] args)
  {
// 0. Startmeldung
...
// 1. Zwei Instanzen der Klasse VarDeklaration erzeugen
    VarDeklaration MyVars1;
    MyVars1 = new VarDeklaration();
    VarDeklaration MyVars2;
    MyVars2 = new VarDeklaration();

// 2. Ausdruck des Wertes von k zur Instanz MyVars1
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("  Aufruf der Methode drucke() zu Instanz MyVars1 ...");
    MyVars1.drucke();
    ScreenOut.kPrintln();

// 3. Erneuter Ausdruck des Wertes von k zur Instanz MyVars1
    ScreenOut.kPrintln("  Erneuter Aufruf der Methode drucke() zu Instanz MyVars1 ...");
    MyVars1.drucke();
    ScreenOut.kPrintln();

// 4. Ausdruck des Wertes von k zur Instanz MyVars2
    ScreenOut.kPrintln("  Aufruf der Methode drucke() zu Instanz MyVars2 ...");
    MyVars2.drucke();
  }
}
```

Es werde also die folgenden Verarbeitungsschritte durchgeführt:

1. Instantiieren zweier Objekte MyVars1 und MyVars2 der Klasse VarDeklaration.
2. Ausdruck des Wertes von k zur Instanz MyVars1
3. Erneuter Ausdruck des Wertes von k zur Instanz MyVars1
4. Ausdruck des Wertes von k zur Instanz MyVars2

Die Verarbeitungsschritte 2 bis 4 ergeben stets den gleichen Wert. Dennoch sind die drei beteiligten Variablen k verschieden, da sie zu unterschiedlichen Zeiten während des Programmablaufs existieren: Sie werden jeweils während des Durchlaufs der Methode drucke()

angelegt und am Ende der Methode drucke() wieder gelöscht, wie man auch an der Bildschirmausgabe von bsp02072.LocalVar erkennen kann:



2.8.3.5 Überdecken von Instanz- und Klassenvariablen durch lokale Variable und der Zeiger this

Die Sprachdefinition von Java erlaubt es, in einer Klasse lokale Variablen zu deklarieren, die den gleichen Namen wie eine Instanz- oder Klassenvariable der Klasse haben, in der die Methode mit der lokalen Variablen enthalten ist. Ein Beispiel hierzu ist die Klasse bsp02073.VarDeklaration, die eine Instanzvariable k und eine lokale Variable k enthält:

```
package bsp02073;

import de.eikelberg.iotools.*;

public class VarDeklaration
{
// Instanzvariable k
  int k=1;

// Methode drucke
  void drucke()
  {
// lokale Variable k
    int k=2;
    ScreenOut.kPrintln("  => Wert der Instanzvariablen k  : " + this.k);
    ScreenOut.kPrintln("  => Wert der lokalen Variablen k : " + k);
  }
}
```

Die Methode drucke() gibt sowohl den Wert der Instanzvariablen als auch den Wert der lokalen Variablen aus. Zur Unterscheidung der beiden gleichnamigen Variablen gibt es den sogenannten **Zeiger this**:

Gibt es in einer Klasse sowohl eine Klassen- oder Instanzvariable als auch in einer Methode eine lokale Variable gleichen Namens, so geschieht der Zugriff auf diese Variablen in der Methode wie folgt:

- Wird in der Methode lediglich der Variablenname codiert, so erfolgt ein Zugriff auf die lokale Variable. Man sagt daher, dass die lokale Variable die Klassen- oder Instanzvariable **überdeckt**.
- Soll in der Methode ein Zugriff auf die Klassen- oder Instanzvariable erfolgen, so codiert man vor dem Variablennamen das Schlüsselwort **this** und fügt zwischen this und dem Variablennamen einen Punkt ein. **this** ist eine spezielle Konstruktion in Java und wird als **Zeiger this** bezeichnet.

Man beachte, dass von einigen Programmierern der Zeiger this je nach Situation auch dann verwendet wird, wenn die Klassen- und Instanzvariable nicht von einer lokalen Variablen verdeckt wird. Eine solche Verwendung des Zeigers this kann die Lesbarkeit eines Programms erhöhen, da dem Leser besonders verdeutlicht wird, dass die verwendete Variable eine Klassen- oder Instanzvariable ist. Ein Beispiel für eine solchen Situation ist die Verwendung des Zeigers this in Get- und Set-Methoden (vgl. 2.8.6).

In der Methode drucke() in bsp02073.VarDeklaration wird also auf die Instanzvariable k mit `this.k` zugegriffen, auf die lokale Variable jedoch mit `k`.

Im Beispielprogramm bsp02073.UeberdeckeVar werden Instanzen von bsp02073.VarDeklaration gebildet Über diese Instanzen wird mehrfach die Methode drucke() aufgerufen:

```
package bsp02073;

import de.eikelberg.iotools.*;

public class UeberdeckeVar
```

```
{

  public static void main(String[] args)
  {
// 0. Startmeldung
...
// 1. Instanzen der Klasse VarDeklaration erzeugen
    VarDeklaration MyVars1;
    MyVars1 = new VarDeklaration();

// 2. Ausdruck des Wertes der Instanzvariablen und der lokalen Variablen
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("  Aufruf der Methode drucke() zu Instanz MyVars1 ...");
    MyVars1.drucke();
    ScreenOut.kPrintln();

// 3. Veränderung des Wertes der Instanzvariablen
    ScreenOut.kPrintln("  Veränderung des Wertes der Instanzvariablen k ...");
    ScreenOut.kPrintln("  neuer Wert der Instanzvariablen k ist 3 ...");
    MyVars1.k = 3;

// 4. Erneuter Ausdruck des Wertes der Instanzvariablen und der lokalen Variablen
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("  Aufruf der Methode drucke() zu Instanz MyVars1 ...");
    MyVars1.drucke();
  }
}
```

Das Programm führt folgende Verarbeitung durch:

1. Instantiieren eines Objektes MyVars1 der Klasse VarDeklaration
2. Aufruf der Methode drucke() des Objektes MyVars1
3. Änderung des Wertes der Klassenvariablen MyVars1.k
4. Erneuter Aufruf der Methode drucke() des Objektes MyVars1

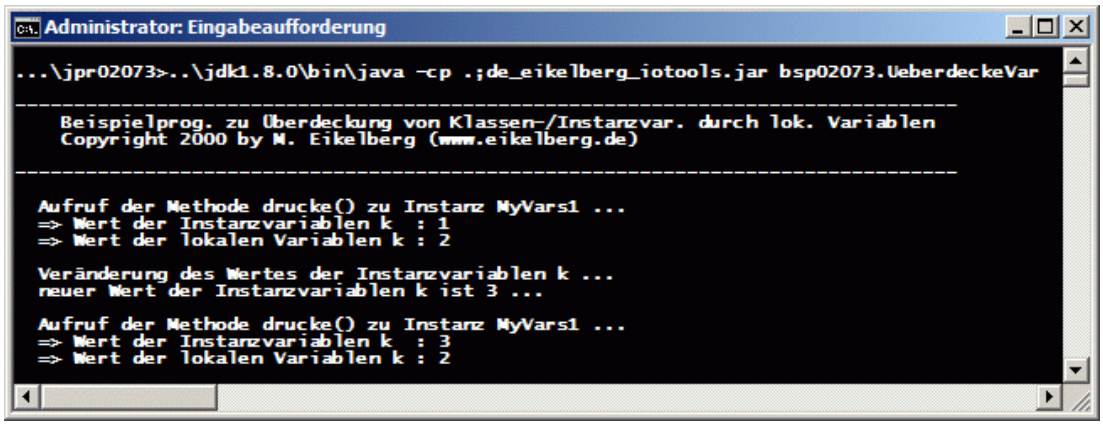
Programmtechnisch äußerst unschön in diesem Beispielprogramm ist im Schritt 3 die direkte Änderung der Instanzvariablen k über den Zugriff MyVars1.k . Empfehlenswerter ist die Änderung des Wertes von MyVars1.k über eine Set-Methode (vgl. 2.8.7).

Ein Ablauf des Beispielprogramms bsp02073.UeberdeckeVar zeigt,

- dass die lokale Variable stets den gleichen Wert hat, da sie jeweils zu Beginn der Methode drucke() deklariert und initialisiert wird und am Ende der Methode drucke() wieder gelöscht wird.
- Die Instanzvariable k dagegen existiert auch ausserhalb der Methode drucke() von der Instantiierung des Objektes MyVars1 an bis zur Löschung des Objektes. Dementsprechend ist auch eine Änderung des Wertes der Instanzvariablen - wie in Schritt 3 des Beispielprogramms - ausserhalb des Ablaufs von Methoden möglich.

Die Löschung des Objektes erfolgt hier bei Programmende, im Allgemeinen sobald die Java Virtuelle Maschine erkennt, dass das Objekt nicht mehr benötigt wird.

Der Bildschirmausdruck von bsp02073.UeberdeckeVar illustriert den zuletzt genannten Sachverhalt sowie die Wirkung des Zeigers this:



Die Instanzvariable k hat zunächst aufgrund der Initialisierung in der Klassendeklaration bsp02073.VarDeklaration den Wert 1. Durch direkten Zugriff auf die Klassenvariable vom Program bsp02073.UeberdeckeVar erhält sie den Wert 3 zugewiesen, der auch beim nächsten Aufruf der Methode drucke() ausgegeben wird. Die lokale Variable k wird dagegen stets beim Start der Methode drucke() neu deklariert und mit dem Wert 2 initialisiert. Daher hat die lokale Variable beim Durchlauf der Methode drucke() stets den Wert 2.

2.8.3.6 Automatische Initialisierung von Instanz- und Klassenvariablen

Im Beispiel bsp02073.VarDeklaration wurde die Instanzvariable k deklariert und zugleich mit dem Wert 1 initialisiert. Eine solche Initialisierung ist nicht explizit notwendig, denn Instanz- und Klassenvariablen werden automatisch mit den in der folgenden Tabelle genannten Werten initialisiert:

Typ der Variablen	Wert nach automatischer Initialisierung (Default-Wert)
boolean	false
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000'
Referenztyp	null-Referenz

'\u0000' bezeichnet wie in 2.3.1.2 beschrieben das Zeichen 'hexadezimal Null' der Unicode-Tabelle. Es handelt sich dabei um ein nicht im Editor darstellbares Zeichen.

Die null-Referenz ist ein Objekt-Name, der durch die Konstante `null` dargestellt wird. Die null-Referenz zeigt auf ein Standard Java-Objekt, das keine Variablen und keine Methoden hat. Dieses Objekt hat zudem die Eigenschaft, das es Instanz jeder beliebigen Klasse sein kann.

Die automatische Initialisierung von Instanz- und Klassenvariablen durch die genannten Default-Werte demonstriert das Beispielprogramm bsp02074.DefaultWert. Hierzu ist zunächst die Codierung der folgenden Klasse bsp02074.VarDefinition erforderlich:

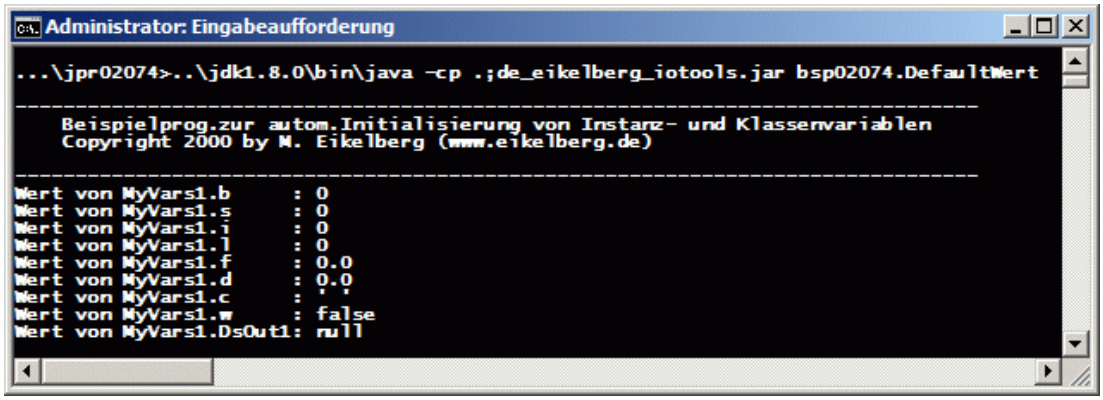
```
package bsp02074;

import de.eikelberg.iotools.*;

public class VarDefinition
{

// Instanzvariablen, nicht initialisiert
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  char c;
  boolean w;
  DatasetOut dsOut1;

}
```

bsp02074.VarDefinition enthält also die Deklaration von Instanzvariablen, die nicht initialisiert werden. Im Hauptprogramm bsp02074.DefaultWert wird jetzt ein Objekt der Klasse VarDefinition instantiiert:

```
package bsp02074;

import de.eikelberg.iotools.*;

public class DefaultWert
{

  public static void main(String[] args)
  {
// Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispielprog.zur autom.Initialisierung von Instanz- und Klassenvariablen",
      "M. Eikelberg (www.eikelberg.de)","2000");

// Objekt der Klasse VarDefinition instantiieren
```

```
    VarDefinition MyVars1;
    MyVars1 = new VarDefinition();

// Inhalt der Instanzvariablen von MyVars1 ausgeben
    ScreenOut.kPrintln("Wert von MyVars1.b     : " + MyVars1.b);
    ScreenOut.kPrintln("Wert von MyVars1.s     : " + MyVars1.s);
    ScreenOut.kPrintln("Wert von MyVars1.i     : " + MyVars1.i);
    ScreenOut.kPrintln("Wert von MyVars1.l     : " + MyVars1.l);
    ScreenOut.kPrintln("Wert von MyVars1.f     : " + MyVars1.f);
    ScreenOut.kPrintln("Wert von MyVars1.d     : " + MyVars1.d);
    ScreenOut.kPrintln("Wert von MyVars1.c     : '" + MyVars1.c +"'");
    ScreenOut.kPrintln("Wert von MyVars1.w     : " + MyVars1.w);
    ScreenOut.kPrintln("Wert von MyVars1.dsOut1: " + MyVars1.dsOut1);
  }
}
```

Nach der Instantiierung des Objektes MyVars wird der Inhalt der Instanzvariablen ausgegeben. Hierbei handelt es sich sowohl um einfache Datentypen als auch den Namen dsOut1 eines (nicht instantiierten) Objektes der Klasse DatasetOut. Der folgende Bildschirmausdruck zeigt die Werte, die die Instanzvariablen bei der Initialisierung des Objektes standardmäßig zugewiesen bekommen:



Möchte man nicht mit diesen Initialisierungs-Werten arbeiten, so ist es auch möglich, Klassen- und Instanzvariablen in einem Initialisierungsblock selbst mit Startwerten zu belegen:

2.8.3.7 Der Initialisierungsblock für Klassenvariablen

Der Initialisierungsblock für Klassenvariablen wird in der Klassendeklaration angeordnet und hat die folgende Form:

```
...
public class Klassenname
{
// Variablendeklaration
  static
  {
// Initialisierung der Variablen
  }
}
```

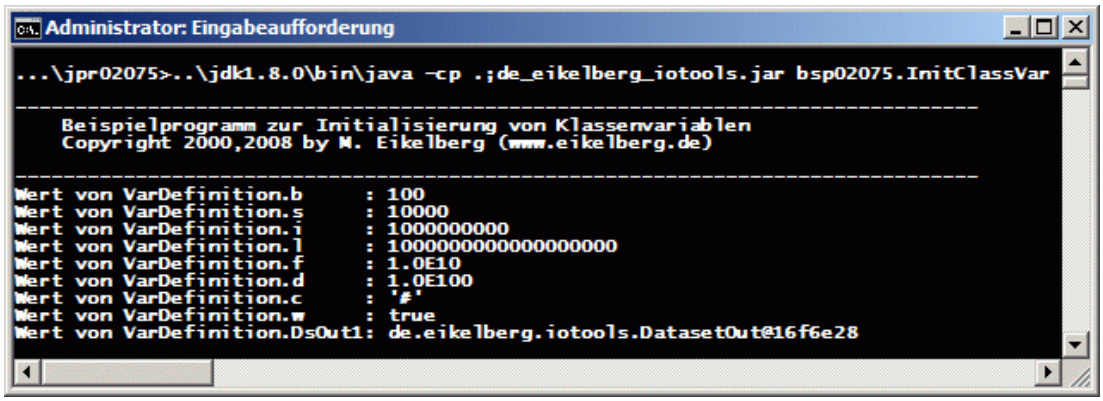
Man beachte, dass die Deklaration der Klassenvariablen nicht im static-Block geschehen darf, da dann die Lebensdauer der Variablen am Schluss des static-Blocks endet!

Das Beispielprogramm bsp02075.InitClassVar besteht im Prinzip aus dem selben Programmcode wie das Beispielprogramm bsp02074.DefaultWert. Ein Unterschied besteht darin, dass in der main-Methode über den Klassennamen auf die Klassenvariablen zugegriffen wird. Der zweite Unterschied besteht im Programmcode der Klasse bsp02075.VarDefinition - dort werden Klassenvariablen deklariert und diese Variablen werden in einem statischen Konstruktor inititalisiert:

```
package bsp02075;

import de.eikelberg.iotools.*;

public class VarDefinition
{

// Deklaration der Klassenvariablen
    static byte b;
    static short s;
    static int i;
```

```
    static long l;
    static float f;
    static double d;
    static char c;
    static boolean w;
    static DatasetOut dsOut1;

// Initialisierung der Klassenvariablen
  static
  {
    b=100;
    s=10000;
    i=1000000000;
    l=1000000000000000000L;
    f=1.0e10f;
    d=1.0e100d;
    c='#';
    w=true;
    dsOut1 = new DatasetOut("daten\\ausgabe.txt");
  }
}
```

Alle Klassenvariablen werden also zunächst direkt in der Klasse deklariert und anschließend im static-Block initialisiert. Der Bildschirmausdruck des Programms bsp02075.InitStaticVar zeigt, dass diese Initialisierung erfolgreich war:



Der Wert zu MyVars1.dsOut1 sieht etwas merkwürdig aus. Angezeigt wird der Wert der sogenannten Referenz, also wie das Objekt MyVars1.dsOut1 im Speicher zu finden ist. Es wird damit jedoch sichtbar, dass MyVars1.dsOut1 nicht mehr auf die Null-Referenz zeigt, sondern auf ein ausdrücklich instantiiertes Objekt.

2.8.3.8 Der Initialisierungsblock für Instanzvariablen

Der Initialisierungsblock für Instanzvariablen hat die folgende Form:

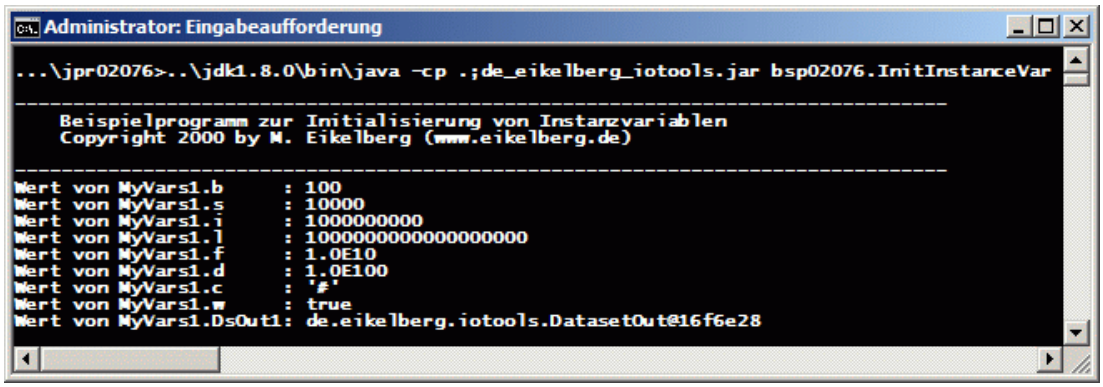
```
...
public class Klassenname
{
// Variablendeklaration
  {
// Initialisierung der Variablen
  }
}
```

Der Initialisierungsblock für Instanzvariablen unterscheidet sich also von dem Block für Klassenvariablen dadurch, dass das Schlüsselwort static fehlt. Durch Entfernen des Schlüsselwortes static aus der Klasse bsp02075.VarDefinition kann man also die folgende Beispielklasse bsp02076.VarDefinition zur Initialisierung von Instanzvariablen ableiten:

```
package bsp02076;

import de.eikelberg.iotools.*;

public class VarDefinition
{

// Deklaration der Instanzvariablen
    byte b;
    short s;
    int i;
```

```
    long l;
    float f;
    double d;
    char c;
    boolean w;
    DatasetOut dsOut1;

// Initialisierung der Instanzvariablen
  {
    b=100;
    s=10000;
    i=1000000000;
    l=1000000000000000000L;
    f=1.0e10f;
    d=1.0e100d;
    c='#';
    w=true;
    dsOut1 = new DatasetOut("daten\\ausgabe.txt");
  }
}
```

Das Hauptprogramm bsp02076.InitInstanceVar entspricht wieder dem Programm bsp02075.InitClassVar und gibt den folgenden Text auf dem Bildschirm aus:



Auch hier sieht man, dass die Inititalisierung der Variablen korrekt erfolgt ist.

2.8.4 Klassen- und Instanzmethoden

2.8.4.1 Begriffserklärung

Mit dem Modifizierer static gekennzeichnete Methoden einer Klasse nennt man **Klassenmethoden**, nicht mit static gekennzeichnete Methoden einer Klasse nennt man dagegen **Instanzmethoden**.

Klassenmethoden werden daher - wie bereits in 2.3.3.4 erklärt - unter Verwendung des *Klassennamens* aufgerufen, Instanzmethoden werden dagegen - wie bereits in 2.3.3.3 erklärt - unter Verwendung des *Objektnamens* aufgerufen. Dieser Sachverhalt liegt darin begründet, dass sich

- Klassenmethoden stets auf die Klasse und
- Instanzmethoden stets auf die jeweilige Instanz

beziehen (sollen). Bei der Frage, ob eine Methode eine Klassen- oder Instanzmethode ist, ist es also entscheidend,

1. ob sich die zu erstellende Methode auf Informationen der Klasse bezieht,
2. ob auf das Instantiieren der Klasse verzichtet werden kann und soll oder
3. ob Instanzen der umgebenden Klasse gebildet werden, auf die die Methode bezug nimmt.

In den ersten beiden Fällen kann eine Klassenmethode entwickelt werden, im dritten Fall muss eine Instanzmethode entworfen werden.

2.8.4.2 Beispiele aus de.eikelberg.iotools

Bereits bekannte Beispiele zu dieser Frage sind die Klassen ScreenOut und DatasetOut. Da ein Arbeitsplatzrechner in aller Regel (logisch) nur mit einem Bildschirm verbunden ist, ist es möglich, alle Variablen und Methoden zur Bildschirmausgabe in einem Objekt anzuordnen, dessen Variablen und Methoden alle statisch sind:

```
package de.eikelberg.iotools;

import java.io.*;

public class ScreenOut
{

  private static CodePage850Writer cpWriter;

// Konstruktor
  public ScreenOut()
  {
    init();
  }

// statischer Initialisierungsblock
  static
  {
    init();
  }

// wirksame Anweisungen von Konstruktor und Initialisierungsblock
  private static void init()
  {
    ...
  }

  public static void kStartScreenOut(String kommentar, String autor, String jahr)
  {
    ...
  }

  public static void kPrintln()
  {
    ...
  }

  public static void kPrintln(boolean aus)
  {
    ...
  }
```

```
public static void kPrint(String aus)
{
  ...
}

public static void kPrint(String aus,int off,int len)
{
  ...
}

public static void kPrintln(String aus,int off,int len)
{
  ...
}

public static void kPrint(char aus[],int off,int len)
{
  ...
}

public static void kPrint(char cbuf[])
{
  ...
}

public static void kPrintln(char aus[],int off,int len)
{
  ...
}

public static void kPrintln(char aus[])
{
  ...
}

public static void kPrintln(String aus)
{
  ...
}

public static void kPrint(char aus)
{
  ...
}

public static void kPrint(boolean b)
{
  ...
}

public static void kPrintln(char aus)
{
  ...
}

public static void kPrint(int aus)
{
  ...
}

public static void kPrint(long aus)
{
  ...
}

public static void kPrintln(long aus)
{
  ...
}

public static void kPrint(float aus)
{
  ...
```

```
  }

  public static void kPrintln(float aus)
  {
    ...
  }

  public static void kPrint(double aus)
  {
    ...
  }

  public static void kPrintln(double aus)
  {
    ...
  }

  public static void kPrintln(int aus)
  {
    ...
  }

  public static void kPrint(byte aus)
  {
    ...
  }

  public static void kPrintln(byte aus)
  {
    ...
  }
}
```

ScreenOut enhält eine Klassenvariable cpWriter sowie eine ganze Anzahl von Klassenmethoden. Nicht statisch (und nicht unbedingt erforderlich) ist einzig der sogenannte Konstruktor (vgl. Kapitel 2.8.5). Ferner enthält die Klasse ScreenOut einen Inititalisierungsblock für Klassenvariablen, der allerdings nicht zur Initialisierung von Klassenvariablen verwendet wird. Vielmehr wird in diesem Initialisierungsblock eine Methode aufgerufen, die einmalig vor Verwendung von Variablen oder Methoden der Klasse ausgeführt wird. Man sieht, dass ein Initialisierungsblock gemäß 2.8.3.7 und 2.8.3.8 auch für Verarbeitungsschritte verwendet werden kann.

ScreenOut ist also Beispiel für eine Klasse, die sich auf eine Thematik bezieht, die maximal ein einziges Mal im Kontext des Programms auftreten kann. Anders ist dies mit Ausgabedateien, von denen ein Programm durchaus mehrere haben kann. Daher ist die Ausgabeverarbeitung in eine Datei ein typisches Beispiel für eine Klasse, von der im Programm (zu jeder Ausgabedatei) Instanzen gebildet werden. Dementsprechend enhält die Klasse DatasetOut in erster Linie Instanzvariablen und Instanzmethoden:

```
package de.eikelberg.iotools;

import java.io.*;
import java.util.*;

public class DatasetOut
{
// Vector definieren, in den die bereits ge►ffneten Dateien eingetragen werden
  static Vector OffeneDateien = new Vector();
...
  private FileWriter OutDatei;
...
  private BufferedWriter OutBuffer;
...
  private PrintWriter OutWriter;
...
  private CodePage1252Writer cpWriter;

  private int dStatus;

  private String DatasetName;

  public DatasetOut(String dsname)
  {
    ...
  }

  public DatasetOut(String dsname, boolean append)
```

```
{
  ...
}

public int dGetStatus()
{
  ...
}

public void dClose()
{
  ...
}

public void dNewLine()
{
  ...
}

public void dPrint(String x)
{
  ...
}

public void dPrint(String x,int off, int len)
{
  ...
}

public void dPrintln(String x)
{
  ...
}

public void dPrintln(String x,int off,int len)
{
  ...
}

public void dPrint(boolean x)
{
  ...
}

public void dPrintln(boolean x)
{
  ...
}

public void dPrint(char x)
{
  ...
}

public void dPrintln(char x)
{
  ...
}

public void dPrint(char x[])
{
  ...
}

public void dPrint(char x[],int off,int len)
{
  ...
}

public void dPrintln(char x[])
{
  ...
}
```

```
  public void dPrintln(char x[],int off,int len)
  {
    ...
  }

  public void dPrint(double x)
  {
    ...
  }

  public void dPrintln(double x)
  {
    ...
  }

  public void dPrint(float x)
  {
    ...
  }

  public void dPrintln(float x)
  {
    ...
  }

  public void dPrint(int x)
  {
    ...
  }

  public void dPrintln(int x)
  {
    ...
  }

  public void dPrint(byte x)
  {
    ...
  }

  public void dPrintln(byte x)
  {
    ...
  }

  public void dPrint(long x)
  {
    ...
  }

  public void dPrintln(long x)
  {
    ...
  }

  public void dPrintln()
  {
    ...
  }
}
```

Die Klasse DatasetOut enthält also mehrere Instanzvariablen sowie eine größere Anzahl von Instanzmethoden. Allerdings enthält DatasetOut auch eine Klassenvariable mit Namen OffeneDateien. Auf diese Klassenvariable wird jedoch nicht aus einer 'normalen' Methode geschrieben, sondern lediglich aus den sogenannten Konstruktoren, der beim Anlegen eines Objektes genau einmal durchlaufen wird (vgl. 2.8.5). Inhaltlich hat die Klassenvariable OffeneDateien den Sinn, dass vom Programm für die Ausgabe geöffnete Dateien beim Anlegen des zugehörigen Objektes in der Variablen OffeneDateien verzeichnet werden, so dass ein zweimaliges Öffnen einer Ausgabedatei aus einem Programm verhindert werden kann.

2.8.4.3 Verhältnis von Instanz- und Klassenmethoden zu Instanz- und Klassenvariablen

Die Beispiele in 2.8.4.2 zeigen, dass inhaltliche Aspekte ausschlaggebend sind bei der Frage, ob Instanz- oder Klassenmethoden entwickelt

werden. Aus Gründen des Software Engineerings (Wartbarkeit, Komplexität bei der Fehlersuche etc) werden hierzu folgende Grundsätze aufgestellt:

1. Klassenmethoden sollten sich möglichst nur auf Klassenvariablen beziehen und keinesfalls Instanzvariablen Daten zuweisen.
2. Instanzvariablen sollten lediglich aus Instanzmethoden heraus Werte zugewiesen werden.
3. Instanzmethoden (ausgenommen Konstruktoren) sollten sich bei schreibendem Zugriff nur auf Instanzvariablen beziehen und Klassenvariablen lediglich lesen.
4. Klassenvariablen sollten lediglich aus Klassenmethoden oder aus Konstruktoren (vgl. 2.8.5) Werte zugewiesen werden.

Regel 1 und 2 sowie Regel 3 und 4 stehen jeweils in deutlichen engem Zusammenhang. Die Beachtung der Regeln 1 und 2 ist dabei besonders wichtig. Daher ist der Zugriff aus Klassenmethoden auf Instanzvariablen auch nur mit einem Trick möglich, so dass die Missachtung der 1. Regel nur absichtlich geschehen kann. Ohne besondere Maßnahmen bei der Codierung verhindert der Compiler den Zugriff aus einer Klassenmethode auf eine Instanzvariable (vgl. 2.8.4.5).

Wir überprüfen die Beachtung dieser Grundsätze bei der Codierung von ScreenOut:

- Regel 1 und 2 sind beachtet - ScreenOut enthält keine Instanzvariablen.
- Regel 3 und 4 sind beachtet - ScreenOut enthält (außer einem Konstruktor) keine Instanzmethoden.

Wir überprüfen jetzt die Beachtung der Grundsätze bei der Codierung von DatasetOut:

- Regel 1 und 2 sind beachtet - DatasetOut enthält keine Klassenmethoden.
- Regel 3 ist beachtet - die Instanzmethoden (ausgenommen die Konstruktoren) verwenden die Klassenvariable nicht.
- Regel 4 ist beachtet - die Klassenvariable von DatasetOut wird lediglich von den Konstruktoren verwendet.

2.8.4.4 Beispiel zum Zugriff von Instanz- und Klassenmethoden auf Instanz- und Klassenvariablen

Den Zugriff von Instanz- und Klassenmethoden auf Instanz- und Klassenvariablen verdeutlicht das Beispielprogramm bsp02077.CountObjects. Dieses Programm verwendet die Klasse bsp02077.MyClass, die aus folgendem Programmcode aufgebaut ist:

```
package bsp02077;

import de.eikelberg.iotools.*;

public class MyClass
{

// Instanzvariable
  int i;
// Klassenvariable - initialisiert mit Default-Wert 0
  static int j;

// Inititalisierungsblock für Instanzvariablen
  {
    i=10;
    countObjects();
  }

// Klassenmethode countObjects
  static void countObjects()
  {
    j = j+1;
    ScreenOut.kPrintln(
      "Es wird das "+j+". Objekt der Klasse MyClass instantiiert...");
    ScreenOut.kPrintln();
  }

// Instanzmethode advanceVar_i()
  void advanceVar_i()
  {
    i = i+1;
  }

// Instanzmethode sPrint()
  void sPrint()
  {
// Zugriff auf Instanzvariable i
    ScreenOut.kPrintln("Inhalt der Instanzvariablen i: " + i);
// Zugriff auf Klassenvariable j
    ScreenOut.kPrintln("Inhalt der Klassenvariablen j: " + j);
  }
}
```

Die Klasse bsp02077.MyClass enthält also eine Instanzvariable i eine Klassenvariable j, eine Klassenmethode countObjects() sowie zwei Instanzmethoden advanceVar_i() und sPrint().

- Die Klassenvariable j wird nicht ausdrücklich initialisiert und erhält somit den Default-Wert 0.
- Die Instanzvariable i wird ien einem Initialisierungsblock für jedes Objekt zunächst auf den Wert 10 gesetzt.
- Ferner wird im Initialisierungsblock die Methode countObjects() aufgerufen, die somit für jedes Objekt einmal bei der Instantiierung des Objektes durchlaufen wird.
- Die Klassenmethode countObjects() erhöht den Wert der Klassenvariablen j um eins und gibt eine Meldung auf dem Bildschirm aus. Diese Meldung zeigt an, dass ein Objekt der Klasse MyClass instantiiert wird.
- Die Instanzmethode advanceVar_i() erhöht den Wert der Instanzvariablen i um eins.
- Die Instanzmethode sPrint() druckt den Wert der Instanzvariablen i und der Klassenvariablen j auf dem Bildschirm aus.

Im Hauptprogramm bsp02077.CountObjects werden zwei Objekte der Klasse MyClass instantiiert:

```
package bsp02077;

import de.eikelberg.iotools.*;

public class CountObjects
{

  public static void main(String[] args)
  {
// 0. Startmeldung
    ...

// 1. erstes Objekt der Klasse MyClass instantiieren
    MyClass MyObj1;
    MyObj1 = new MyClass();

// 2. Werte der Instanz- und Klassenvariablen ausdrucken
    ScreenOut.kPrintln("Die Instanzvariablen haben folgende Werte:");
    MyObj1.sPrint();
    ScreenOut.kPrintln();

// 3. zweites Objekt der Klasse MyClass instantiieren
    MyClass MyObj2;
    MyObj2 = new MyClass();

// 4. Werte der Instanz- und Klassenvariablen ausdrucken
    ScreenOut.kPrintln("Die Instanzvariablen haben folgende Werte:");
    MyObj2.sPrint();
    ScreenOut.kPrintln();

// 5. Instanzvariable i in Objekt MyObj2 erhöhen
    ScreenOut.kPrintln("Wert der Instanzvariablen des zweiten Objektes erhöhen...");
    ScreenOut.kPrintln();
    MyObj2.advanceVar_i();

// 6. Werte der Instanz- und Klassenvariablen ausdrucken
    ScreenOut.kPrintln("Die Instanzvariablen des 1. Objektes haben folgende Werte:");
    MyObj1.sPrint();
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("Die Instanzvariablen des 2. Objektes haben folgende Werte:");
    MyObj2.sPrint();
  }
}
```

Die Verarbeitung von bsp02077.CountObjects läuft wie folgt ab:

1. Es wird ein Objekt MyObj1 der Klasse MyClass instantiiert.
2. Mit der Instanzmethode sPrint() werden die Werte der Instanzvariable MyObj1.i und der Klassenvariable MyClass.j ausgedruckt.
3. Es wird ein zweites Objekt MyObj2 der Klasse MyClass instantiiert.
4. Mit der Instanzmethode sPrint() werden die Werte der Instanzvariable MyObj2.i und der Klassenvariable MyClass.j ausgedruckt.
5. Mit der Instanzmethode advanceVar_i() wird der Wert der Instanzvariablen MyObj2.i um eins erhöht.
6. Mit der Instanzmethode sPrint() werden über jedes Objekt die Werte der Instanzvariableni und der Klassenvariable j ausgedruckt.

Mit jedem Instantiieren eines Objektes erhöht sich die Klassenvariable um eins, da im Initialisierungsblock für Instanzvariablen die Methode countObjects() aufgerufen wird. Ferner wird beim Instantiieren eines Objektes der Wert der jeweiligen Instanzvariablen i auf 10 gesetzt. Insgesamt wird mit *drei* Variablen gearbeitet: *einer* Klassenvariablen j und *für jedes Objekt eine* Instanzvariable i.

Ein Durchlauf von bsp02077.CountObjects ergibt somit folgenden Bildschirmausdruck:

```
Administrator: Eingabeaufforderung
...\jpr02077>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02077.CountObjects
--------------------------------------------------------------------------------
    Beispielprog. zur Verwendung von Instanz- und Klassenvariablen in Methoden
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Es wird das 1. Objekt der Klasse MyClass instantiiert...

Die Instanzvariablen haben folgende Werte:
Inhalt der Instanzvariablen i: 10
Inhalt der Klassenvariablen j: 1

Es wird das 2. Objekt der Klasse MyClass instantiiert...

Die Instanzvariablen haben folgende Werte:
Inhalt der Instanzvariablen i: 10
Inhalt der Klassenvariablen j: 2

Wert der Instanzvariablen des zweiten Objektes erhöhen...

Die Instanzvariablen des 1. Objektes haben folgende Werte:
Inhalt der Instanzvariablen i: 10
Inhalt der Klassenvariablen j: 2

Die Instanzvariablen des 2. Objektes haben folgende Werte:
Inhalt der Instanzvariablen i: 11
Inhalt der Klassenvariablen j: 2
```

2.8.4.5 Der Zugriff auf Instanzvariablen aus einer Klassenmethode heraus

Generell sollten Klassenmethoden gemäß Regel 1 und 2 aus 2.8.4.3 möglichst überhaupt nicht und wenn überhaupt dann nur lesend Instanzvariablen verwenden. Soll derartiges ausnahmsweise doch einmal codiert werden, so muss die Verbindung der Klassenmethode mit der Instanz, zu der die Variable gehört, mit besonderen Maßnahmen hergestellt werden. Der Compiler verhindert nämlich zunächst, dass ein derartiger Zugriff geschieht. Dies ist sinnvoll, da der Compiler nicht erkennen kann, zu welche Instanz die Instanzvariable gehören soll. Hierzu folgendes Beispiel-Klasse bsp02087.VarDefinition:

```
package bsp02078;

public class VarDefinition
{
// Instanzvariable
  int i;
// Klassenvariable
  int j;

// Instanzmethode
  void advanceByTwo()
  {
    i = i+2;
  }

// Klassenmethode
  static void advanceByThree()
  {
    i = i+1;            // Zeile 19
    advanceByTwo();     // Zeile 20
  }
}
```

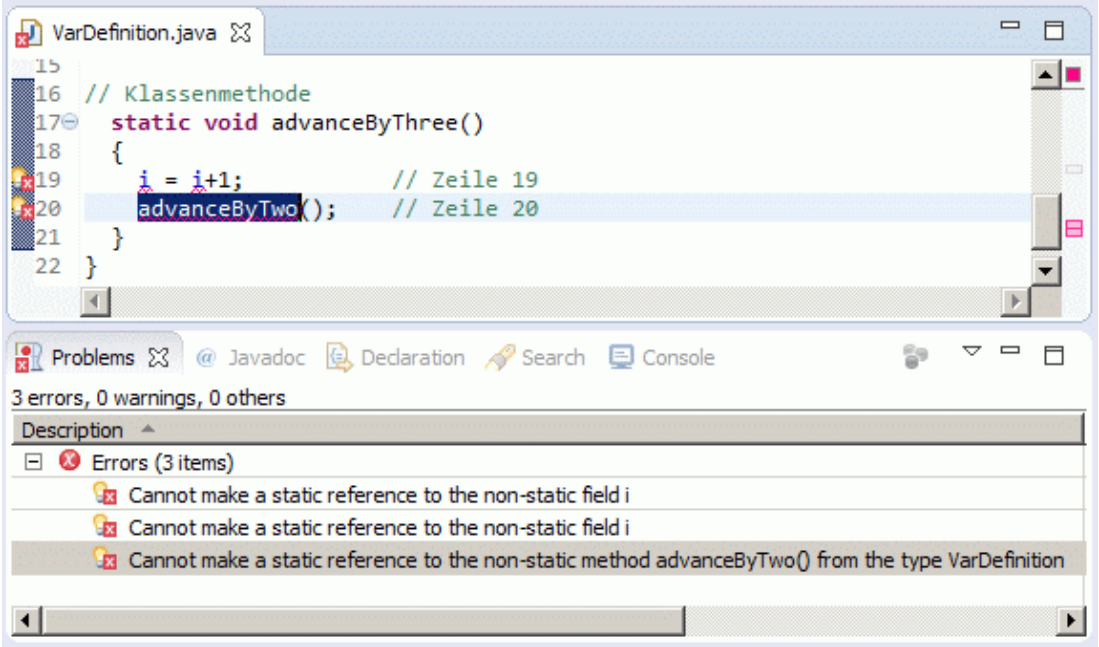
bsp02078.VarDefinition enthält die Deklaration

- einer Instanzvariablen i,
- einer Klassenvariablen j,
- einer Instanzmethode advanceByTwo() und
- einer Klassenmethode advanceByThree().

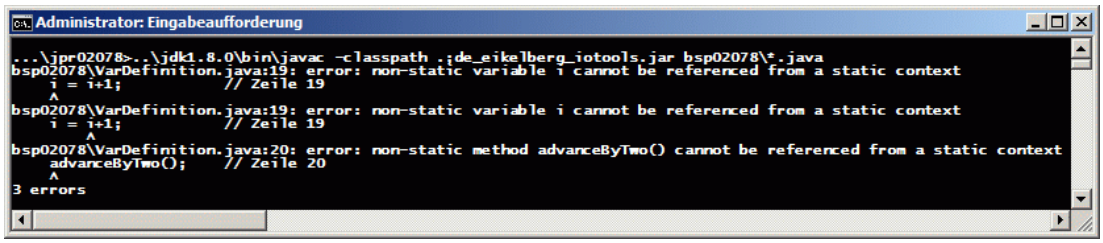
Die Instanzmethode greift (korrekterweise nur) auf die Instanzvariable zu. Die Klassenmethode advanceByThree() verwendet jedoch sowohl eine Instanzvariable als auch auf eine Instanzmethode:

1. In advanceByThree() wird unter Verstoß gegen die Regeln 1 und 2 aus 2.8.4.3 versucht, der Instanzvariablen i einen neuen Wert zuzuweisen.
2. Ferner wird aus der Klassenmethode ein Aufruf der Instanzmethode advanceByTwo() versucht.

Compiliert man die Klasse bsp02078.VarDefinition so erhält man bei Verwendung von Eclipse folgende Fehlermeldungen:



Es wird also in Zeile 19 die zweimalige Verwendung (einmal lesend und einmal schreibend) der (nicht statischen) Instanzvariablen in der (statischen) Klassenmethode als Fehler markiert. In Zeile 20 wird der Aufruf der (nicht statischen) Instanzmethode aus einer (statischen) Klassenmethode heraus als Fehler markiert. Mit anderen Worten wird bsp02078.VarDefinition nicht von Eclipse compiliert. Auch das JDK compiliert diese Klasse nicht, wie der folgende Bildschirmausdruck zeigt:



2.8.4.6 Techniken zur Realisierung eines Zugriffs auf Instanzvariablen und Instanzmethoden aus Klassenmethoden

Wir befassen uns in diesem Kapitel mit Maßnahmen zur Umgehung des in 2.8.4.5 gezeigten Problems. Generell sollte vor Anwendung der Techniken dieses Abschnitts stets untersucht werden, ob nicht eine Design-Änderung eine besser Lösung gibt. Nur wenn es unbedingt notwendig ist, sollten die hier erklärten beiden Möglichkeiten für den Zugriff auf (nicht statische) Instanzvariablen oder Instanzmethoden aus (statischen) Klassenmethoden heraus genutzt werden.

Eine erste Lösung ist die **Instantiierung der Klasse**, die die Instanzvariable bzw. Instanzmethode enthält, **als Objekt**. Auf diese Weise kann auch die Klasse, die die Methode main(String args[]) enthät, instantiiert werden. Hierzu folgendes Beispiel jpr02079.SelbstInstanz:

```
package bsp02079;

import de.eikelberg.iotools.*;

public class SelbstInstanz
{

// Instanzvariable deklarieren
  int zaehler;

  public static void main(String[] args)
  {
// 0. Startmeldung
    ...
// 1. folgender Zugriff auf die Instanzvariable zaehler ergibt
//    eine Fehlermeldung wie in Kapitel 2.8.4.5 beschrieben
//  zaehler = 10;

// 2. Lösung: Instantiierung eines Objektes der Klasse SelbstInstanz
    SelbstInstanz SbIn1;
    SbIn1 = new SelbstInstanz();

    SbIn1.zaehler = SbIn1.zaehler + 1;
```

```
    ScreenOut.kPrintln(
      "Wert von SbIn1.zaehler ist gleich : " + SbIn1.zaehler);
  }
}
```

Man beachte, dass hinter dem fehlerhaften Versuch, im Schritt 1 der Variablen zaehler einen Wert zuzuweisen, streng genommen auch ein Denkfehler steckt. Es wird schließlich versucht, der Variablen zaehler einen Wert zuzuweisen, obwohl sie eine Instanzvariable ist und damit an die Existenz eines Objektes gebunden ist. Im Schritt 2 wird daher korrekterweise eine solche Instanz mit Namen SbIn1 angelegt. Allerdings ist diese Lösung nicht elegant, vermutlich ist in aller Regel eine Designänderung günstiger: Das Anlegen einer zweiten Klasse, die die Instanzvariable enthält, erscheint hier vorteilhafter. Dennoch taucht die obige Konstruktion in einigen Java-Programmen auf.

Diese zu bsp02079.SelbstInstanz angeführten Bemerkungen gelten im Prinzip in gleicher Weise für das folgende Programm bsp02080.InstanzAlsArgument:

```
package bsp02080;

import de.eikelberg.iotools.*;

public class InstanzAlsArgument
{

// Instanzmethode
  void druckeHallo()
  {
    ScreenOut.kPrintln("Instanzmethode ist ausgeführt worden ...");
  }

// Klassenmethode
  static void callInstMethod(InstanzAlsArgument Object1)
  {
    Object1.druckeHallo();
  }

  public static void main(String[] args)
  {
// 0. Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispielprog. Zugriff auf nicht statische Elemente aus Klassenmethoden",
      "M. Eikelberg (www.eikelberg.de)","2000");

// 1. Aufruf der Instanzmethode druckeHallo() ergibt Fehlermeldung
//    wie in Kapitel 2.8.4.5 beschrieben:
//  druckeHallo();

// 2. Erste Lösungsmöglichkeit:
    InstanzAlsArgument InstAArg1;
    InstAArg1 = new InstanzAlsArgument();

    InstAArg1.druckeHallo();

// 3. Zweite Lösungsmöglichkeit unter Verwendung der Klassenmethode
//    callInstMethod() mit übergabe eines Objektes als Argument
    callInstMethod(InstAArg1);
  }
}
```

Der in Schritt 1 versuchte Zugriff führt wieder zu der in 2.8.4.5 beschriebenen Fehlermeldung. Als erste Umgehung im Schritt 2 bietet sich ein Vorgehen wie im Beispielprogramm bsp02079.SelbstInstanz an - dementsprechend gelten alle Anmerkungen zu bsp02079 auch für diesen Schritt.

Die zweite Umgehung verwendet eine zusätzliche statische Methode callInstMethod(InstanzAlsArgument Object1), die ihrerseits ein Objekt der Klasse InstanzAlsArgument als Argument erwartet. In dieser Methode gelingt dann der Übergang vom statischen in den nicht statischen Kontext. Allerdings basiert diese zweite Umgehung wieder auf der (unumgänglichen) Instantiierung eines Objektes. Ferner ist das Einfügen der Methode callInstMethod(InstanzAlsArgument Object1) nicht elegant. Vermutlich lässt sich in aller Regel auch hier wieder durch eine Design-Änderung eine Verbesserung der Programmstruktur mit gleichzeitiger eleganter Umgehung des vorliegenden Problems erreichen.

2.8.5 Konstruktoren

2.8.5.1 Begriffserklärung

In den bisherigen Beispielprogrammen wurden Objekte in der Regel wie folgt instantiiert:

```
  MyClass MyObject1;
  MyObject1 = new Myclass();
```

Auch folgende Form war möglich:

```
  DatasetOut dsOut1;
  dsOut1 = new DatasetOut("daten\\ausgabe.txt");
```

Diese Formen der Instantiierung von Objekten wurde bereits in 2.3.3.2 erklärt. Dort wurde auch als Name für die rechts vom new-Operator zu codierende Methode der Name **Konstruktor** eingeführt.

Generell sind Konstruktoren die Methoden, deren Name gleich dem Namen der Klasse sind, in der sie codiert sind. Jede Klasse hat mindestens einen Konstruktor. Dies wird vom Compiler sichergestellt. Das entsprechende Verhalten der Compiler der verschiedenen JDK-Versionen in Bezug auf das Erzeugen eines parameterlosen Konstruktors ist dabei nicht einheitlich. Für Java 6 gilt:

- Wenn ein Programmierer keinen Konstruktor codiert, wird automatisch ein parameterloser Konstruktor vom Compiler automatisch angelegt.
- Der parameterlose Konstruktor wird jedoch nicht erzeugt, wenn der Programmierer mindestens einen Konstruktor codiert.
- Eine Klasse kann auch mehrere vom Programmierer codierte Konstruktoren enthalten, die verschiedenartige Parameter haben.

Beim Instantiieren eines Objektes werden ein einziges mal die Anweisungen des Konstruktors ausgeführt, der nach dem new-Operator angegeben wurde. Der Name Konstruktor erklärt sich also daraus, dass Konstruktoren beim Anlegen (konstruieren) des Objektes ausgeführt werden. Neben der Ausführung der Anweisungen des Konstruktors erfolgen im übrigen bei der Instantiierung eines Objektes auch systemtechnische Vorgänge wie zum Beispiel die Bereitstellung von Speicherplatz für die Instanzvariablen des Objektes.

In den Anweisungsblöcken der Konstruktoren können also all die Operationen codiert werden, die beim Instantiieren von Objekten durchgeführt werden sollen. Somit können auch die Anweisungen des Initialisierungsblocks für Instanzvariablen in den Konstruktor verlegt werden. Man beachte jedoch, dass sich die Anweisungen des Initialisierungsblocks für *Klassen*variablen nicht in den Konstruktor verlegen lassen.

2.8.5.2 Syntax von Konstruktoren

Man beachte, dass anders als Instanz- und Klassenmethoden Konstruktoren keinen Rückgabewert haben dürfen. Die Deklaration einer Konstruktor-Methode als void führt dazu, dass sie nicht als Konstruktor erkannt wird! Ferner macht die Deklaration eines Konstruktors als private keinen Sinn und führt zu einem Konflikt. In der Regel deklariert man einen Konstruktor vielmehr als public (vgl. 2.8.6).

Im allgemeinen geschieht die Deklaration eines parameterlosen Konstruktors also gemäß folgender Syntax:

```
...
public class Klassenname
{

  public Klassenname()
  {
// Anweisungen des Konstruktors
  }
...
}
```

Parameterlose Konstruktoren *mit leerem Anweisungsblock* brauchen nur dann explizit codiert zu werden, wenn die Klasse über weitere Konstruktoren verfügt.

2.8.5.3 Beispiel - Verwendung eines Konstruktors statt eines Initialisierungsblocks für Instanzvariablen

Im Beispiel bsp02076.VarDefinition wurden Instanzvariablen im Initialisierungsblock für Instanzvariablen initialisiert. Wir wandeln dieses Beispiel ab, ersetzen den Initialisierungsblock für Instanzvariablen durch einen parameterlosen Konstruktor und fügen in diesen Konstruktor zusätzlich noch die Ausgabe einer Meldung ein. Wir erhalten so das folgende Beispiel bsp02081.VarDefinition:

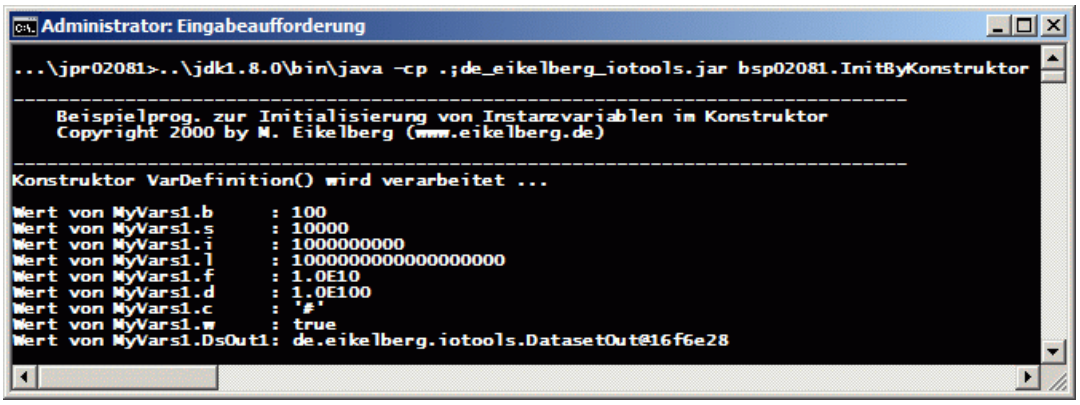
```
package bsp02081;

import de.eikelberg.iotools.*;
```

```
public class VarDefinition
{

// Deklaration der Instanzvariablen
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    char c;
    boolean w;
    DatasetOut dsOut1;

// Initialisierung der Instanzvariablen im Konstruktor
  public VarDefinition()
  {
    ScreenOut.kPrintln("Konstruktor VarDefinition() wird verarbeitet ...");
    ScreenOut.kPrintln();
    b=100;
    s=10000;
    i=1000000000;
    l=1000000000000000000L;
    f=1.0e10f;
    d=1.0e100d;
    c='#';
    w=true;
    dsOut1 = new DatasetOut("daten\\ausgabe.txt");
  }
}
```

Diese Klasse analog zu Beispiel bsp02076 im folgenden Beispielprogramm bsp02081.InitByKonstruktor instantiiert:

```
package bsp02081;

import de.eikelberg.iotools.*;

public class InitByKonstruktor
{

  public static void main(String[] args)
  {
// Startmeldung
    ...
// Objekt der Klasse VarDefinition instantiieren
    VarDefinition MyVars1;
    MyVars1 = new VarDefinition();

// Inhalt der Instanzvariablen von MyVars1 ausgeben
    ScreenOut.kPrintln("Wert von MyVars1.b     : " + MyVars1.b);
    ScreenOut.kPrintln("Wert von MyVars1.s     : " + MyVars1.s);
    ScreenOut.kPrintln("Wert von MyVars1.i     : " + MyVars1.i);
    ScreenOut.kPrintln("Wert von MyVars1.l     : " + MyVars1.l);
    ScreenOut.kPrintln("Wert von MyVars1.f     : " + MyVars1.f);
    ScreenOut.kPrintln("Wert von MyVars1.d     : " + MyVars1.d);
    ScreenOut.kPrintln("Wert von MyVars1.c     : '" + MyVars1.c +"'");
    ScreenOut.kPrintln("Wert von MyVars1.w     : " + MyVars1.w);
    ScreenOut.kPrintln("Wert von MyVars1.dsOut1: " + MyVars1.dsOut1);
  }
}
```

bsp02081.InitByKonstruktor erzeugt folgende Bildschirmausgabe:



2.8.5.4 Beispiele zum Überladen von Konstruktoren

Da es durchaus unterschiedliche Abläufe geben kann, die beim Instantiieren von Objekten erfolgen sollen, gibt es auch die Möglichkeit, zu einem Objekt mehrere Konstruktoren zu codieren. Diese Konstruktoren müssen sich dann durch die Anzahl bzw. Art ihrer Argumente unterschieden (das heisst die Konstruktoren sind "überladen").

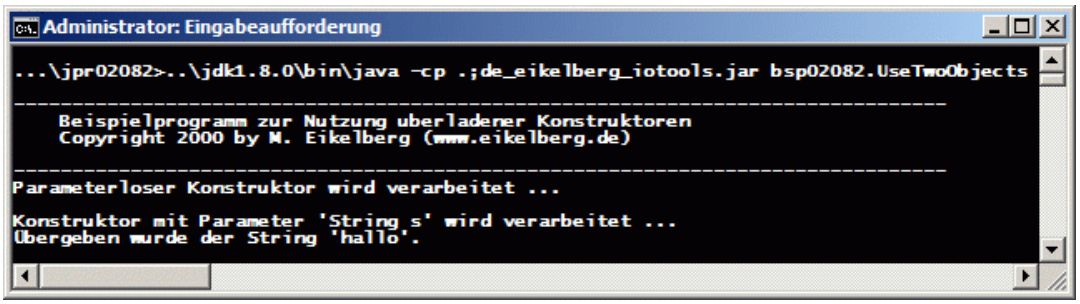
Die Verwendung überladener Konstruktoren geschieht wie im Beispiel bsp02082. In der Klasse bsp02082.MyClass sind sowohl ein parameterloser Konstruktor als auch ein Konstruktor mit Parameter codiert:

```
public class MyClass
{

  public MyClass()
  {
    ScreenOut.kPrintln("Parameterloser Konstruktor wird verarbeitet ...");
    ScreenOut.kPrintln();
  }

  public MyClass(String s)
  {
    ScreenOut.kPrintln("Konstruktor mit Parameter 'String s' wird verarbeitet ...");
    ScreenOut.kPrintln("Übergeben wurde der String '"+s+"'.");
    ScreenOut.kPrintln();
  }
}
```

Dies beiden Konstruktoren der Klasse MyClass werden im Programm bsp02082.UseTwoObjects wie folgt verwendet:

```
package bsp02082;
...
public class UseTwoObjects
{
  @SuppressWarnings("unused")  // Unterdrücken der Warnungen wg. unbenutzter lokaler Variablen
  public static void main(String[] args)
  {
...
// 1. Erstes Objekt anlegen mit parameterlosem Konstruktor
    MyClass MyObject1;
    MyObject1 = new MyClass();

// 2. Zweites Objekt anlegen mit Übergabe einer Zeichenkette
    MyClass MyObject2;
    MyObject2 = new MyClass("hallo");
...
  }
}
```

Der Verwendung der beiden Konstruktoren führt zu folgender Bildschirmausgabe:

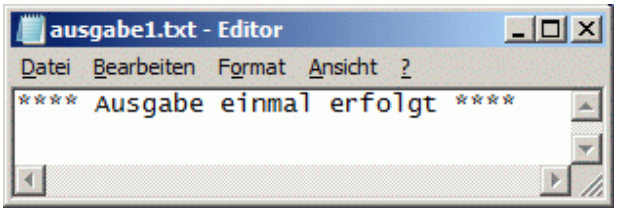


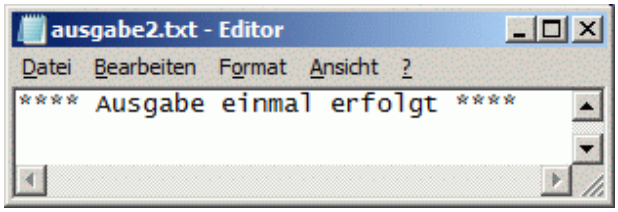
Eine sinnvolle Anwendung überladener Konstruktoren findet sich in der Klasse DatasetOut. Die Aufgabe der Konstruktoren der Klasse DatatsetOut besteht in dem Öffnen der Ausgabedateien. Dabei gibt es prinzipiell die Möglichkeit, vorhandene Dateien gleichen Namens zu überschreiben oder aber die zu schreibenden Daten an die Daten in vorhandenen Dateien anzufügen. Beide Möglichkeiten werden in folgendem Beispielprogramm bsp02083.DoppelteAusgabe verwendet:

```
package bsp02083;

import de.eikelberg.iotools.*;

public class DoppelteAusgabe
{

  public static void main(String[] args)
  {
// 0. Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispiel zur Verwendung von überladenen Konstruktoren",
      "M. Eikelberg (www.eikelberg.de)","2000");

// 1. Erste Ausgabedatei öffnen mit Überschreiben vorhandener Dateien
    DatasetOut dsOut1;
    dsOut1 = new DatasetOut("daten\\ausgabe1.txt");

// 2. In Datei schreiben und Datei wieder schließen
    dsOut1.dPrintln("**** Ausgabe einmal erfolgt ****");
    dsOut1.dClose();

// 3. Zweite Ausgabedatei öffnen mit Anhängen an vorhandene Dateien
    DatasetOut dsOut2;
    dsOut2 = new DatasetOut("daten\\ausgabe2.txt",true);

// 4. In Datei schreiben und Datei wieder schließen
    dsOut2.dPrintln("**** Ausgabe einmal erfolgt ****");
    dsOut2.dClose();
  }
}
```

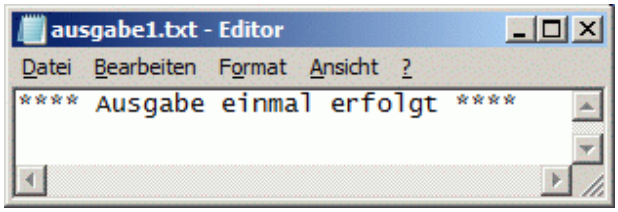
Im Verarbeitungsschritt 1 wird zunächst der Konstruktor DatasetOut(String) verwendet, um eine Ausgabedatei zu öffnen und dabei eine gegebenenfalls vorhandene gleichnamige Datei zu überschreiben. Der in Verarbeitungsschritt 3 dagegen verwendete Konstruktor DatasetOut(String, boolean) lässt die Möglichkeit zu, durch Einsetzen von true als zweitem Parameter ein Anfügen an die Daten einer gegebenenfalls vorhandenen gleichnamigen Datei zu erreichen. Im Beispielprogramm bsp02083.DoppelteAusgabe wirkt sich dieser Unterschied wie folgt aus:

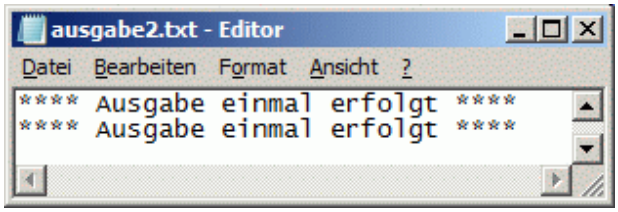
- Nach einmaliger Abarbeitung des Programms ist der Inhalt der beiden Ausgabedateien identisch:





- Nach einem zweiten Programmlauf ist der Inhalt der beiden Ausgabedateien verschieden, da bei der Ausgabe des zweiten Programmlaufs die Datei ausgabe1.txt überschrieben wird, in der Datei ausgabe2.txt der neu geschriebene Text jedoch an den vorhandenen Text angehängt wird:





2.8.5.5 Datenübergabe an Objekte über den Konstruktor

Zu den Standard-Techniken bei der objektorientierten Programmierung

- gehört die Datenübergabe an Objekte über den Konstruktor.

Hierdurch wird sichergestellt,

- dass ein Objekt beim Anlegen stets mit den erforderlichen Daten gefüllt wird.

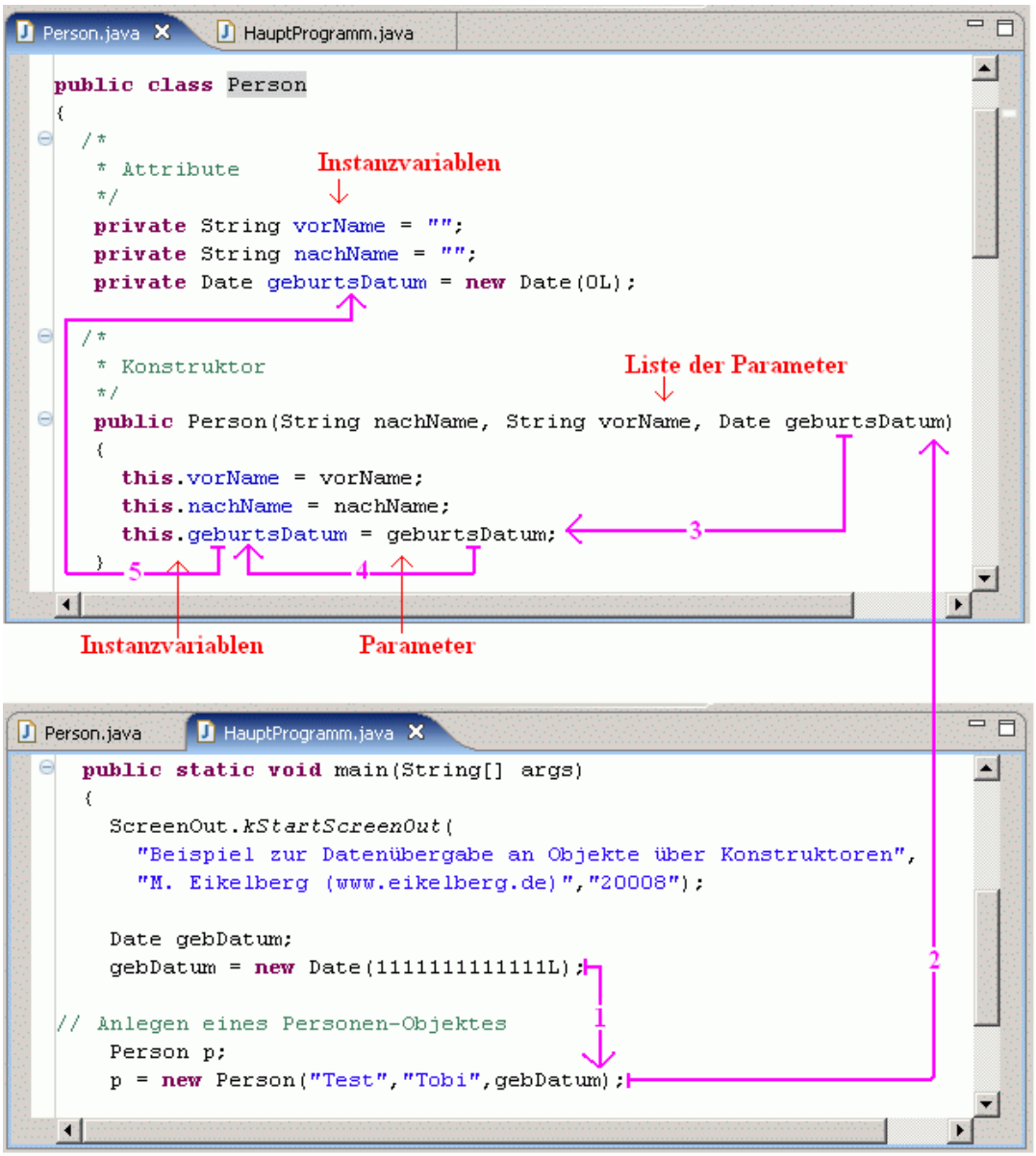
Da kein parameterloser Konstruktor codiert wird und da ein parameterbehafteter Konstruktor existiert (weshalb kein parameterloser Konstruktor vom Compiler erzeugt wird), hat die Klasse entsprechend keinen parameterlosen Konstruktor:

```
package bsp02092;

import java.util.Date;

public class Person
{
  /*
   * Attribute
   */
   private String vorName = "";
   private String nachName = "";
   private Date geburtsDatum = new Date(0L);

  /*
   * Konstruktoren
   */
   public Person(String nachName, String vorName, Date geburtsDatum)
   {
     this.vorName = vorName;
     this.nachName = nachName;
     this.geburtsDatum = geburtsDatum;
   }

...
}
```

In obigem Programmcode sind folgende wichtigen Schritte enthalten:

- Für die Daten des Objektes werden Instanzvariablen (Attribute) angelegt. Diese erhalten - wie allgemein für die objektorientierte Programmierung empfohlen - die Sichtbarkeit private.
- Der parameter-behaftete Konstruktor `Person(String nachName, String vorName, Date geburtsDatum)` hat eine Liste von Parametern, die gleichnamig mit Instanzvariablen der Klasse sind. Diese Parameter überdecken im Methodenrumpf des Konstruktors zunächst die gleichnamigen Instanzvariablen, auf die daher beim Umspeichern der Daten mit dem Zeiger this zugegriffen wird:

Die Datenübergabe an ein Person-Objekt geschieht wie in folgender Grafik aufgezeigt:



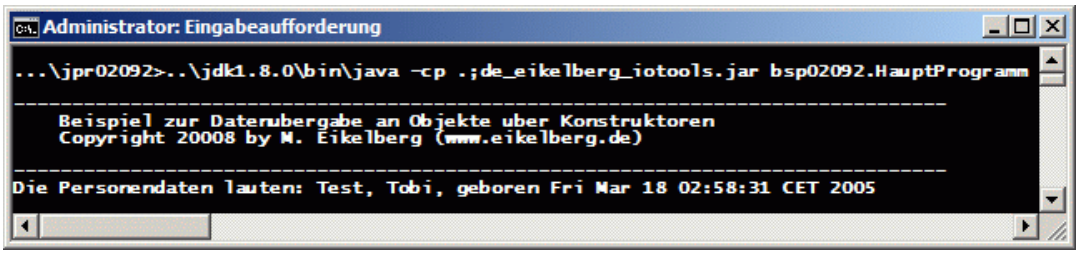
Die in der Grafik gezeigten Schritte haben folgende Bedeutung:

1. Die Variable `geburtsDatum` des Hauptprogramms enthält die Adresse eines Objektes der Klasse java.util.Date. Diese Adresse wird beim Methodenaufruf in einen Parameter eines Konstruktors der Klasse Person eingetragen.
2. Beim Aufruf des Konstruktors enthält der Parameter die Adresse des Date-Objektes.
3. Im Methodenrumpf des Konstruktors kann der Parameter (fast) wie eine lokale Variable verwendet werden. Insbesondere überdeckt der Parameter die gleichnamige Instanzvariable des Person-Objektes.
4. Auf die Instanzvariable des Person-Objektes kann mit dem Zeiger this zugegriffen werden, wenn die Variable durch eine lokale Variable oder einen Parameter überdeckt wird. Hier wird die Adresse des Date-Objektes vom Parameter in die gleichnamige Instanzvariable umkopiert.
5. Der Zeiger this zeigt stets auf Instanzvariablen. Damit ist die Adresse des Date-Objektes jetzt in der Instanzvariablen `geburtsDatum` gespeichert und kann in anderen Methoden der Klasse Person verwendet werden.

Im obigen Programmcode werden somit *drei gleichnamige Speicherplätze* `geburtsDatum` verwendet:

- die lokale Variable `geburtsDatum` der main-Methode
- der Parameter `geburtsDatum` des Konstruktors Person(String nachName, String vorName, Date geburtsDatum)
- die Instanzvariable `geburtsDatum` der Klasse Person

Bei der weiteren Verarbeitung kann auf die Instanzvariable `geburtsDatum` der Klasse Person von einer anderen Methode aus zugegriffen werden. Dies zeigt der Bildschirmausdruck eines Programmlaufs:



Die vom Programm durchgeführte Ausgabe erfolgt aufgrund des folgenden Methodenaufrufes in der main-Methode:

```
ScreenOut.kPrintln("Die Personendaten lauten: "+p.getPersonendaten());
```

Der Methodenaufruf ist in der Klasse Person wie folgt codiert worden:

```
public String getPersonendaten()
{
  String pDat;
  pDat = this.nachName + ", " + this.vorName +
      ", geboren " + this.geburtsDatum;
  return pDat;
}
```

In der obigen Methode wurde somit erfolgreich auf die Adresse des Date-Objektes, gespeichert in der Instanzvariablen `geburtsDatum` der Klasse Person zugegriffen.

2.8.6 Get- und Set-Methoden

2.8.6.1 Schutz der Daten durch Kapselung und Information Hiding

Es ist ein wichtiges Prinzip des Software Engineering, möglichst unabhängige und robuste Software-Bausteine zu erstellen, die auch von außen in ihrer Funktion nur schwer zu stören sind.

Jeder Programmierer mit einer etwas längeren Programmiererfahrung auch in älteren Programmiersprachen weiß, dass eine Ursache für schwer zu analysierende Programmierfehler mit unter Umständen schwerwiegenden Folgen der direkte Zugriff 'von außen' auf Variablen von Prozeduren, Methoden oder Klassen ist. Ein solcher Zugriff erfolgt an der Logik und Struktur des Programmteils vorbei, das die Variable enthält. Solch ein 'schlauer' Direktzugriff ist oft im ersten Moment sehr bequem, kann jedoch später schwer zu untersuchende Laufzeitfehler hervorrufen.

Zur Vermeidung dieser Situation ist in der Informatik das Prinzip der **Kapselung** definiert worden. Dieses Prinzip beinhaltet, dass man Variablen und Methoden als zusammengehörig betrachtet und entsprechend gemeinsam in Klassen bzw. Objekten anordnet. Die Variablen sollten dann nur noch von Methoden der gleichen Klasse verwendet werden. Auf diese Weise wird dann möglich, dass man die Daten in den Variablen davor schützt, dass nicht in der Klasse oder im Objekt vorhandene Methoden auf sie zugreifen und sie möglicherweise verändern. Dieses Vorgehensweise entspricht dem Prinzip des **Information Hiding**.

2.8.6.2 Get- und Set-Methoden

Der Erfahrungsschatz der Informatiker besagt also, dass es die Qualität von Software verbessert, wenn die Daten in den Variablen von Klassen oder Objekten nur von Methoden der gleichen Klasse bzw. des gleichen Objektes bearbeitet werden. Dies kann man wie folgt erreichen:

1. Variablen sollten stets als private deklariert werden.
2. Falls ein lesender Zugriff auf Daten einer Variablen ausserhalb der Klasse oder des Objektes erforderlich ist, sollte in der Klasse eine - protected oder public - Methode codiert werden, die die Daten herausgibt. Eine solche lesende Methode heisst **Get-Methode**.
3. Falls ein schreibender Zugriff auf Daten einer Variablen ausserhalb der Klasse oder des Objektes erforderlich ist, sollte in der Klasse eine - protected oder public - Methode codiert werden, die die Daten in der Klasse oder im Objekt verändert. Eine solche schreibende Methode heisst **Set-Methode**.
4. Für Daten, die ausserhalb der Klasse bzw. des Objektes nicht bearbeitet werden sollen, sollten auch keine Get- und Set-Methoden codiert werden. Diese Daten sind damit geschützt.

Die genannten Modifizierer wurden bereits in 2.3.3.1 eingeführt und werden in 2.8.7 erneut angesprochen.

Der wichtigste Vorteil von Get- und Set-Methoden ist der bessere Schutz von Daten:

- Man kann den lesenden und schreibenden Zugriff unterschiedlich zulassen, zum Beispiel public lesenden Zugriff und protected schreibenden Zugriff oder aber überhaupt keinen schreibenden Zugriff.
- In die Set-Methode kann man Verarbeitungsschritte integrieren, innerhalb derer die Daten daraufhin überprüft werden, ob sie sachlich oder technisch zulässig sind. Auf diese Weise verhindert man die Übergabe von ungültigen Daten an ein Objekt.

Man beachte, dass die Wirksamkeit von Get- und Set-Methoden nur dann gegeben ist, wenn gleichzeitig mit Hilfe der Modifizierer eine geeignete Sichtbarkeit der Daten festgelegt wird. Man beachte daher die erneute Behandlung der Modifizierer in Kapitel 2.8.7.

Ein einfaches Beispiel für eine Get- und eine Set-Methode ist in der Klasse bsp02084.HiddenData enthalten:

```
package bsp02084;

public class HiddenData
{
// Instanzvariable
  private int zaehler;

// Set-Methode
  protected void setZaehler(int zaehler)
  {
    this.zaehler = zaehler;
  }

// Get-Methode
  public int getZaehler()
  {
    return this.zaehler;
  }

// Methode zur Verarbeitung
  public void advanceZaehler()
  {
```

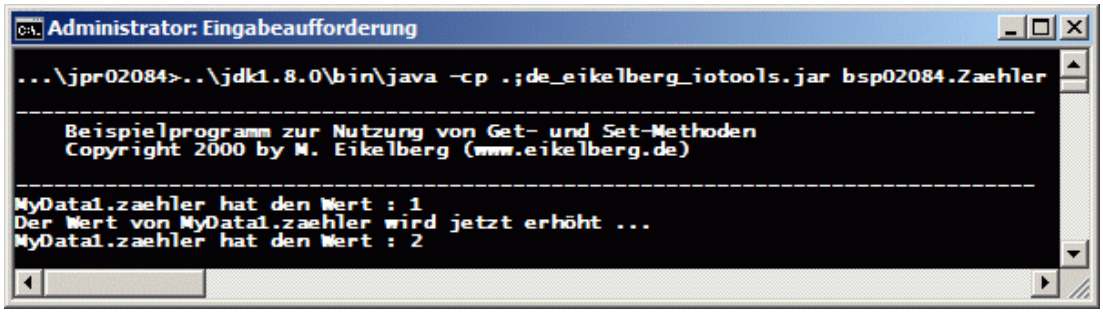
```
    zaehler = zaehler + 1;
  }
}
```

In der Set-Methode setZaehler(int) wurde bei der Implementierung der Zeiger this verwendet. Dies ist an dieser Stelle notwendig, da die Instanzvariable zaehler durch eine gleichnamige lokale Variable verdeckt wird. Aber auch wenn eine Verwendung von this nicht erforderlich ist, wird in Get- und Set-Methoden oft der Zeiger this zur Verdeutlichung der Tatsache codiert, dass ein Zugriff auf eine Instanz- oder Klassenvariable erfolgt. Daher ist in bsp02084.HiddenData in der Get-Methode getZaehler() ebenfalls der Zeiger this codiert worden.

In bsp02084.Zaehler wird die Instanzvariablen zaehler der Klasse bsp02084.HiddenData wie folgt verarbeitet:

```
package bsp02084;

import de.eikelberg.iotools.*;

public class Zaehler
{

  public static void main(String[] args)
  {
// Startmeldung
...
// Objekt der Klasse HiddenData instantiieren
    HiddenData MyData1;
    MyData1 = new HiddenData();

// Wert in MyData1 setzen und auslesen
    MyData1.setZaehler(1);
    ScreenOut.kPrintln(
      "MyData1.zaehler hat den Wert : " + MyData1.getZaehler());

// Wert in MyData1 erhöhen und wieder auslesen
    ScreenOut.kPrintln("Der Wert von MyData1.zaehler wird jetzt erhöht ...");
    MyData1.advanceZaehler();
    ScreenOut.kPrintln(
      "MyData1.zaehler hat den Wert : " + MyData1.getZaehler());
  }
}
```

Es erfolgt also kein direkter Zugriff mehr mit MyData1.zaehler auf die Instanzvariable. Diese Zugriffe wurden vielmehr ersetzt durch die Methodenaufrufe MyData1.setZaehler(1) und MyData1.getZaehler(). Die Wirkung des Programms bsp02084.Zaehler zeigt der folgende Bildschirmausdruck:



Mit der Methode MyData1.setZaehler(1) wurde also zunächst der Instanzvariablen zaehler der Wert 1 zugewiesen. Mit MyData1.getZaehler() wurde dieser Wert wieder ausgelesen. Mit MyData1.advanceZaehler() wurde der Wert der Variablen zaehler anschließend um 1 erhöht. Zum Abschluss wurde dann der neue Wert 2 mit der Methode MyData1.getZaehler() wieder ausgelesen.

2.8.6.3 Beispielprogramm zur Verwendung von Get- und Set-Methoden.

Zur Verdeutlichung folgt ein weiteres Beispielprogramm zur sinnvollen Verwendung von Get- und Set-Methoden. In diesem Beispiel *fehlen* Set-Methoden, so dass verhindert wird, dass die Daten des Objektes von außen manipuliert werden. Hierzu ist es zusätzlich erforderlich, die Variablen des Objektes als private zu deklarieren und eine Get-Methode zum Auslesen der Daten aus dem Objekt zu codieren.

Inhaltlich realisiert das folgende Beispiel bsp02085.StoppUhr die Funktion einer Stoppuhr. Es besteht die Möglichkeit, mit der Methode setTimeStamp() einen Zeitpunkt zu markieren. Die Methode getInterval() liefert die zwischen den beiden letzten Zeitpunkten vergangene Zeit in Millisekunden. Damit die Verarbeitung stets sinnvoll erfolgt, wird bei der Instantiierung vom Konstruktor ein erster Zeitpunkt markiert. bsp02085.StoppUhr besteht insgesamt aus folgendem Java Quellcode:

```
package bsp02085;
```

```
public class StoppUhr
{
// Instanzvariable für die Zeitpunkte
  private long startTime, endTime;

// Instanzvariable für die Länge des Zeitintervalls
  private long interval;

// Konstruktor sichert die korrekte Verarbeitung durch
// Setzen eines ersten Zeitpunktes
  public StoppUhr()
  {
    setTimeStamp();
  }

// Get-Methode für die Länge des Zeitintervalls
  public long getInterval()
  {
    return this.interval;
  }

// Methode zum Setzen eines Zeitpunktes
  public void setTimeStamp()
  {
    startTime = endTime;
// Zeitabfrage bei Betriebssystem mit System.currentTimeMillis()
    endTime = System.currentTimeMillis();
    calcInterval();
  }

// Methode zur Berechnung der Länge des Zeitintervalls
  private void calcInterval()
  {
    this.interval = endTime - startTime;
  }
}
```

Verwendet wird die Klasse bsp02085.StoppUhr im Programm bsp02085.Sinusmessung für die Messung der Zeit, die für die Berechnung von 10.000.000 Sinus-Werten benötigt wird. Der Programmcode von bsp02085.StoppUhr lautet wie folgt:

```
package bsp02085;

import de.eikelberg.iotools.*;

public class Sinusmessung
{

  public static void main(String[] args)
  {
// 0. Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispielprogramm Zeitmessung zu Kapselung und Information Hiding",
      "M. Eikelberg (www.eikelberg.de)","2000");

// 1. Objekt der Klasse StoppUhr instantiieren
    StoppUhr Uhr1;
    Uhr1 = new StoppUhr();

// 2. Dauer der Verarbeitung im Objekt Uhr1 messen:
//    Zweimaliges Setzen von Zeitpunkten unmittelbar hintereinander
//    ergibt bei anschließender Ermittlung der zwischen den
//    beiden Messzeitpunkten vergangenen Zeit die Zeit, die die
//    Stoppuhr für die eigene Verarbeitung benötigt.
    Uhr1.setTimeStamp();
    Uhr1.setTimeStamp();
    long messZeit = Uhr1.getInterval();

// 3. ersten Zeitpunkt setzen
    Uhr1.setTimeStamp();

// 4. zu messende Verarbeitung durchführen -
```

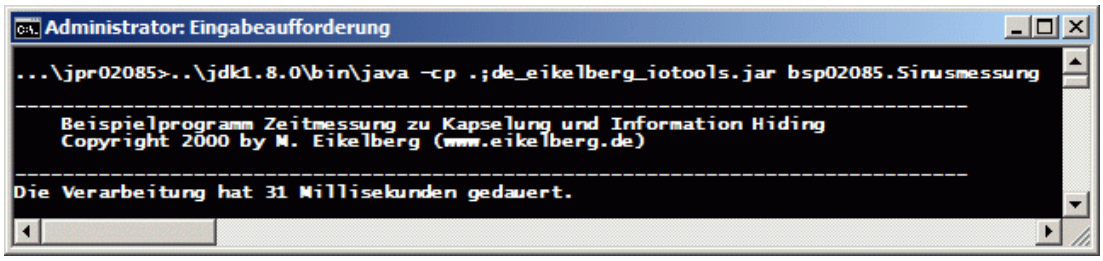
```
//    10.000.000 mal einen Sinus berechnen
    for (long i=0; i<10000000; i++)
      Math.sin((double)i);

// 5. zweiten Zeitpunkt setzen
    Uhr1.setTimeStamp();

// 6. Länge der Verarbeitung berechnen und ausgeben
    long verarbZeit;
    verarbZeit = Uhr1.getInterval() - messZeit;
    ScreenOut.kPrintln(
      "Die Verarbeitung hat " + verarbZeit + " Millisekunden gedauert.");
  }
}
```

Der Programmcode zeigt, dass mit einem Objekt der Klasse bsp02085.StoppUhr wie mit einer Stoppuhr gearbeitet werden kann. Die interne Funktion der Klasse bsp02085.StoppUhr interessiert im Programm bsp02085.Sinusmessung nicht mehr. Andererseits kann die interne Funktion der Klasse bsp02085.StoppUhr wie bei einer richtigen Stoppuhr nicht von außen beeinflußt werden. Die Verarbeitung im Programm bsp02085.Sinusmessung geschieht wie folgt:

1. Es wird ein Objekt der Klasse StoppUhr intantiiert, man erhält also funktional eine Stoppuhr.
2. Es erfolgt ein zweimaliger Aufruf der Methode setTimeStamp(), das heißt die Stoppuhr wird zweimal betätigt. Anschließend wird mit der Methode getInterval() die gemessene Zeit für das zweimalige Betätigen der Stoppuhr ausgelesen.
3. Die eigentliche Messung wird mit der Methode setTimeStamp() begonnen.
4. Die zu messende Verarbeitung wird durchgeführt.
5. Die eigentliche Messung wird mit der Methode setTimeStamp() beendet.
6. Die gemessene Zeit wird aus dem Objekt Uhr1 ausgelesen und um im Schritt 2 gemessene Zeit verringert. Anschließend wird diese verringerte Zeit auf dem Bildschirm ausgegeben.

Das Beispiel bsp02085 zeigt neben der sinnvollen Umsetzung der Prinzipien von Kapselung und Information Hiding auch, dass sinnvolle Klassen und Objekte durch Modellierung von Dingen der realen Welt entwickelt werden können.

Zum Abschluss dieses Kapitels folgt ein Bildschirmausdruck des Programms bsp02085.Sinusmessung, erzeugt auf einem Rechner unter Windows 7 professinal 64 bit mit Intel Core 2 Duo 2,33 GHz Prozessoren und 8 GByte Speicher:



2.8.7 Modifizierer zur Sichtbarkeit

2.8.7.1 Überblick über die wichtigsten Modifizerer

Vorbemerkung: In diesem Kapitel wird der Klarheit und Vollständigkeit halber gelegentlich auch von sogenannten **Sohn-Klassen** gesprochen. Sohn-Klassen **erben** von Ihren sogenannten **Vater-Klassen** Variablen und Methoden, das heisst in den Sohn-Klassen sind die in Vater-Klassen codierten Variablen und Methoden automatisch ebenfalls vorhanden. Das diesem Vorgang zugrundeliegende Prinzip der **Vererbung** ist ein wichtiges Prinzip der (vollen) objektorientierten Programmierung (vgl. 2.8.1), das bisher nicht besprochen wurde. Daher können die entsprechenden Bemerkungen zu Sohn-Klassen beim Durcharbeiten auch ignoriert werden.

Modifizierer regeln wie bereits in 2.3.3.1 angesprochen die *Sichtbarkeit, Lebensdauer* und *Veränderbarkeit* der Variablen bzw. der Methode.

- Eine Klasse, Variable oder Methode heisst **sichtbar**, wenn sie in einer anderen Klasse verwendet werden kann.
- Die **Lebensdauer** ist der Zeitraum, in dem die Variable verfügbar ist. Instanzvariablen sind an die Existenz eines Objektes gebunden, Klassenvariablen sind dagegen unabhängig von der Existenz von Objekten der Klasse verfügbar.
- Eine Variable oder Methode ist **veränderbar**, wenn sie nach der ersten Wertzuweisung oder Deklaration noch einmal verändert werden kann. In Bezug auf Variablen ist dabei der Begriff der Veränderbarkeit verständlich. In Bezug auf Methoden wird dieser Begriff erst verständlich, wenn das Prinzip des **Überschreibens** erklärt ist. Das Prinzip des **Überschreibens** ist ein weiteres wichtiges Prinzip der (vollen) objektorientierten Programmierung, das wie das Prinzip der Vererbung bisher nicht erklärt wurde.

Die folgende Tabelle stellt die im aktuellen Zusammenhang wichtigen Modifizierer zusammen. Die Sprachdefinition von Java kennt noch einige weitere Modifizierer, die zum Beispiel im Zusammenhang mit der sogenannten Nebenläufigkeit (Threads) interessant sind.

Zweck	Modifizierer
Sichtbarkeit	private, protected, public
Lebensdauer	static
Veränderbarkeit	final

Es ist durchaus unterschiedlich, wann und wo welche Modifizier erforderlich, erlaubt oder nicht erlaubt sind. Die folgende Tabelle zeigt, welche Modifizierer zur Sichtbarkeit und Lebensdauer an welcher Stelle bzw. zu welchem Schlüsselwort im Programmcode einer Klasse verlangt (!) werden bzw. erlaubt (+) bzw. nicht erlaubt (-) sind:

Ort / Schlüsselwort	private	protected	public	static
class	-	-	+	-
main	-	-	!	!
Deklaration Konstruktor	+	+	+	-
Deklaration Klassenvariable	+	+	+	!
Deklaration Instanzvariable	+	+	+	-
Deklaration Klassenmethode	+	+	+	!
Deklaration Instanzmethode	+	+	+	-

Wichtig ist, dass nur eine als public und static gekennzeichnete Methode main als Haupt-Methode eines Java-Programms verwendet wird. Man beachte ferner, dass das Weglassen eines Modifizierers zur Sichtbarkeit dazu führt, dass standardmäßig folgende Sichtbarkeit gesetzt wird:

Ort / Schlüsselwort	Standard-Sichtbarkeit bei Weglassen der Modifizierer (Sichbarkeit default)
class	Sichtbarkeit der Klasse nur aus Klassen des gleichen Paketes
main	Fehlt der Modifizierer public, so ist die Methode main(String[] args) nicht mehr Haupt-Methode und damit nicht mehr Startpunkt eines Hauptprogramms.
Deklaration Konstruktor	Klassen aus dem gleichen Paket (und alle sogenannten Sohn-Klassen, die von solchen Klassen erben) erhalten Zugriff, andere Klassen erhalten keinen Zugriff
Deklaration Klassenvariable	
Deklaration Instanzvariable	
Deklaration Klassenmethode	
Deklaration Instanzmethode	

Die Sichtbarkeit default wird also nicht durch einen Modifizierer festgelegt, sondern durch das *Weglassen* des Modifizierers!

2.8.7.2 Wirkung der Modifizierer auf Klassen

Wie die obige Tabelle zeigt, gibt es zu Klassen nur zwei Möglichkeiten für die Festsetzung des Zugriffsschutzes:

1. Es wird der Modifizierer **public** codiert. In diesem Fall ist die Klasse in allen anderen Klassen sichtbar. Soll die Klasse in einer Klasse eines anderen Paketes verwendet werden, muss sie in diesem Fall zusätzlich mit der **import**-Anweisung importiert werden.
2. Wird kein Modifizierer codiert, so gilt Sichtbarkeit default. Die Klasse ist dann lediglich in Klassen des gleichen Paketes sichtbar. Da Klassen des gleichen Paketes nicht importiert werden brauchen, ist hier auch keine import-Anweisung erforderlich.

Die Bedeutung der Gruppierung der Klassen in Pakete, die Anwendung der import-Anweisung und die unterschiedlich einstellbare Sichtbarkeit von Klassen zeigt das Beispiel 88, das aus allen Klassen der Pakete bsp02088 und bsp02088a besteht. Die Methode main(String[] args) ist in der Klasse bsp02088.HauptProgramm enthalten. In der Klasse bsp02088.HauptProgramm wird jeweils ein Objekt der Klassen bsp02088.Class88 und bsp02088a.Class88a instantiiert.

Die Klasse bsp02088.Class88 besteht aus folgendem Java Quellcode:

```
package bsp02088;                  // (1)

import de.eikelberg.iotools.*;

class Class88                      // (2)
{

  protected Class88()              // (3)
  {
    ScreenOut.kPrintln(
      "Es wird ein Objekt der Klasse Class88 aus Paket bsp02088 instantiiert...");
    ScreenOut.kPrintln();
  }
}
```

Folgende Besonderheiten sind hier zu beobachten:

1. Die Klasse Class88 gehört in das gleiche Paket wie die Klasse bsp02088.HauptProgramm.
2. Da Class88 und HauptProgramm zum gleichen Paket gehören, kann die Sichtbarkeit von public auf default eingeschränkt werden.
3. Da Class88 nur default Sichtbarkeit hat, macht ein public Konstruktor keinen Sinn. Vielmehr kann die Sichtbarkeit auch auf protected eingeschränkt werden. Auch das Weglassen des Modifizieres würde Sinn machen - es läge dann auch für den Konstruktor default Zugriffsschutz vor (vgl. 2.8.7.4).

Die zweite in bsp02088.HauptProgramm verwendete Klasse ist die Klasse bsp02088a.Class88a. Diese Klasse besteht aus folgendem Quellcode:

```
package bsp02088a;                 // (1)

import de.eikelberg.iotools.*;

public class Class88a              // (2)
{

  public Class88a()                // (3)
  {
    ScreenOut.kPrintln(
      "Es wird ein Objekt der Klasse Class88a aus Paket bsp02088a instantiiert...");
    ScreenOut.kPrintln();
  }
}
```

Die Klasse Class88a unterscheidet sich in folgenden Punkten von der Klasse Class88:

1. Die Klasse Class88a nicht Bestandteil des Paketes bsp02088, sondern gehört zum Pakete bsp02088a.
2. Wenn die Klasse bsp02088a.Class88a in bsp02088.HauptProgramm verwendbar sein soll, muss sie public sein.
3. Wenn in bsp02088.HauptProgramm ein Objekt der Klasse bsp02088a.Class88a angelegt werden soll, muss der Konstruktor von Class88a ebenfalls public sein.

In bsp02088.HauptProgramm werden zu beiden Klassen bsp02088.Class88 und bsp02088a.Class88a Objekte angelegt. Dabei sind jedoch wegen der unterschiedlichen Pakete verschiedene Maßnahmen erforderlich, die im Quellcode zu bsp02088.HauptProgramm sichtbar werden:

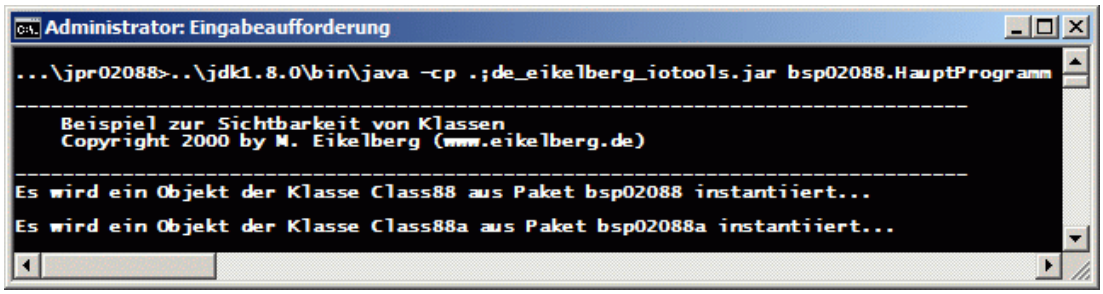
```
package bsp02088;                  // (1)

import de.eikelberg.iotools.*;
import bsp02088a.*;                // (2)
                                   // (3)

public class HauptProgramm
{
  @SuppressWarnings("unused")  // Unterdrücken der Warnung wg. unbenutzter Variablen
```

```
  public static void main(String[] args)
  {
// Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispiel zur Sichtbarkeit von Klassen",
      "M. Eikelberg (www.eikelberg.de)","2000");

// Objekt der Klasse Class88, Paket bsp02088 instantiieren
    Class88 Obj88;
    Obj88 = new Class88();

// Objekt der Klasse Class88, Paket bsp02088 instantiieren
    Class88a Obj88a;
    Obj88a = new Class88a();
  }
}
```

Die Unterschiede bei der Verwendung der Klassen Class88 und Class88a sind sämtlich ausserhalb der Methode main zu finden:

1. HauptProgramm gehört zum Paket bsp02088.
2. Klassen des 'fremden' Paketes bsp02088a müssen mit einer import-Anweisung importiert werden, ansonsten sind sie überhaupt nicht sichtbar.
3. Klassen des 'eigenen' Paketes brauchen nicht importiert zu werden. Daher fehlt hier eine import-Anweisung zu den Klassen des Paketes bsp02088.

Nachdem die Klassen des 'fremden' Paketes importiert sind, können sie verwendet werden, sofern sie - wie die Klasse Class88a - public sichtbar sind. Entsprechend werden in bsp02088.HauptProgramm auch korrekt Objekte der Klassen Class88 und Class88a angelegt, wie der folgende ScreenShot zeigt:

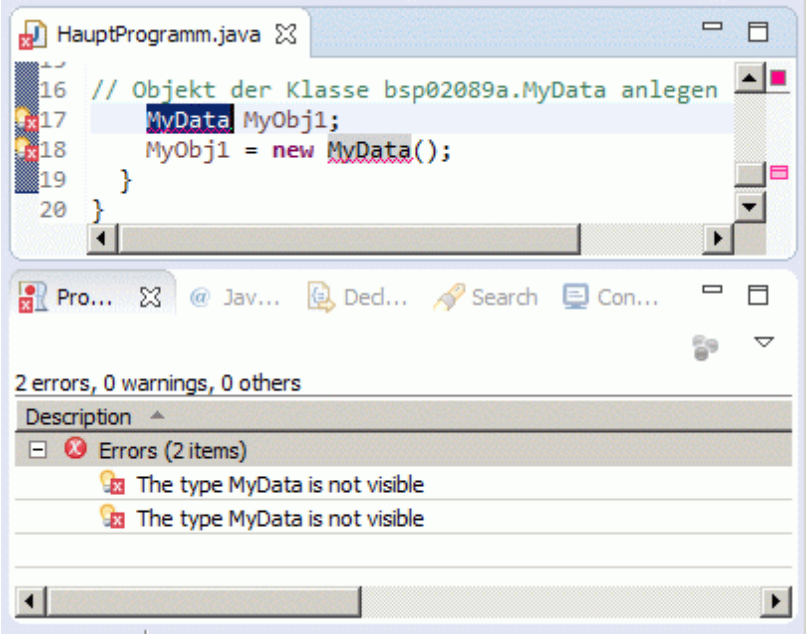


2.8.7.3 Wirkung der Modifizierer auf Variablen und Methoden

Die folgende Tabelle stellt zusammen, welche Wirkung die Modifizierer zur Sichtbarkeit auf Variablen und Methoden haben.

Attribut	Wirkung
private	Die Variable bzw. die Methode ist nur in der *Klasse* verwendbar, in der sie deklariert ist.
protected	Die Variable bzw. die Methode ist in den Klassen des *Paketes* sichtbar, in der sie deklariert ist. Ferner ist die Variable bzw. die Methode in allen Sohn-Klassen dieser Klassen sichtbar.
public	Die Variable bzw. die Methode ist überall dort verwendbar, wo die umgebende Klasse importiert worden ist.
kein Attribut / default Sichtbarkeit	Die Variable bzw. die Methode ist in allen Klassen des Paketes sichtbar, in der sie deklariert ist.

Vorausgesetzt wird dabei, dass die Sichtbarkeit der ganzen Klasse nicht bereits durch einen Modifizierer stärker eingeengt wurde als das jeweilige Attribut zur Variablen oder Methode besagt. Man beachte also, dass eine entsprechende Sichtbarkeit der Variablen und Methoden nur dann errreicht wird, wenn die umgebende Klasse mindestens die gleiche Sichtbarkeit hat. Ist also wie im Beispiel bsp02089.Hauptprogramm und bsp02089a.MyData die Sichtbarkeit der Klasse MyData gleich default, so werden public Instanzvariablen in MyData schon deshalb nicht sichtbar, weil bereits das Anlegen von Objekten der Klasse bsp02089a.MyData im Hauptprogramm bsp02089.Hauptprogramm misslingt, wie der folgende Bildschirmausdruck zeigt:



Die Spezifikation der in bsp02089.HauptProgramm verwendeten Klasse bsp02089a.MyData lautet wie folgt:

```
package bsp02089a;

class MyData            // (1)
{
// Instanzvariablen
  @SuppressWarnings("unused")  // Unterdrücken der Warnung wg. unbenutzer private lokaler
Variable i1
  private int i1;
  protected int i2;
  public int i3;        // (2)
}
```

Wenig Sinn macht hier das Zusammenspiel der beiden folgenden Punkte:

1. Die Sichtbarkeit der Klasse ist default. Eine Verwendung der Klasse außerhalb des Paketes wird damit unmöglich.
2. Die Sichtbarkeit der Instanzvariablen i3 ist public. Eine solche Sichtbarkeit wird aber bereits in der Klassendeklaration verhindert. Also wird auch insbesondere die Instanzvariable i3 lediglich im Paket bsp02089a sichtbar.

Da bsp02089.HauptProgramm jedoch nicht in der Klasse bsp02089a enthalten ist, ist ein Instantiieren eines Objektes zu bsp02089a.MyData nicht möglich. Damit ist auch eine Verwendung der Instanzvariablen i3 von MyData unmöglich.

2.8.7.4 Wirkung der Modifizierer auf Konstruktoren

Modifizierer haben auf Konstruktoren die gleiche Wirkung wie auf Methoden. Da die Konstruktoren bei der Instantiierung von Objekten verwendet werden, bestimmen die Modifizierer zur Sichtbarkeit damit gleichzeitig, in welchen Klassen Objekte gebildet werden können:

- public Konstruktoren können in allen anderen Klassen aufgerufen werden, die eine entsprechende import-Anweisung enthalten. Damit können in beliebigen Klassen Objekte gebildet werden.
- protected Konstruktoren können nur in Klassen des gleichen Paketes sowie in allen Sohn-Klassen dieser Klassen aufgerufen werden. Damit können in allen Klassen des gleichen Paketes sowie allen Sohn-Klassen Instanzen gebildet werden.
- private Konstruktoren können nur in der Klasse aufgerufen werden, in der sie codiert sind! In diesem Fall ist das Instantiieren der Klasse nur auf dem Umweg über eine Klassenmethode möglich (siehe Beispiel bsp02090).
- Konstruktoren mit default Sichtbarkeit können nur in Klassen des gleichen Paketes aufgerufen werden. Damit ist das Bilden von Objekten nur innerhalb des Paketes möglich.

Für die Programmieraufgaben zu diesem Kapitel bilden also die Sichtbarkeiten public und default die beiden wichtigsten Fälle. Wichtig ist ferner, dass vom Compiler in der Klassendatei automatisch ein parameterloser Konstruktor mit leerem Anweisungsblock erstellt wird, wenn vom Programmierer kein Konstruktor codiert wird. Dieser automatisch erstellt Konstruktor erhält die Sichtbarkeit der umgebenden Klasse.

Als Beispiel betrachten wir den Fall eines Konstruktors mit private Zugriffsschutz. In diesem Fall kann ein Objekt nur über eine Klassenmethode instantiiert werden. In Beispiel bsp02090.ObjektOderNicht wird dies genutzt, um über eine Klassenvariable zu steuern, ob in einer Situation Objekte gebildet werden können oder nicht:

```
package bsp02090;

import de.eikelberg.iotools.*;
```

```
public class ObjektOderNicht
{

// Klassenvariable zur Steuerung, ob Instantiierung möglich ist
  public static boolean instantiierungErlaubt;

// Initialisierungsblock für Klassenvariablen
  static
  {
    instantiierungErlaubt = false;
  }

// Konstruktor
  private ObjektOderNicht()
  {
    ScreenOut.kPrintln(
      "Meldung aus Konstruktor: ");
    ScreenOut.kPrintln(
      "Es wird ein Objekt der Klasse ObjektOderNicht instantiiert ...");
    ScreenOut.kPrintln();
  }

// Klassenmethode, die den private Konstruktor aufruft
// und die gegebenenfalls ein Objekt als Rückgabewert zurückgibt
  public static ObjektOderNicht instantiiere()
  {
    ObjektOderNicht Obj1;
    if (instantiierungErlaubt == false)
    {
      ScreenOut.kPrintln(
        "Instantiierung von ObjektOderNicht ist aktuell nicht erlaubt ...");
      ScreenOut.kPrintln();
// statt eines Objektes die Null-Referenz zuweisen (vgl. Kap 2.8.3.6)
      Obj1 = null;
    }
    else
    {
      Obj1 = new ObjektOderNicht();
    }
    return Obj1;
  }
}
```

Die Klasse ObjektOderNicht besteht aus folgenden vier Bestandteilen:

1. Einer booleschen Klassenvariablen instantiierungErlaubt, über die gesteuert wird, ob Instanzen gebildet werden dürfen oder nicht.
2. Einem Initialisierungsblock für Klassenvariablen, in dem instantiierungErlaubt auf false gesetzt wird
3. Einem Konstruktor mit private Sichtbarkeit, der eine Meldung ausgibt, wenn er durchlaufen wird.
4. Einer Klassenmethode, die ein Objekt Obj1 der Klasse ObjektOderNicht deklariert und die Obj1 in Abhängigkeit des Wertes von instantiierungErlaubt
   - entweder die Null-Referenz zuweisst
   - oder eine neue Instanz von ObjektOderNicht

   Rückgabewert der Klassenmethode ist der Wert von Obj1.

Die Funktion der Klasse bsp02090.ObjektOderNicht wird getestet mit dem folgenden Hauptprogramm:

```
package bsp02090;

import de.eikelberg.iotools.*;

public class HauptProgramm
{

  public static void main(String[] args)
  {
// 0. Startmeldung
    ScreenOut.kStartScreenOut(
      "Beispielprogramm zu Klassen mit private Konstruktor",
      "M. Eikelberg (www.eikelberg.de)","2000");
```

```
// 1. Wert der Variablen instantiierungErlaubt auslesen
    liesInstantiierungErlaubt();

// 2. Erster Versuch, Objekt der Klasse ObjektOderNicht zu instantiieren
    ObjektOderNicht Obj1;
    Obj1 = ObjektOderNicht.instantiiere();

// 3. Untersuchung des erzeugten Objektes
    testeObjekt(Obj1);

// 4. Anlegen von Objekten erlauben
    ObjektOderNicht.instantiierungErlaubt = true;

// 5. Wert der Variablen instantiierungErlaubt auslesen
    liesInstantiierungErlaubt();

// 6. Zweiter Versuch, Objekt der Klasse ObjektOderNicht zu instantiieren
    Obj1 = ObjektOderNicht.instantiiere();

// 7. Untersuchung des erzeugten Objektes
    testeObjekt(Obj1);
  }

// statische Methode zum Lesen der Variablen instantiierungErlaubt
  private static void liesInstantiierungErlaubt()
  {
    ScreenOut.kPrintln(
      "Der Wert der Variablen ObjektOderNicht.instantiierungErlaubt"
      +" ist gleich: " + ObjektOderNicht.instantiierungErlaubt);
    ScreenOut.kPrintln();
  }

// statische Methode zur Untersuchung des erzeugten Objektes
  private static void testeObjekt(ObjektOderNicht Obj1)
  {
// Typ des Objektes Obj1 ausgeben
    try
    {
// Obj1.getClass() ergibt NullPointerException, falls kein Objekt angelegt wurde
// falls Obj1 angelegt wurde, ergibt Obj1.getClass() eine Zeichenkette, die
// Paketnamen und Klassenname zu Obj1 enthält
      ScreenOut.kPrintln(
        "Es wurde ein Objekt der Klasse '"+Obj1.getClass()+"' instantiiert!");
      ScreenOut.kPrintln();
    } catch (NullPointerException e)
    {
      ScreenOut.kPrintln(
        "Es wurde kein Objekt angelegt - NullPointerException!");
      e.printStackTrace();
      ScreenOut.kPrintln();
    }
  }
}
```

Die Verarbeitungsschritte der Methode main(String[] args) sollten zu folgendem Ergebnis führen:

1. Es wird festgestellt, dass der Wert der Variablen instantiierungErlaubt gleich false ist.
2. Die Instantiierung eines Objektes der Klasse ObjektOderNicht gelingt nicht, stattdessen gibt die Methode instantiiere() den null-Pointer zurück.
3. Die Methode testObjekt(ObjektOderNicht) findet den null-Pointer. Der Zugriff Obj1.getClass() ergibt eine NullPointerException in der Methode. Diese Exception wird gefangen und auf dem Konsolbildschirm berichtet.
4. Die Variable instantiierungErlaubt erhält den Wert true,
5. Es wird festgestellt, dass der Wert der Variablen instantiierungErlaubt gleich true ist.
6. Die Instantiierung eines Objektes der Klasse ObjektOderNicht gelingt. Der Konstruktor schreibt bei der Instantiierung eine entsprechende Meldung auf die Konsole. Die Methode instantiiere() gibt das instantiierte Objekt der Klasse ObjektOderNicht zurück.
7. Der Test von Obj1 mit der Methode testeObjekt(ObjektOderNicht) ergibt, dass in der Tat ein Objekt der Klasse ObjektOderNicht angelegt wurde.

Der folgende Bildschirmausdruck zeigt, dass die Verarbeitung von bsp02090.HauptProgramm wie beschrieben abläuft.

```
Administrator: Eingabeaufforderung

...\jpr02090>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02090.HauptProgramm

--------------------------------------------------------------------------------
    Beispielprogramm zu Klassen mit private Konstruktor
    Copyright 2000 by M. Eikelberg (www.eikelberg.de)

--------------------------------------------------------------------------------
Der Wert der Variablen ObjektOderNicht.instantiierungErlaubt ist gleich: false

Instantiierung von ObjektOderNicht ist aktuell nicht erlaubt ...

Es wurde kein Objekt angelegt - NullPointerException!
java.lang.NullPointerException
        at bsp02090.HauptProgramm.testeObjekt(HauptProgramm.java:57)
        at bsp02090.HauptProgramm.main(HauptProgramm.java:23)

Der Wert der Variablen ObjektOderNicht.instantiierungErlaubt ist gleich: true

Meldung aus Konstruktor:
Es wird ein Objekt der Klasse ObjektOderNicht instantiiert ...

Es wurde ein Objekt der Klasse 'class bsp02090.ObjektOderNicht' instantiiert!
```

2.8.8 Datenrückgabe aus Methoden mit Objekten

Dieses Kapitel befasst sich mit der Datenrückgabe aus Methoden an die rufende Methode. Basis der Betrachtungen sind folgende Grundregeln:

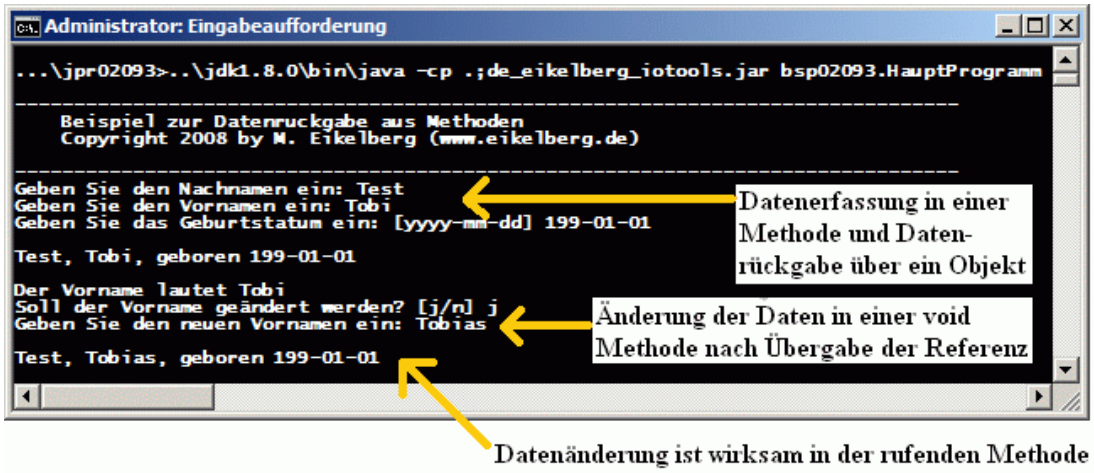
- Beim Methodenaufruf werden Variablen stets mit "call by value" übergeben. Die Methode arbeitet also stets mit einer Kopie der Daten:
  - Bei einfachen Datentypen wird also beim Methodenaufruf der Wert der Variablen kopiert. Eine Veränderung des Wertes wirkt sich damit *nicht* auf den Wert in der rufenden Methode aus (vgl. Kap. 2.7.2.3).
  - Bei Referenzdatentypen wird beim Methodenaufruf die Refenz kopiert. Die gerufene Methode arbeitet somit mit dem gleichen Objekt wie die rufende Methode. Eine Veränderung der Daten des Objektes in der gerufenen Methode *bewirkt somit auch die Veränderung der Objekt-Daten in der rufenden Methode* (vgl. Kap. 2.7.2.4).
- Eine Methode kann *stets nur einen Rückgabewert* haben. Dieser Rückgabewert kann ein einfacher Datentyp oder ein Referenzdatentyp sein (vgl. Kap. 2.7.1.4).

Damit besteht zunächst das Problem, dass umfangreichere Daten nicht über mehrere Variablen aus einer Methode zurückgegeben werden können. Die Lösung des Problems besteht darin,

- zurückzugebende Daten in einem Objekt zusammenzufassen und dieses Objekt zurückzugeben.

Das folgende Beispielprogramm bsp02093.HauptProgramm zeigt

- das Vorgehen bei Datenrückgabe in einem Objekt sowie
- das Vorgehen bei Veränderung der Daten einer rufenden Methode nach Methodenaufruf mit einem Referenzdatentyp als Parameter:



Erläuterung des Beispiels - Klasse Person

Im vorliegenden Beispiel werden die vom Programm bearbeiteten Daten in einem Objekt der folgenden Klasse Person gespeichert. Diese Klasse enthält

- private Instanzvariablen (Attribute) für die Personendaten
- einen Konstruktor mit Parametern zur Datenübergabe an das neue Objekt bei der Instanziierung (vgl. Kap. 2.8.5.5)
- get- und set-Methoden zum Lesen und Verändern der Personendaten

```
package bsp02093;

public class Person
{
  /*
   * Attribute
   */
   private String vorName = "";
   private String nachName = "";
//   Es gibt bessere Wege, ein Geburtsdatum zu speichern (vgl. jpr02113). Diese sind
//   allerdings aufwändig und werden daher für dieses Beispiel nicht gewählt
   private String geburtsDatum = "0000-00-00";

  /*
   * Konstruktoren
   */
   public Person(String nachName, String vorName, String geburtsDatum)
```

```
    {
      this.vorName = vorName;
      this.nachName = nachName;
      this.geburtsDatum = geburtsDatum;
    }

  /*
   * get-Methoden
   */
  public String getNachName()
    {
      return this.nachName;
    }

  public String getVorName()
    {
      return this.vorName;
    }

  public String getGeburtsDatum()
    {
      return this.geburtsDatum;
    }

   public String getPersonendaten()
    {
      String pDat;
      pDat = this.nachName + ", " + this.vorName +
          ", geboren " + this.geburtsDatum;
      return pDat;
    }

  /*
   * set-Methoden
   */
  public void setNachName(String nachName)
    {
      this.nachName = nachName;
    }

  public void setVorName(String vorName)
    {
      this.vorName = vorName;
    }

  public void setGeburtsDatum(String geburtsDatum)
    {
      this.geburtsDatum = geburtsDatum;
    }
}
```

Erläuterung des Beispiels - Datenrückgabe in einem Objekt

Im Beispiel bsp02093.HauptProgramm erfolgt in der main-Methode folgender Methodenaufruf einer statischen Methode der Klasse bsp02093.Verarbeitung:

```
  Person p;
  p = Verarbeitung.erfassePersonenDaten();
```

- In der ersten Anweisung wird eine Variable deklariert, die die Referenz auf ein Objekt der Klasse Person speichern kann.
- In der zweiten wird eine Methode aufgerufen, die als Rückgabewert eine Referenz auf ein Objekt der Klasse Person zurückgibt. In diesem Objekt sind die von der gerufenen Methode erfassten Personendaten gespeichert. Die gerufene Methode ist wie folgt codiert:

```
// Methode zur Erfassung von Personendaten
  public static Person erfassePersonenDaten()
  {
    ScreenOut.kPrint("Geben Sie den Nachnamen ein: ");
    String nachName = KeyboardIn.kReadLine();
    ScreenOut.kPrint("Geben Sie den Vornamen ein: ");
    String vorName = KeyboardIn.kReadLine();
    ScreenOut.kPrint("Geben Sie das Geburtstatum ein: [yyyy-mm-dd] ");
    String gebDatum = KeyboardIn.kReadLine();
// Datenrückgabe über ein Objekt
```

```
    Person p;
    p = new Person(nachName,vorName,gebDatum);
    return p;
  }
```

Erläuterung des Beispiels - Datenrückgabe durch Änderung von Objektdaten nach Übergabe der Referenz

Der zweite Abschnitt des Beispielprogramms bsp02093.HauptProgramm besteht aus folgender Anweisung:

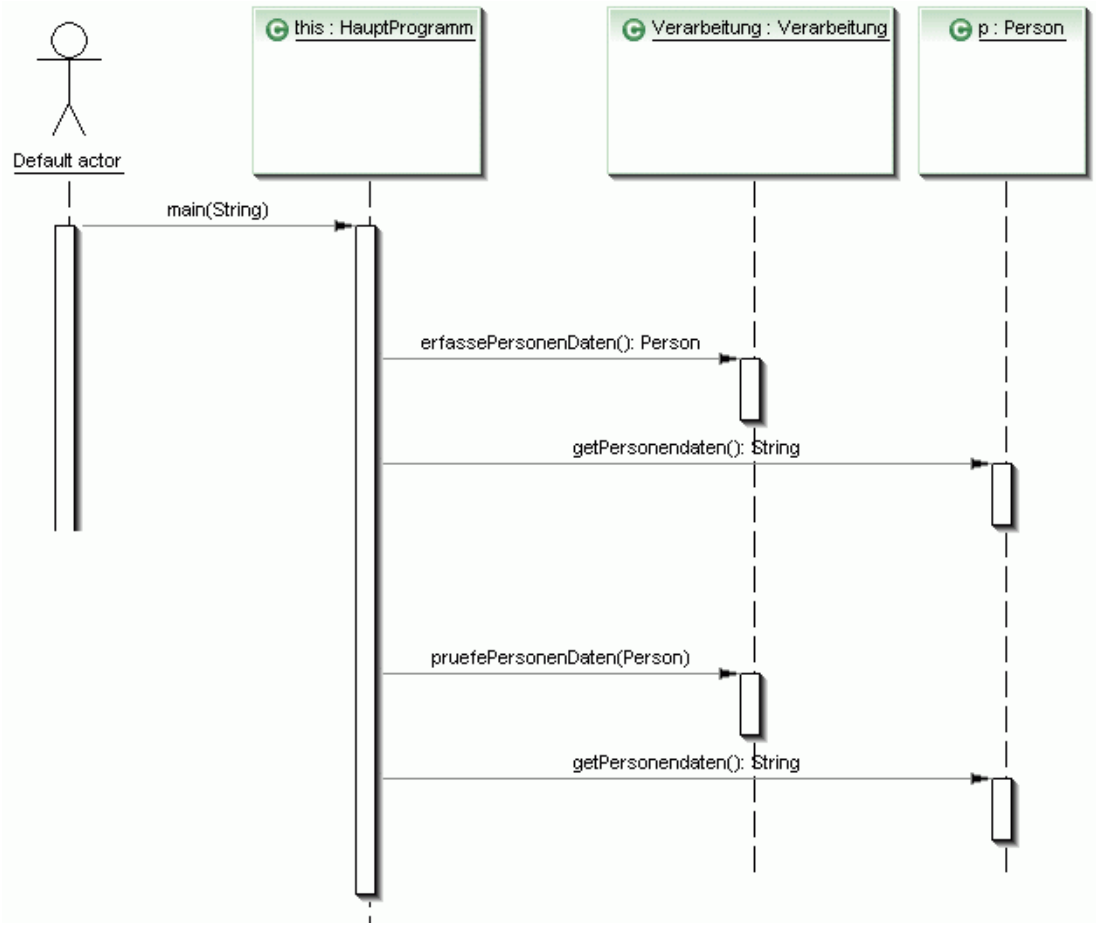
```
Verarbeitung.pruefePersonenDaten(p);
```

- Im Programm wurde weiter oben ein Objekt p der Klasse `Person` angelegt. Bei dem Aufruf der Methode `pruefePersonenDaten(Person p)` wird die Referenz auf das Objekt p an die Methode weitergegeben. Die Methode verarbeitet damit das (identische) Objekt der rufenden Methode.
- In der gerufenen Methode werden die Objektdaten wie folgt geändert:

```
  public static void pruefePersonenDaten(Person p)
  {
    ScreenOut.kPrintln("\nDer Vorname lautet "+p.getVorName());
    ScreenOut.kPrint("Soll der Vorname geändert werden? [j/n] ");
    char c = KeyboardIn.kReadLine().trim().charAt(0);
    if (c == 'j' | c == 'J')
    {
      ScreenOut.kPrint("Geben Sie den neuen Vornamen ein: ");
      String neuerVorName;
      neuerVorName = KeyboardIn.kReadLine();
// Die nächste Anweisung führt zur Datenänderung für die
// rufende Methode, ohne dass der Programmierer der
// rufenden Methode dies bemerkt!
      p.setVorName(neuerVorName);
    }
```

Beurteilung der beiden Vorgehensweisen

Die Betrachtung der Frage, welche der beiden Vorgehensweisen empfehlenswerter ist, erfolgt anhand des nachfolgenden UML-Sequenzdiagramms. Das Diagramm zeigt, dass mit beiden Vorgehensweisen die gleiche Struktur in der Software erzeugt wird:



Beide Vorgehensweisen sind somit geeignet. Die Entscheidung, welche der Vorgehensweisen gewählt wird, muss also inhaltlich erfolgen wie

zum Beispiel mit den nachfolgenden Argumenten:

- Die Erfassung der Daten einer neuen Person erfordert das Anlegen eines neuen Person-Objektes. Dieses ist beim Methodenaufruf noch nicht vorhanden, muss also in der gerufenen Methode erzeugt werden und ist damit der Rückgabewert der Methode.
- Die Änderung der Daten einer bereits gespeicherten Person setzt voraus, dass bereits ein Person-Objekt vorhanden ist. Damit muss die Referenz auf das Objekt an die gerufene Methode übergeben werden. Der Name des Methodenaufrufs lässt erkennen, dass die Objektdaten ggf. geändert werden. Damit ist es nicht erforderlich, dass das Person-Objekt auch Rückgabewert ist.

2.8.9 Beispiele zum klassenbasierten Programmentwurf

2.8.9.1 Beispiel - Modellierung eines Rechners zur Ausführung der Grundrechenarten

Generell dient die Verwendung der in 2.8.7 erklärten Modifizierer dem Schutz der Daten in den Variablen. Um einen möglichst hohen Schutz und damit einen möglichst positiven Effekt in Bezug auf die Qualität des Programms zu erreichen, sollte man sich nach folgenden Regeln richten:

1. Man wähle die Sichtbarkeit stets so gering wie möglich.
2. Nicht als final gekennzeichnete Klassen- und Instanzvariablen deklariere man möglichst als private. Soweit erforderlich codiere man anschließend für den schreibenden oder lesenden Zugriff Set- und Get-Methoden zu den Variablen (vgl. 2.8.6).

Anwendung finden die Modifizierer auch bei der Umsetzung der sogenannten objektorientierten Analyse sowie des sogenannten objektorientierten Designs. Diese Vorgehensweise basiert auf der Idee, dass man Klassen, Variablen und Methoden so entwirft, dass man das Verhalten nachbildet, das ein System oder ein Gerät der Wirklichkeit zeigt. Hierzu bildet man zunächst die nach außen sichtbaren Bestandteile ab. Anschließend erarbeitet man die interne Realisierung, die man bei der Implementierung mit Hilfe der Modifikatoren zur Sichtbarkeit auch programmtechnisch versteckt. In diesem Kapitel soll dieses Vorgehen an einem Beispiel erkäutert werden. Im Kapitel 3 werden das entsprechende Vorgehen und die zugehörigen Hilfsmittel eingeführt.

Da das Ziel der Programmentwicklung an dieser Stelle des Kurses die Erstellung eines klassenbasierten (objektorientierten) Programms ist, wird das Vorgehen der objektorientierten Analyse und des objektorientierten Designs zunächst auf die klassenbasierter Programmierung zugeschnitten und entsprechend verkürzt erklärt. Daher wird im Folgenden auch zunächst vom **klassenbasierten Programmentwurf** gesprochen.

Als Beispiel für die Vorgehensweise beim klassenbasierten Programmentwurf entwickeln wir eine Klasse, die Grundrechenarten für reelle Zahlen realisiert. Um eine geeignete Vorstellung über den richtigen Aufbau dieser Klasse zu bekommen, stellen wir uns die Klasse zunächst als ein Gerät (black box) mit Namen GrundRechner vor, die Daten erhält, Funktionen durchführt und Ergebnisse liefert. Wir klären also zunächst folgende Fragen

1. Welche Funktionen soll das Gerät GrundRechner realisieren?
2. Welche Daten muss das Gerät GrundRechner von uns erhalten?
3. Welche Ergebnisse soll das Gerät GrundRechner liefern?

Diese Fragen können zum Beispiel wie folgt beantwortet werden:

1. Das Gerät soll die vier Grundrechenarten ausführen. Hierzu muss das Gerät vier Bedienungselemente erhalten.
2. Jede Grundrechenart benötigt zwei Operanden. Das Gerät muss also von uns zwei Operanden erhalten. Die Bereitstellung eines oder keines Operanden führt dazu, dass das Gerät nicht funktionsfähig ist.
3. Das Gerät muss Ergebnisse zu den vier Grundrechenarten ausgeben. Falls das Gerät keine Daten für die Operanden erhalten hat, soll es eine Fehlermeldung herausgeben.

Mit der Beantwortung der obigen beiden Fragen ist die nach außen sichtbare Struktur des Gerätes GrundRechner geklärt. Wir gehen jetzt zur programmtechnischen Realisierung über und sammeln folgende Überlegungen:

a. Das Gerät muss sich merken, ob es Daten für die Operanden erhalten hat. Nach der Ausgabe des Ergebnisses einer Operation muss das Gerät erkennen, dass es neue Daten für die Operanden benötigt.
b. Wird eines der vier Bedienungselemente des Gerätes ausgelöst, sollte folgender Ablauf durchgeführt werden:
    - Prüfung, ob Daten zu den Operanden vorhanden sind.
    - Wenn Operanden vorhanden sind:
        - Berechnung der angeforderten Operation
        - Rückgabe des Ergebnisses der Operation
    - Wenn keine Operanden vorhanden sind:
        - Rückgabe einer Fehlermeldung.
c. Das Gerät muss sich zum Abschluss merken, dass neue Operanden für die nächste Operation benötigt werden.
d. Die Besonderheiten bei der Durchführung der Divisionsoperation (Division durch Null) müssen beachtet werden.

Eine programmtechnische Umsetzung der Punkte a bis d ergibt folgende Erkenntnisse zu den Daten der Klasse GrundRechner:

e. Die zu codierende Klasse benötigt eine boolean Variable **bereit**. Diese kann **private** gesetzt werden. Sie wird gleich true gesetzt, wenn Daten zu den Operanden vorhanden sind und gleich false, wenn keine Daten zu den Operanden vorhanden sind.
f. Die interne Verarbeitung erfolgt am besten mit Datentyp double. Es werden zwei double Variablen **op1** und **op2** für die Summanden sowie eine double Variable **res** für das Rechenergebnis benötigt. Die drei Variablen op1, op2 und res können als **private** deklariert werden.
g. Das Ergebnis der Operation kann auf einer String-Variablen **Meldung1** gespeichert werden. Diese String-Variable kann als lokale Variable der Methode main(String[]) deklariert werden. Die Analyse der Situation (vgl. 3.3.4) mit den Methoden aus Kapitel 3.3 ergibt, dass es am günstigsten ist, den Inhalt der Variablen Meldung1 mit Hilfe einer Klassenmethode **druckeErgebnis(String)** durchzuführen. Ferner ergibt diese Analyse, dass die Methode druckeErgebnis(String) am besten in einer gesonderten Hilfsmittel-

Klasse **DruckTools** angeordnet werden sollte.

In Bezug auf die Methoden liefert die weitere Analyse der Punkte a bis d folgende Erkenntnisse:

h. Die Übergabe der Operanden an das Gerät kann am besten durch **public Set-Methoden** erfolgen, die in den Datentyp double umwandeln. Hierbei ist es günstig, die Set-Methoden zu überladen (d.h. sie alle Set-Methoden sollten den gleichen Methodennamen haben).
i. Jedes der vier **Bedienungselemente** des Gerätes kann als **public** Methode realisiert werden. Jede dieser Methoden liefert den Rückgabewert in Form einer String-Variablen, die in der lokalen Variablen Meldung1 zwischengespeichert und anschließend über die Methode druckeErgebnis(String) ausgedruckt wird.
j. Das Gerät GrundRechner benöitgt eine interne Methode **resetGrundRechner()**, die nach einer Berechnung bewirkt, dass der Grundrechner neue Daten für die Operanden erwartet. Diese Methode muss als **private** gewählt werden, da sie von außen nicht aufgerufen werden darf.

In Java Quellcode umgesetzt erhalten wir aus diesen Punkten folgende Struktur der Klasse GrundRechner:

```
package bsp02086;

class GrundRechner
{

  private double op1,op2,res;
  private String Meldung1;
  private boolean bereit;

// Rechner ist nicht bereit nach Instantiierung:
  public GrundRechner()
  {
    resetGrundRechner();
  }

// Set-Methode für ganzzahlige Operanden
  public void setzeOperanden(long op1, long op2)
  {
    ... // Verarbeitungsschritte zur Set-Methode
  }

// Set-Methode für Gleitpunkt-Operanden
  public void setzeOperanden(double op1, double op2)
  {
    ... // Verarbeitungsschritte zur Set-Methode
  }

// Set-Methoden für gemischte Typen von Operanden
  public void setzeOperanden(long op1, double op2)
  {
    ... // Verarbeitungsschritte zur Set-Methode
  }

  public void setzeOperanden(double op1, long op2)
  {
    ... // Verarbeitungsschritte zur Set-Methode
  }

// Bedienungselemente zu den vier Grundrechenarten
  public String berechneSumme()
  {
    ... // Verarbeitungsschritte zur Addition
    resetGrundRechner();
    return Meldung1;
  }

  public String berechneDifferenz()
  {
    ... // Verarbeitungsschritte zur Subtraktion
    resetGrundRechner();
    return Meldung1;
  }

  public String berechneProdukt()
  {
    ... // Verarbeitungsschritte zur Multiplikation
```

```
    resetGrundRechner();
    return Meldung1;
  }

  public String berechneQuotient()
  {
    ... // Verarbeitungsschritte zur Division
    resetGrundRechner();
    return Meldung1;
  }

// interne Funktionen
  private void resetGrundRechner()
  {
    bereit = false;
  }
}
```

Damit steht die Struktur der Klasse. Es fehlen noch die Verarbeitungsschritte zu den Set-Methoden sowie der Ablauf der vier Verarbeitungsschritte zu den Grundrechenarten. Die Set-Methoden sind im vorliegenden Fall sehr einfach aufgebaut und wandeln die als Argument übergebenen unterschiedlichen Daten stets in den Datentyp double um:

```
// Set-Methode für ganzzahlige Operanden
  public void setzeOperanden(long op1, long op2)
  {
    this.op1 = (double)op1;
    this.op2 = (double)op2;
    this.bereit = true;
  }

// Set-Methode für Gleitpunkt-Operanden
  public void setzeOperanden(double op1, double op2)
  {
    this.op1 = op1;
    this.op2 = op2;
    this.bereit = true;
  }

// Set-Methoden für gemischte Typen von Operanden
  public void setzeOperanden(long op1, double op2)
  {
    this.op1 = (double)op1;
    this.op2 = op2;
    this.bereit = true;
  }

  public void setzeOperanden(double op1, long op2)
  {
    this.op1 = op1;
    this.op2 = (double)op2;
    this.bereit = true;
  }
```

Man sieht, dass alle vier Set-Methoden den gleichen Methoden-Namen haben und damit überladen sind. Zusammen mit der automatischen Typkonversion beim Methodenaufruf ist es damit möglich, die Übergabe der Operanden für alle sechs einfachen Datentypen zu Zahlwerten wie folgt zu codieren:

```
  setzeOperanden(Zahlwert1,Zahlwert2)
```

Den Ablauf der Verarbeitungsschritte zu den vier Grundrechenarten beschreibt man am besten durch Struktogramme. Auf diese Weise erhält man zum Beispiel die folgende Implementierung der Verarbeitungsschritte:

```
// Bedienungselemente zu den vier Grundrechenarten
  public String berechneSumme()
  {
    if (bereit == true)
    {
      res = op1 + op2;
      Meldung1 = "Ergebnis: "+op1+" + "+op2+" = "+res;
    }
    else
    {
```

```
      Meldung1 = "Gerät nicht bereit - Daten zu Operanden fehlen";
    }
    resetGrundRechner();
    return Meldung1;
  }

  public String berechneDifferenz()
  {
    if (bereit == true)
    {
      res = op1 - op2;
      Meldung1 = "Ergebnis: "+op1+" - "+op2+" = "+res;
    }
    else
    {
      Meldung1 = "Gerät nicht bereit - Daten zu Operanden fehlen";
    }
    resetGrundRechner();
    return Meldung1;
  }

  public String berechneProdukt()
  {
    if (bereit == true)
    {
      res = op1 * op2;
      Meldung1 = "Ergebnis: "+op1+" * "+op2+" = "+res;
    }
    else
    {
      Meldung1 = "Gerät nicht bereit - Daten zu Operanden fehlen";
    }
    resetGrundRechner();
    return Meldung1;
  }

  public String berechneQuotient()
  {
    if (bereit == true)
    {
      if (op2 == 0)
      {
        Meldung1 = "Operation nicht möglich - Division durch Null";
      }
      else
      {
        res = op1 / op2;
        Meldung1 = "Ergebnis: "+op1+" / "+op2+" = "+res;
      }
    }
    else
    {
      Meldung1 = "Gerät nicht bereit - Daten zu Operanden fehlen";
    }
    resetGrundRechner();
    return Meldung1;
  }
```

Man beachte, dass bis zu dieser Stelle kein Hauptprogramm entwickelt wurde, sondern eine Klasse mit Namen GrundRechner, die die Funktionalität eines Taschenrechners nachbildet. Die Klasse GrundRechner besteht aus nach außen sichtbaren Bedienungselementen sowie Elementen zur Eingabe und Ausgabe von Daten. Die interne Struktur ist durch die Anordnung der Daten in private Variablen und durch die Deklaration der internen Methode als private nicht nach außen sichtbar. Sie kann daher auch nicht von außen gestört werden.

Das gezeigte Vorgehen bei der Modellierung und Entwicklung der Klasse GrundRechner ist beispielhaft für den allgemeinen Fall. Sie verläuft - wie man leicht sieht - anders als die Entwicklung eines strukturierten Hauptprogramms, jedoch können die Verarbeitungsschritte der einzelnen Methoden im Allgemeinen mit Struktogrammen entwickelt werden. Um wie im Beispiel der Entwicklung der Klasse GrundRechner geeignete (Groß-) Strukturen zu finden, erfolgt zunächst eine Modellierung nach der außen sichtbaren Funktionalität und anschließend die schrittweise Konkretisierung der 'inneren' Abläufe und Zustände. Ein Hilfsmittel zu einer solchen **(objektorientierten) Analyse** und zur Erstellung des entsprechenden **(objektorientierten) Designs** sind im Allgemeinen die Graphiken der sogenannten **Unified Modeling Language (UML)**. Die wichtigsten Hilfsmittel der UML werden in Kapitel 3.3 und 3.4 vorgestellt.

Wir beenden das Kapitel mit dem Programmcode der Hilfsmittel-Klasse bsp02086.DruckTools sowie mit dem Testprogramm

bsp02086.GrundRechnerTest, in dem die Verarbeitung der Klasse GrundRechner getestet wird. Es folgt zunächst der Programmcode der Hilfsmittelklasse bsp02086.DruckTools:

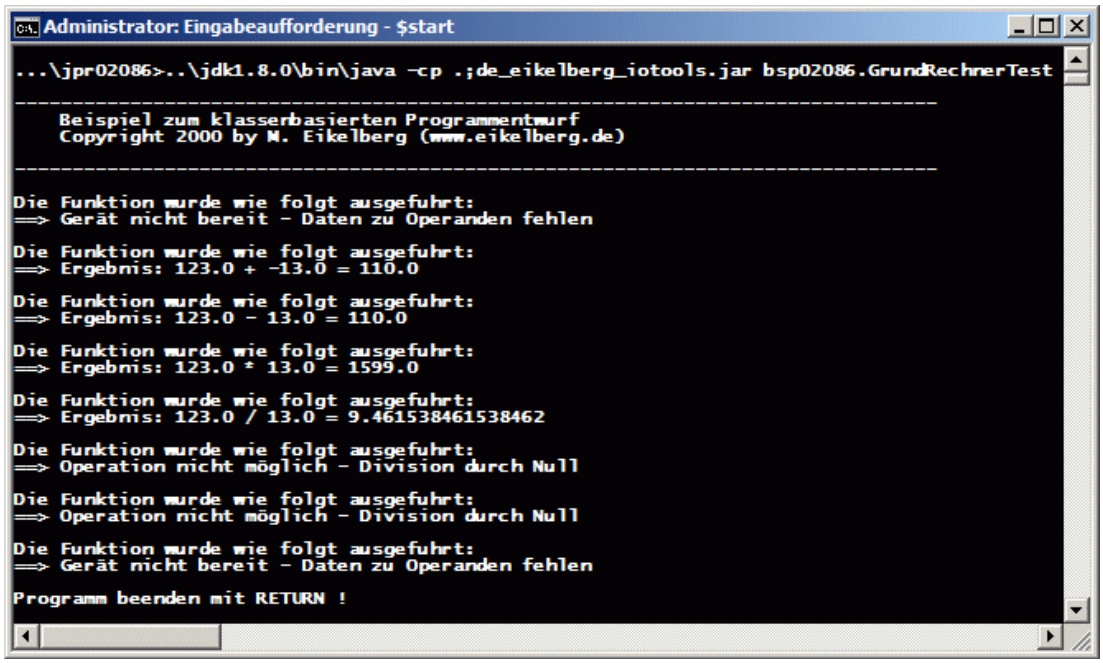
```
package bsp02086;

import de.eikelberg.iotools.*;

// default Zugriff:
// Zugriff nur aus Klassen des Paketes bsp02086 möglich
class DruckTools
{

// kein Instantiieren der Klasse möglich
  private DruckTools()
  {
  }

// Klassenmethode druckeErgebnis(...)
  protected static void druckeErgebnis(String Meldung)
  {
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("Die Funktion wurde wie folgt ausgeführt: ");
    ScreenOut.kPrintln("==> "+Meldung);
  }
}
```

Die Klasse bsp02086.GrundRechner kann jetzt unter Verwendung der folgenden Klasse bsp02086.GrundRechnerTest getestet werden:

```
package bsp02086;

import de.eikelberg.iotools.*;

class GrundRechnerTest
{

  public static void main(String[] args)
  {
// Startmeldung ausgeben
    ScreenOut.kStartScreenOut(
      "Testklasse zur Verwendung von Modifzierern",
      "M. Eikelberg (www.eikelberg.de)","2000");

// Objekt der Klasse Grundrechner instantiieren
    GrundRechner MyRechner1;
    MyRechner1 = new GrundRechner();

// erster Aufruf einer Funktion ergibt Fehler
    String Meldung1 = MyRechner1.berechneSumme();
    DruckTools.druckeErgebnis(Meldung1);

// Durchführung einer Summation
    MyRechner1.setzeOperanden(123,-13);
    Meldung1 = MyRechner1.berechneSumme();
    DruckTools.druckeErgebnis(Meldung1);

// Durchführung einer Subtraktion
    MyRechner1.setzeOperanden(123,13.0f);
    Meldung1 = MyRechner1.berechneDifferenz();
    DruckTools.druckeErgebnis(Meldung1);

// Durchführung einer Multiplikation
    MyRechner1.setzeOperanden(123.0f,13);
    Meldung1 = MyRechner1.berechneProdukt();
    DruckTools.druckeErgebnis(Meldung1);

// Durchführung einer Division
    MyRechner1.setzeOperanden(123.0,13.0);
    Meldung1 = MyRechner1.berechneQuotient();
    DruckTools.druckeErgebnis(Meldung1);

// Durchführung einer fehlerhaften Division
    MyRechner1.setzeOperanden(123,0);
```

```
    Meldung1 = MyRechner1.berechneQuotient();
    DruckTools.druckeErgebnis(Meldung1);

// Durchführung einer fehlerhaften Division
    MyRechner1.setzeOperanden(0,0);
    Meldung1 = MyRechner1.berechneQuotient();
    DruckTools.druckeErgebnis(Meldung1);

// erneut Verarbeitung ohne Daten versuchen
    Meldung1 = MyRechner1.berechneQuotient();
    DruckTools.druckeErgebnis(Meldung1);

// Bildschirmausgabe anhalten
    HoldScreen.stop();
  }
}
```

Die Verarbeitung des Programms bsp02086.GrundRechnerTest führt zu folgender Bildschirmausgabe:



2.8.9.2 Beispiel Grundrechner - Datenrückgabe aus Methoden mit Objekten

Unter Umständen kann es ungünstig sein, dass die Methoden einer Klasse wie in bsp02086.GrundRechner ihr Ergebnis in Form einer Zeichenkette zurückgeben. Bei der Implementierung von bsp02086.GrundRechner wurde dies so umgesetzt, da jede Methode nur einen Rückgabewert haben kann. Soll eine Methode mehrere Rückgabewerte mit unterschiedlichem Datentyp haben, so muss also ein alternatives Vorgehen gefunden werden. Eine sehr elegante Lösung ist die Rückgabe mehrer Ergebniswerte in Form eines Objektes. Dieses Objekt wird gebildet als Instanz der Klasse, die alle Daten eines Ergebnis-Datensatzes bechreibt. Verfolgt man diese Idee weiter, so erkennt man, dass dann auch die Methode druckeErgebnis(...) in die Klasse zu den Ergebniswerten verlegt werden sollte, da sich diese Methode ja genau auf die Ergebniswerte bezieht.

Zur Illustration dieses Vorgehens wird die Klasse bsp02086.GrundRechner so in eine Klasse bsp02087.GrundRechner abgeändert, dass die Rückgabe der vier Verarbeitungsmethoden zu den Grundrechenarten über ein Objekt geschieht. Die Klasse bsp02086.DruckTools wird dabei ersetzt durch die Klasse bsp02087.Ergebnis. Die Übergabe der Daten an diese Klasse geschieht dabei über den Konstruktor. Ferner können die Ergebnisdaten in der Klasse bsp02087.Ergebnis als private gesetzt werden, da die Daten von einer Methode der eigenen Klasse ausgegeben werden. Denkbar wäre auch die Codierung von Get-Methoden, die den Inhalt von einzelnen Variablen an andere Klassen weitergeben. Beispiele für solche Get-Methoden sind die (nicht in bsp02087 verwendeten) Methoden returnErgebnisZahl() und returnMeldung():

```
package bsp02087;

import de.eikelberg.iotools.*;

public class Ergebnis
{

// Instanzvariablen
  private double op1;
  private double op2;
  private double res;
```

```
  private char operation;
  private String Kommentar;

// Konstruktor führt das Speichern der Daten durch
  protected Ergebnis(double op1,
                 double op2,
                 double res,
                 char operation,
                 String Kommentar)
  {
    this.op1 = op1;
    this.op2 = op2;
    this.res = res;
    this.operation = operation;
    this.Kommentar = Kommentar;
  }

// Get-Methode für den Ergebniswert
  protected double returnErgebnisAlsZahl()
  {
    return this.res;
  }

// Get-Methode für die Meldung
  protected String returnMeldung()
  {
    return this.Kommentar;
  }

  protected void druckeErgebnis()
  {
    ScreenOut.kPrintln();
    ScreenOut.kPrintln("Die Funktion wurde wie folgt ausgeführt: ");
    ScreenOut.kPrintln(
      "==> " + Kommentar);
    ScreenOut.kPrintln(
      "==> " + op1 +" "+ operation +" "+ op2 + " = " + res);
  }
}
```

Die Methoden der Klasse GrundRechner zur Verarbeitung der Grundrechenarten werden jetzt so verändert, dass sie ein Objekt der Klasse Ergebnis zurückgeben. Diese Änderung wird im Folgenden beispielhaft für die Methode berechneSumme erklärt:

```
  public Ergebnis berechneSumme()                          // (1)
  {
    operation = '+';
    if (bereit == true)
    {
      res = op1 + op2;
      Meldung1 = "Operation korrekt durchgeführt";         // (2)
    }
    else
    {
      Meldung1 = "Gerät nicht bereit - Daten zu Operanden fehlen";
      resetGrundRechner();                                 // (3)
    }
    Ergebnis Erg1;                                         // (4)
    Erg1 = new Ergebnis(op1,op2,res,operation,Meldung1);   // (5)
    resetGrundRechner();                                   // (6)
    return Erg1;                                           // (7)
  }
```

Es wurden folgende Änderungen in der Methode berechneSumme() durchgeführt:

1. Der Rückgabewert der Methode ist jetzt vom Typ (also von der Klasse) Ergebnis.
2. Die Umarbeitung des Rechenergebnisses in eine Zeichenkette ist an dieser Stelle entfallen. Dafür muss jetzt an dieser Stelle die String-Variable Meldung1 mit einem sinnvollen Wert gefüllt werden.
3. Da die Werte der Variablen op1, op2 und res jetzt stets als Zahlen mit ausgegeben werden, müssen sie im Fehlerfall einen sinnvollen Wert erhalten. Entsprechend wurde die Methode resetGrundRechner so erweitert, dass in ihr die Variablen op1, op2, res und operation mit den Werten 0 bzw. '?' belegt werden.
4. Es wird ein Objekt Erg1 der Klasse Ergebnis deklariert.
5. Das Objekt Erg1 wird instantiiert. Über den Konstruktor wird Erg1 mit Werten gefüllt.

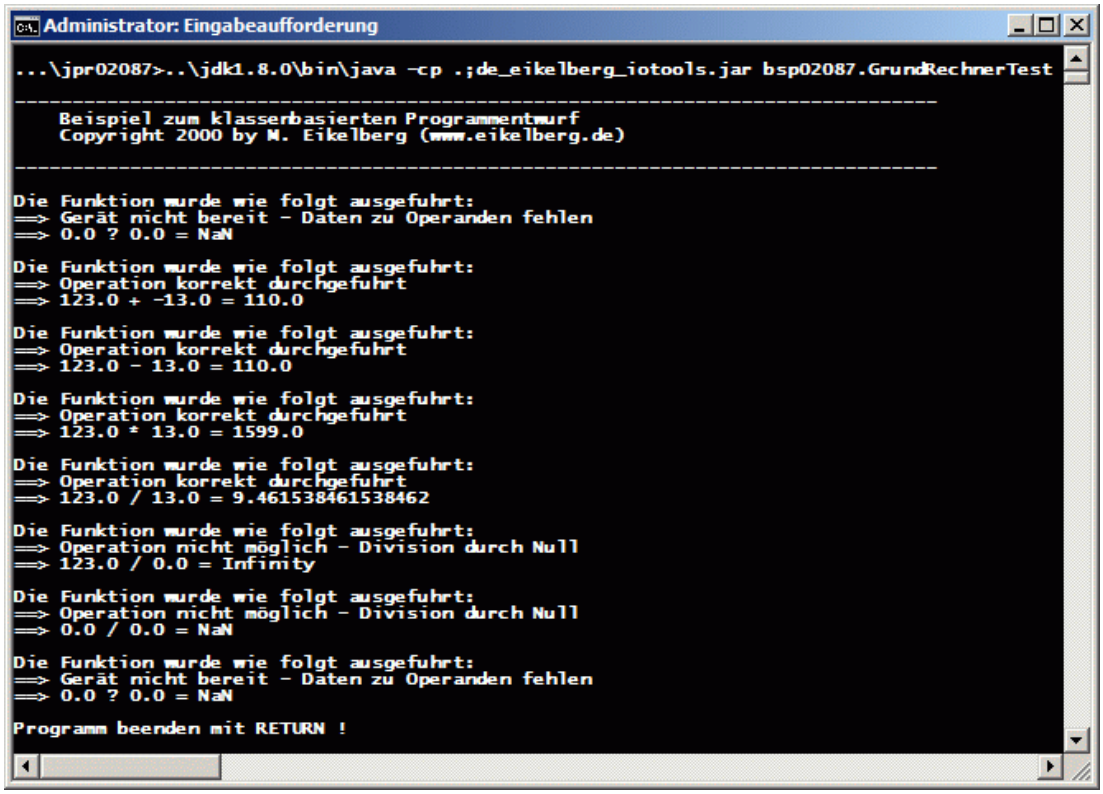
6. Die Variablen op1, op2, res und operation werden zurückgesetzt, das heisst also mit dem Wert 0 bzw. '?' belegt.
7. Das Objekt Erg1 wird als Rückgabewert der Methode festgelegt.

Die für die Methode berechneSummer() gezeigte Umarbeitung ist auch für die drei anderen public Methoden der Klasse bsp02087.GrundRechner durchgeführt worden.

Im Hauptprogramm bsp02087.GrundRechnerTest werden jetzt die modifizierten Methoden wie im Folgenden für den Aufruf der Methode berechneSumme() gezeigt verwendet:

```
package bsp02087;
...
class GrundRechnerTest
{

  public static void main(String[] args)
  {

    ...

// erster Aufruf einer Funktion ergibt Fehler
    Ergebnis Erg1 = MyRechner1.berechneSumme();
    Erg1.druckeErgebnis();

    ...

// Durchführung einer Summation
    MyRechner1.setzeOperanden(123,-13);
    Erg1 = MyRechner1.berechneSumme();
    Erg1.druckeErgebnis();

    ...

  }
}
```

Ein Lauf des Programms bsp02087.GrundRechnerTest ergibt folgende Bildschirmausgabe:



2.9 Objektorientierte Programmierung

2.9.1 Begriffe der objektorientierten Programmierung

2.9.1.1 Übersicht über die Begriffe der objektorientierten Programmierung

Die folgende Tabelle zeigt die wichtigsten drei Begriffe der objektorientierten Programmierung:

Begriff	Kapitel	Beispiel für einen Verwendungszweck
Vererbung	2.9.2	Effektive und geringstmöglich redundante Anordnung von Variablen und Methoden in Klassen
Polymorphismus	2.9.3	Vereinfachung des Programmcode dadurch, dass gleichnamige Methoden in verschiedenen Klassen unterschiedliche Implementierung haben dürfen bzw. dass Objekt-Referenzen stets auch in Objektvariablen übergeordneter Klassen gespeichert werden können.
Kapselung	2.8.6, 2.9.2.4	Kapselung schützt die Inhalte der Klassen vor unbefugter Manipulation durch Bereitstellung von get- und set-Methoden sowie durch Setzen der Variablen als private

Neben diesen zentralen Begriffen gibt es weitere Anforderungen, die eine objektorientierte Programmiersprache erfüllen sollte **(vgl. Horn/Kerner, Band 3, S. 270)**:

Begriff	Kapitel	Beispiel für einen Verwendungszweck
Überladen	2.6.2.7, 2.6.4.3, 2.7.2.1, 2.8.5.4, 2.9.3	Die Programmiersprache sollte zulassen, dass in einer Klasse mehrere gleichnamige Methoden codiert werden, die sich nur durch ihre Parameter unterscheiden. Bei der Java-Programmierung ist es allerdings nicht möglich, dass sich Methoden nur durch ihren Rückgabetyp unterscheiden. Überladene Methoden unterscheiden sich durch ihre Signatur (das ist die Liste der Datentypen der Parameter und des Rückgabewertes).
Überschreiben	2.9.2.6	Bei Vererbung von Methoden und Variablen kann in der erbenden Klasse die geerbte Methode oder Variable durch eine in der erbenden Klasse programmierte Methode oder Variable ersetzt (überschrieben) werden. Bei der Java-Programmierung bis einschießlich JDK 1.4 muss der Rückgabetyp einer überschreibenden Methode vom gleichen Typ wie der Rückgabetyp der überschriebenen Methode sein. Ab JDK 1.5 kann der Rückgabetyp auch ein Typ sein, der durch Vererbung von dem Rückgabetyp der überschriebenen Klassen abgeleitet ist.
strenge Typisierung	2.9.5	Die Programmiersprache sollte Mechanismen bereitstellen, die zur Laufzeit - bei Auftreten von Polymorphismus und Überladen - eine Zuweisung von Methoden verhindert, für die die aktuellen Parameter beim Aufruf nicht zu den formalen Parametern der Methode passen (vgl. 2.7.2.1-2.7.2.2). Dies wird bei der Programmierspache Java durch Compiler-Fehler und Exceptions (2.9.5) realisiert.
abstrakte Datentypen	2.9.4	Die Programmiersprache sollte mindestens eine Funktionalität bereitstellen, wodurch man in Klassen, die in Bezug auf die Vererbung übergeordnet angeordnet sind, die Implementierung von Methoden in untergeordneten Klassen erzwingt, jedoch keine Implementierung in der übergeordneten Klasse bereitstellt. Java kennt hier die Eigenschaft abstrakt für Klassen mit abstrakten Methoden sowie die Schnittstelle (Interface).
Nebenläufigkeit	3.9	Der Methodenaufruf und das Ausführen von Methoden sollte zeitlich parallel durchgeführt werden können. Java stellt hierzu Threads zur Verfügung sowie Methoden zur Serialisierung von zeitlich nicht parallel durchführbaren Vorgängen.
Persistenz	4	Die Programmiersprache sollte eine auch über das Ende des Programmlaufs mögliche Speicherung der Daten eines Objektes ermöglichen. Diese erfolgt heutzutage in der Regel mit Hilfe von Java Database Connectivity (JDBC) auf SQL-Servern, wobei auf den SQL-Servern - nach Durchführung eines Datenbankentwurfs - für jede Klasse eine oder mehrere Tabellen bereitgestellt werden. Bei Datenaustausch zwischen Systemen werden auch Textdateien mit XML-Code verwendet.

Programmiersprachen, die das Programmieren unter Verwendung der oben genannten Prinzipien unterstützen heißen **objektorientiert** (**(vgl. Horn/Kerner, Band 3, S. 270)**:

Eigenschaft der Programmiersprache	Kapitel	Methodik die die Programmiersprache unterstützen muss
objektbasiert	2.3.3	Die Programmiersprache ermöglicht die Arbeit mit Objekten zu bereits in der Sprache fest vorgegebenen Klassen.
klassenbasiert	2.8	Die Programmiersprache ist objektbasiert und ermöglicht ferner das Programmieren benutzerdefinierter Klassen, für die anschließend auch Objekte angelegt werden können.

objektorientiert	2.9	Die Programmiersprache ist klassenbasiert und ermöglicht mindestens eine Kapselung der Klassen sowie die Anordnung von Klassen, so dass Vererbung und Polymorphismus realisiert werden können.

2.9.1.2 UML Klassendiagramme

Das wichtigste Hilfsmittel zur Beschreibung von Klassen und den zwischen den Klassen bestehenden objektorientierten Beziehungen ist das UML Klassendiagramm. Es kann sehr einfach mit dem Omodo eclipseUML Plugin (Version 2.1.0 ist geeignet für eclipse 3.2) gezeichnet werden. Die folgende Tabelle nennt die benötigten UML Diagrammelemente sowie ihre Darstellung in eclipseUML. Die Auflistung der UML-Elemente erfolgt dabei gemäß dem Buch **Heide Balzert, UML kompakt**.

Begriff	UML-Element	Darstellung in eclipseUML
Klasse	Klasse mit Namensfeld, Attributliste (Variablen) und Operationsliste (Methoden)	MyClass1 myVar1: String loesche() myMethod1(in windows: boolean, in x: float): int myMethod2(in drucke: String) ←Namensfeld ←Attributliste mit Attribut ←Operationsliste mit Operationen
Attribut	Attribut (Variable)	
Operation	Operation (Methode)	Farbe grün / C = Class / S = statische Variable oder Methode
abstrakte Klasse	Klasse mit mindestens einer abstrakten Methode	MyClass2 wert: double fuegeEin(in wert: double) loesche() A = abstrakte Methode bzw. abstrakte Klasse
Sichtbarkeit	Sichtbarkeit	MyClass3 myVar1: String wert: double zaehler: int
Signatur	Datentyp Parameter und Rückgabewert	fuegeEin(in wert: double) getMyVar1(): String loesche() main(in args: String[]) myMethod1(in windows: boolean, in x: float): int myMethod2(in drucke: String) setMyVar1(in myVar1: String) Datentyp Parameter Datentyp Rückgabewert grüner Kreis = public / gelbes Karo = protected / rotes Quadrat = private
Klassen-attribut	Unterstreichung des Attributnamens	Markierung des Attributs mit "S"
Klassen-operation	Unterstreichung des Methodennamens	Markierung der Methode mit "S"
Vererbung	Vererbung (generalization)	MyClass1 myVar1: String loesche() myMethod1(in windows: boolean, in x: float): int myMethod2(in drucke: String) MyClass4 myVar1: String myMethod3() Vererbungspfeil - MyClass4 erbt von MyClass1

Schnittstelle	Schnittstelle (Interface)	«interface» MyInterface1 fuegeEin(in wert: double) getStatus(): int Farbe lila / I = Interface / Beschriftung <<interface>>
Implementierung einer Schnittstelle	gestrichelter Vererbungspfeil	«interface» MyInterface1 fuegeEin(in wert: double) getWert(): double MyImplementierung1 wert: double fuegeEin(in wert: double) getWert(): double
Abhängigkeit	Abhängigkeit (Dependency)	MyClass5 main(in args: String[]) «access» MyClass6 verarbeite() MyClass5 greift auf MyClass6 zu
Paket	Paket	bsp02110a ToolKlasse verarbeite() «access» MyClass7 main(in args: String[])

2.9.1.3 Erste Beispiele für die Bedeutung der Prinzipien der objektorientierten Programmierung

Die Prinzipien der objektorientierten Programmierung werden bei der fortgeschrittenen Java-Programmierung ständig verwendet. Hier einige Beispiele für die Anwendung von

- Vererbung und
- Schnittstellen

in den nachfolgenden Kapiteln:

Beispiele zur Vererbung:

Das Java-Schlüsselwort für die Vererbung lautet `extends`.

```
MyFrame extends JFrame
```

bedeutet: Die Klasse MyFrame erweitert die Klasse JFrame, erbt also die Variablen und Methoden von JFrame. Die nachfolgende Tabelle listet einige Vererbungsvorgänge auf, die im weiteren Verlauf dieses Kurses auftreten:

Vererbungsvorgang	Beispiel-Klasse	Zweck des Vererbungsvorgangs
`extends JFrame`	bsp03010.SimpleExample	Aufbau eines Fensters
`extends JApplet`	bsp03015.SimpleJApplet	Aufbau eines Applets
`extends JPanel`	bsp03022.MyPanel	Programmierung eines Stücks Benutzeroberfläche
`extends KeyAdapter` `extends MouseAdapter`	bsp03035.SimpleJFrame	Behandlung von Tastatur- und Mausereignissen
`extends Thread`	bsp02063.ZufallsZahlThread	Programmierung von Threads

Beispiele zur Verwendung von Schnittstellen:

Die Implementierung von Schnittstellen erfolgt mit dem Java-Schlüsselwort `implements`

```
MyThread implements Runnable
```

bedeutet: Die Klasse MyThread enthält mindestens eine Implementierung aller in der Schnittstelle Runnable aufgelisteten Methoden. Die folgende Tabelle zeigt einige wichtige Implementierungsvorgänge:

Implementierungsvorgang	Beispiel-Klasse	Zweck des Implementierungsvorgangs
`implements WindowListener`	bsp03040.MyFrame_WindowAdapter	Ereignisbehandlung für Fenster-Ereignisse
`implements Printable`	bsp03072.MyPrint	Programmierung eines druckbaren Dokumentes
`implements Runnable`	bsp03065.DateiLeseVorgang	Programmierung von Verarbeitung, die von einem Thread ausgeführt werden kann und die in einer anderen als von Object abgeleiteten Klasse angeordnet werden soll.

Weitere wichtige Gebiete der Programmierung, die die Implementierung von Schnittstellen verwenden, sind

- Datenbankprogrammierung mit JDBC
- Client-Server Programmierung mit Java RMI

2.9.2 Vererben und Überschreiben

2.9.2.1 Das Prinzip der Vererbung

Das Prinzip der Vererbung wird eingesetzt, wenn mathematisch gesehen eine Teilmengenbeziehung vorliegt:

Das Prinzip der **Vererbung** wird zwischen zwei Klassen eingesetzt, wenn die Objekte der einen Klasse eine "ist ein"-Beziehung zu den Objekten der anderen Klasse haben: Wenn gilt:

- objekt1 ist ein Objekt der Klasse Klasse1
- objekt2 ist ein Objekt der Klasse Klasse2
- objekt2 **ist ein**e besondere Ausprägung von objekt1,

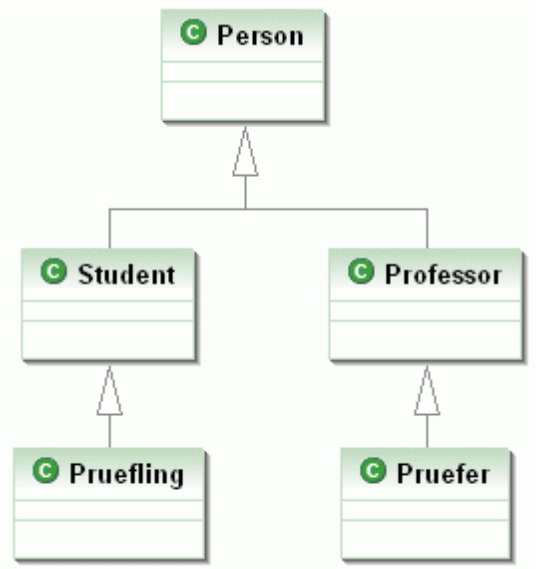
so kann dies am besten dadurch informationstechnisch abgebildet werden, **dass die Klasse Klasse2 alle Methoden und Attribute von Klasse1 erbt**. Mengentheoretisch ausgedrückt gilt dann nämlich dann;

- Die Menge der Objekte der Klasse Klasse2 **ist eine** Teilmenge der Menge der Objekte der Klasse Klasse1.
- Ein Objekt der Klasse Klasse2 hat alle Eigenschaften (Attribute) und Funktionalitäten (Methoden) der Objekte der Klasse Klasse1.
- Ein Objekt der Klasse Klasse2 **ist ein** Spezialfall der Objekte der Klasse Klasse1.

Beispiel: Gegeben seien die Klassen Person, Student, Prüfling, Professor, Prüfer:

Ein Student ist eine Person => Student erbt von Person
Ein Prüfling ist ein Student => Prüfling erbt von Student
Ein Professor ist eine Person => Professor erbt von Person
Ein Prüfer ist ein Professor => Prüfer erbt von Professor

Somit kann man folgende **Vererbungshierarchie** aufbauen (Beispiel jpr02111):



2.9.2.2 Was bedeutet das Prinzip der Vererbung?

Vererbung mit Hilfe des Java-Schlüsselwortes **`extends` (erweitert)** programmiert.

Erbt eine Klasse Klasse2 alle Attribute und Methoden von Klasse1, so bildet die Menge der protected und public Attribute und Methoden von Klasse2 eine *Obermenge* der Attribute und Methoden von Klasse1. Man sagt auch:

- Klasse2 **erweitert** Klasse1
- Klasse2 **ist abgeleitet von** Klasse1

private Attribute und private Methoden werden nicht vererbt. Vererbt werden sowohl statische als auch nicht statische Elemente der vererbenden Klasse.

- Die vererbende Klasse nennt man auch **Superklasse**, **Basisklasse** bzw. **übergeordnete Klasse**
- Die erbende Klasse nennt man auch **Subklasse**, **Unterklasse**, **abgeleitete Klasse** bzw. **untergeordnete Klasse** Bei der Java-Programmierung kann eine Klasse stets nur von einer Superklasse abgeleitet werden. Eine sogenannte

> Mehrfachvererbung ist nicht möglich. Stattdessen werden bei der Java-Programmierung Schnittstellen verwendet.

Für das obige Beispiel Person und die davon abgeleiteten Klassen bedeutet dies:

Student erbt von Person:	`public class Student extends Person`
Prüfling erbt von Student:	`public class Pruefling extends Student`
Professor erbt von Person:	`public class Professor extends Person`
Prüfer erbt von Professor:	`public class Pruefer extends Professor`

2.9.2.3 Umsetzung von Vererbung beim Datenbankentwurf

In diesem Abschnitt wird erklärt, wie Klassen, die in einer Vererbungshierarchie miteinander verbunden sind, in einer Datenbank gespeichert werden. Der gesamte Ablauf ausgehend vom Klassendiagramm bis zum Relationenmodell kann hier natürlich nicht vollständig beschrieben werden. Vielmehr wird hier der Ablauf anhand des Beispiels aus 2.9.2.1 skizziert. Eine genauere Erklärung des Datenbankentwurfs findet man in dem Skript **Eikelberg, Relationale Datenbanken, Entwurf und Implementierung mit SQL**

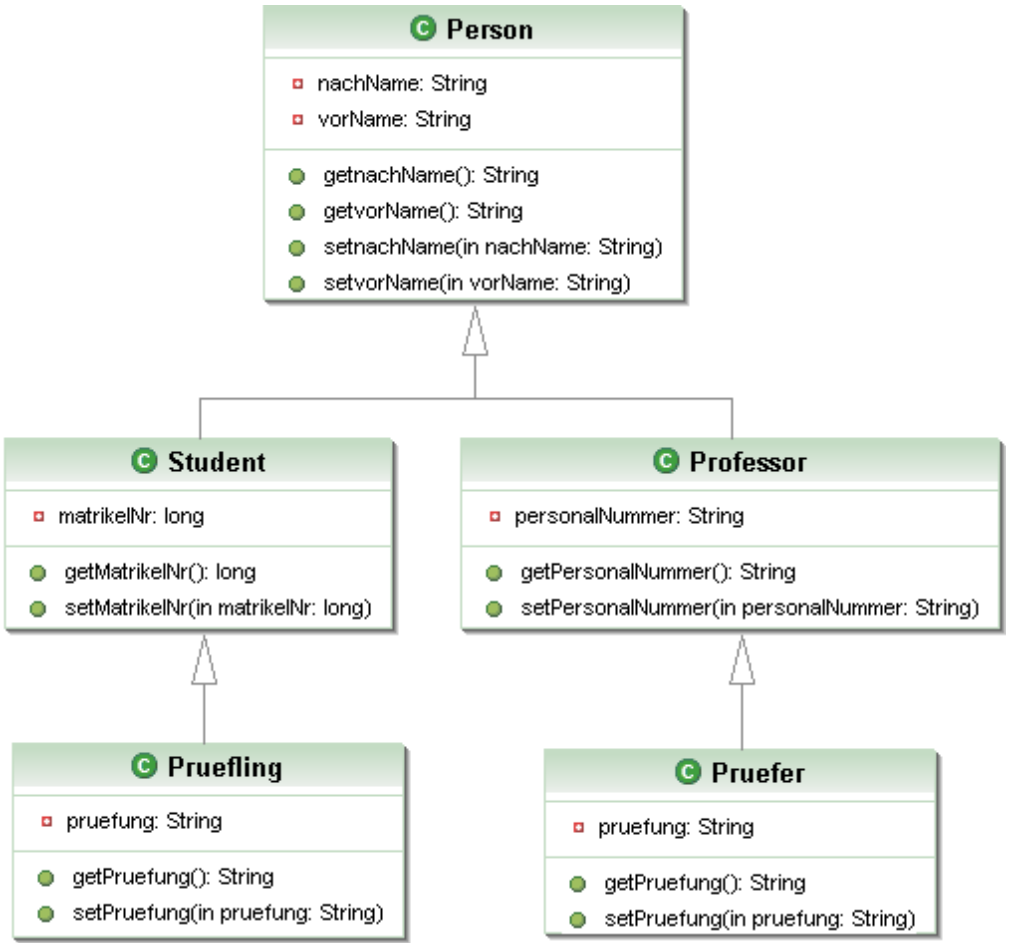
Die Übersetzung des Klassendiagramms in eine Relationenmodell geschieht in folgenden Schritten:

1. Aus dem Klassendiagramm wird wie folgt ein **Entity-Relationship-Diagramm** erstellt:
    - Jede Klasse ergibt eine Entität (einen in der Datenbank zu speichernden Typ von Objekten).
    - Jede Vererbungsbeziehung ergibt eine Zuordnung vom Typ "1 - c", d.h.
        - jeder Entität zur erbenden Klasse ist genau eine (1) Entität zur vererbenden Klasse zugeordnet und
        - jeder Entität zur vererbenden Klasse ist keine oder eine (c) Entität der erbenden Klasse zugeordnet.
    - Weitere Zuordnungen zwischen den Entitäten müssen gefunden und eingezeichnet werden.
2. Aus dem Entity-Relationship-Diagramm wird anschließend (mit Hilfe der Methode der Datenbanknormalisierung) ein **Relationenmodell** erarbeitet:
    - Jede Entitität ergibt eine Tabelle im Relationenmodell. Im Rahmen der Datenbanknormalisierung kann es sein, dass diese Tabelle zur Vermeidung von Redundanzen in mehrere Tabellen zerlegt wird.
    - Die aus Vererbungshierarchien entstandenen "1-c"-Beziehung ist hierarchisch und ergibt eine Fremdschlüsselverknüpfung im Relationenmodell: Der Primärschlüssel der Tabelle zur vererbenden Klasse wird Fremdschlüssel in der Tabelle bzw. den Tabellen zur erbenden Klasse
    - Weitere zwischen den Entitäten gefundene Zuordnungen müssen mit Hilfe der Transformationsregeln in hierarchische Beziehungen umgeformt werden. Diese hierarchischen Beziehungen ergeben dann Fremdschlüsselverknüpfungen im Relationenmodell

Zur Illustration des oben beschriebenen Vorgangs wird das Beispiel aus 2.9.2.1 geringfügig erweitert. Anschließend wird für das erweiterte Beispiel ein Entity-Relationship-Diagramm und ein Relationenmodell abgeleitet.

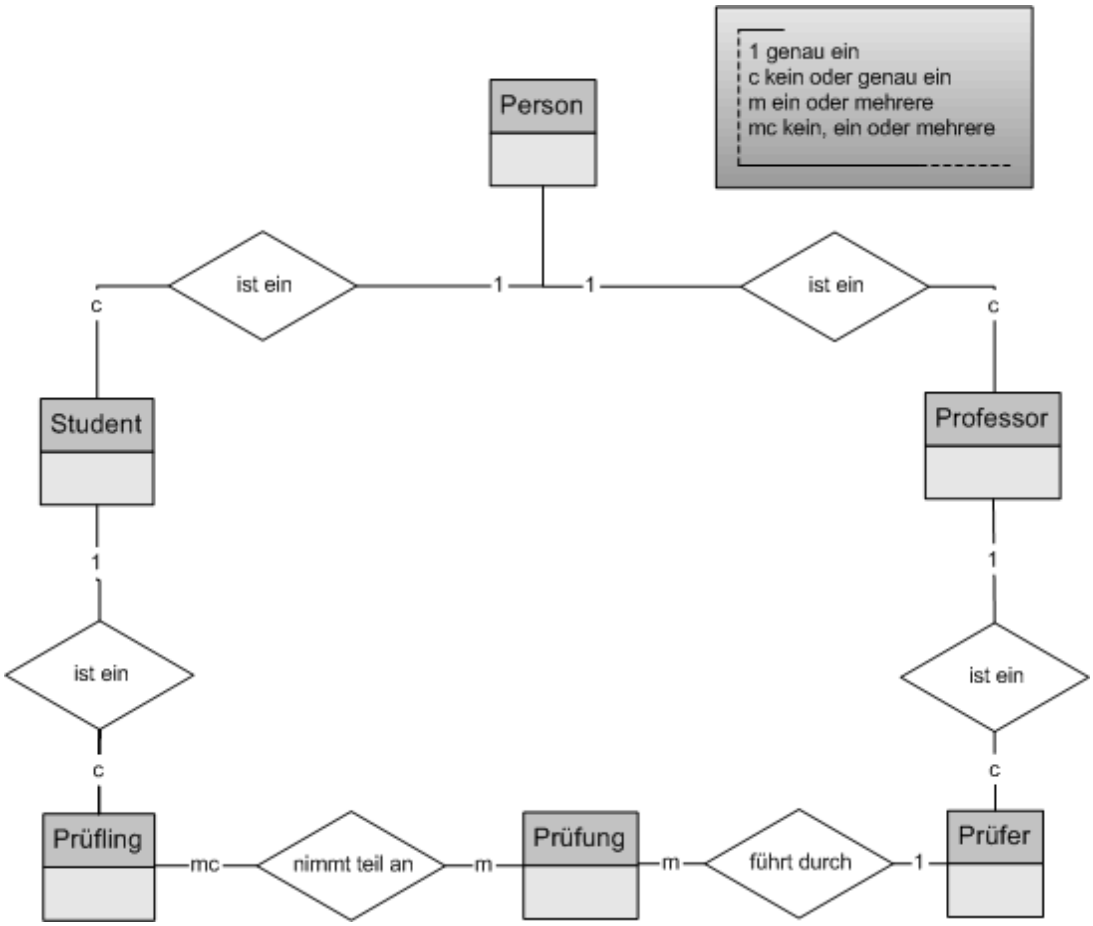
Beispiel - Diagramm 1: Klassendiagramm mit Vererbung, Attributen, get- und set-Methoden

Das Beispiel beginnt mit dem Klassendiagramm aus 2.9.2.1, jedoch um Attribute sowie get- und set-Methoden erweitert:

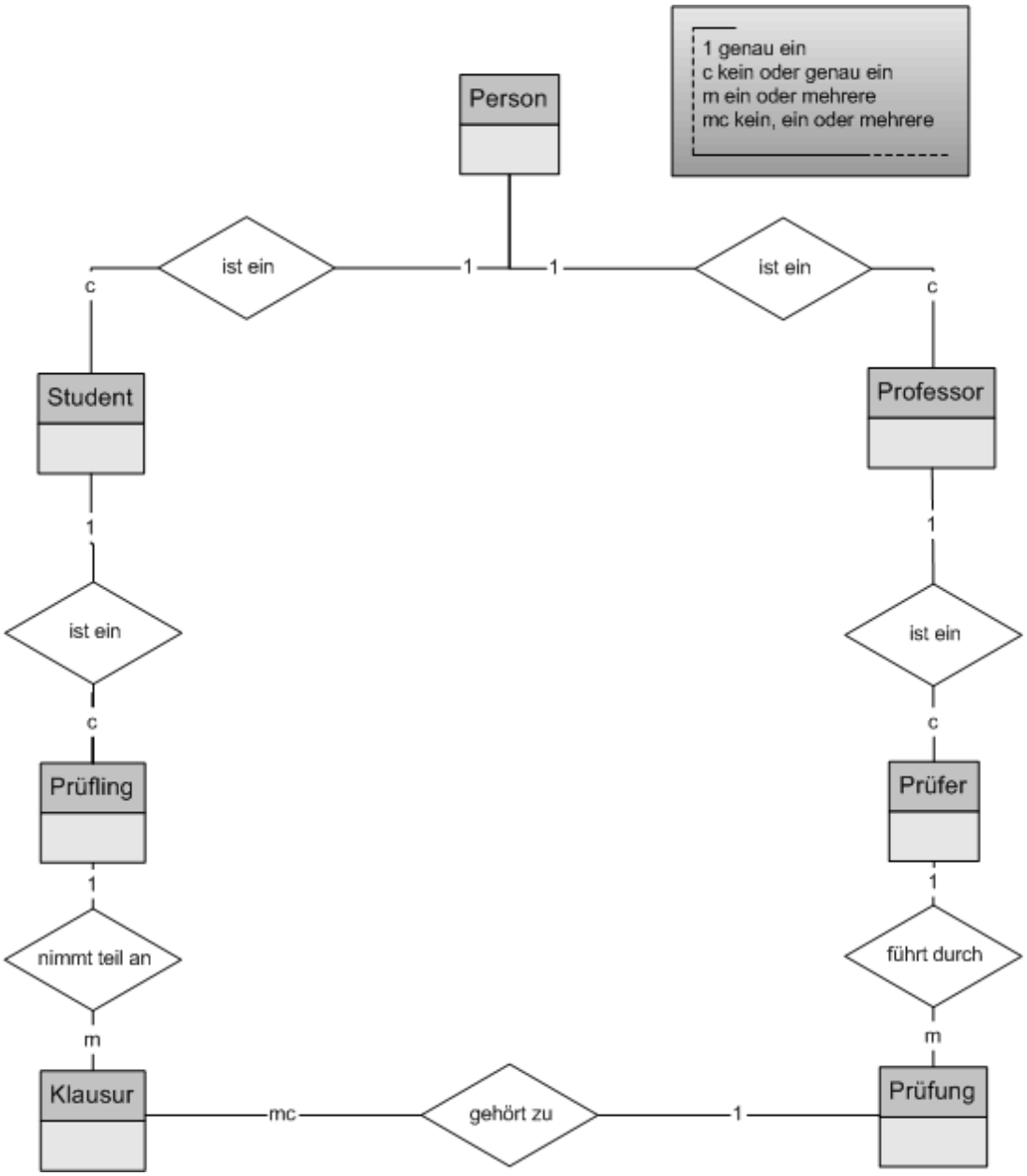


Beispiel - Diagramm 2: Entity-Relationship-Diagramm

Beim Übergang zum Entity-Relationship-Diagramm ergeben die Klassen die Entitäten. Die Vererbungsbeziehungen ergeben "1-c"-Beziehungen zwischen den Entitäten. Zusätzlich kann man im vorliegenden Beispiel eine weitere Entität sowie weitere Beziehungen zwischen den Entitäten finden:



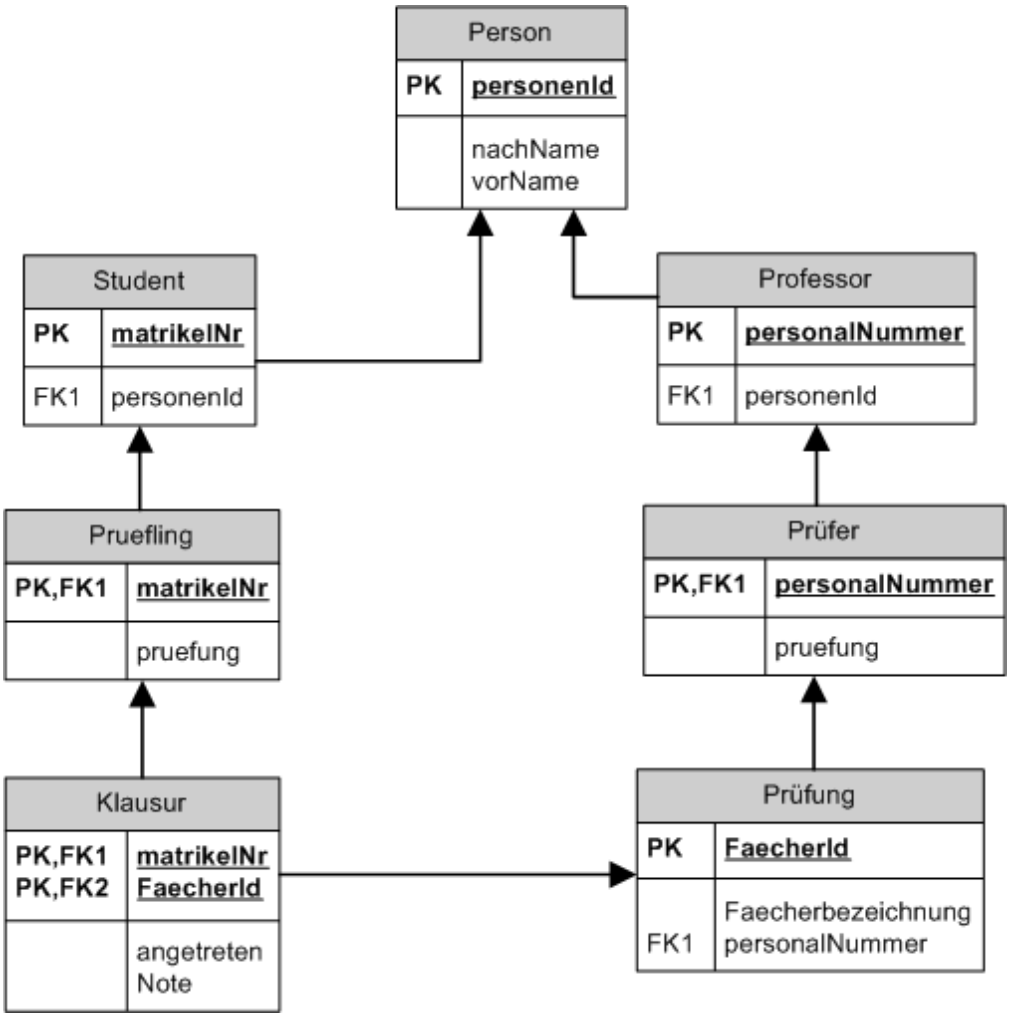
Die im obigen Entity-Relationship-Diagramm enthaltene "mc - m"-Beziehung muss vor der Umformung in ein Relationenmodell in eine "1 - m"-Beziehung sowie eine "mc - 1"-Beziehung umgeformt werden. Dies ergibt eine neue Entität "Klausur" sowie das nachfolgende Modell:



Hinweis: Man beachte, dass im Entity-Relationship-Diagramm gegenüber dem Klassendiagramm jetzt zwei weitere Entitäten erscheinen. Diese Entitäten müssen im Programm ebenfalls durch Klassen abgebildet werden und müssen entsprechend in das Klassendiagramm eingefügt werden. *Allerdings besteht zwischen den Klassen "Klausur" bzw. "Prüfung" sowie den im Entity-Relationship-Diagramm übergeordneten Klassen keine Vererbungsbeziehung - es gilt zum Beispiel* ***nicht:*** *"eine Klausur ist ein Prüfling" bzw "eine Prüfung ist ein Prüfer". Eine Beziehung im Entity-Relationship-Diagramm ist somit eine allgemeinere Beziehung als eine Vererbungsbeziehung.*

Beispiel - Diagramm 3: Relationenmodell

Im vorliegenden Fall ist die Transformation des Entity-Relationship-Diagramms in ein Relationenmodell sehr einfach: Die Entitäten ergeben genau die Relationen (Tabellen), da keine der vorhandenen Tabellen im Rahmen der Datenbanknormalisierung aufgespalten werden muss. Die Beziehungen zwischen den Entitäten sind bereits alle in hierarchische Beziehungen aufgelöst worden und ergeben damit genau die Fremdschlüsselbeziehungen (PK = primary key = Primärschlüssel, FK = foreign key = Fremdschlüssel):



Mit der Erstellung des Relationenmodells ist der Datenbankentwurf abgeschlossen. Alle Vererbungsbeziehungen im Klassendiagramm sind in Fremdschlüsselverknüpfungen des Relationenmodells umgeformt worden.

2.9.2.4 Kapselung: Schutz der Attribute mit Get- und Set-Methoden

Das Prinzip der **Kapselung** besagt, dass alle Attribute vor unbefugter Verwendung oder Änderung geschützt werden sollten. Um dies zu erreichen, gelten bei der Vererbung folgende Regeln bzw. Empfehlungen:

- private Methoden und private Attribute werden nicht vererbt.
- protected und public Attribute sollten nur dann codiert werden, wenn sie mit dem Schlüsselwort `final` einen Schreibschutz erhalten und auf diese Weise nicht veränderbar codiert sind.
- Der Zugriff auf nicht `final` gesetzte Attribute von außerhalb der die Attribute enthaltenden Klassen sollte nur mit get- und set-Methoden erfolgen. Hierzu sollte folgende Regeln:
  - Nicht `final` Attribute sollten grundsätzlich **private** gesetzt werden.
  - Für die Veränderung des Wertes eines nicht `final` Attributs sollte eine **set-Methode** bereitgestellt werden, die den neuen Attributwert annimmt, ggf. überprüft und im Attribut speichert.
  - Für das Lesen des Wertes eines nicht `final` Attributes sollte eine **get-Methode** bereitgestellt werden, die den Wert des Attributs zurückgibt.

  Die get- und set-Methoden sollten je nach Situation dann protected oder public gesetzt werden.

Das nachfolgende Beispiel jpr02112 zeigt die Umsetzung der obigen Prinzipien auf das Beispiel "Person" (jpr02111) sowie die von Person abgeleiteten Klassen. In der obersten Klasse werden alle Eigenschaften (Attribute) codiert, die für alle Klassen der Vererbungshierarchie gemeinsam vorhanden sein sollten - also die Variablen für den Nachnamen, den Vorname und das Geburtsdatum:

```
package bsp02112;

public class Person
{
/*
 * Attribute
 */
// Personendaten - protected und public Variablen werden vererbt, private Variablen
```

```
// nicht. Es wird empfohlen, keine protected bzw. public Variablen zu deklarieren.
// So wird erzwungen, dass auf die Variablen der vererbenden Klasse stets nur über
// die get- und set-Methoden zugegriffen wird.
 private String vorName = "";
 private String nachName = "";
// Es gibt bessere Wege, ein Geburtsdatum zu speichern (vgl. jpr02113). Diese sind
// allerdings aufwändig und werden daher für dieses Beispiel nicht gewählt
 private String geburtsDatum = "0000-00-00";

/*
 * Konstruktoren
 */
 public Person(String vorName, String nachName, String geburtsDatum)
 {
   this.vorName = vorName;
   this.nachName = nachName;
   this.geburtsDatum = geburtsDatum;
 }

/*
 * get-Methoden
 */
 public String getNachName()
 {
   return this.nachName;
 }

 public String getVorName()
 {
   return this.vorName;
 }

 public String getGeburtsDatum()
 {
   return this.geburtsDatum.toString();
 }

/*
 * set-Methoden
 */
 public void setNachName(String nachName)
 {
   this.nachName = nachName;
 }

 public void setVorName(String vorName)
 {
   this.vorName = vorName;
 }

 public void setGeburtsDatum(String geburtsDatum)
 {
   this.geburtsDatum = geburtsDatum;
 }
}
```

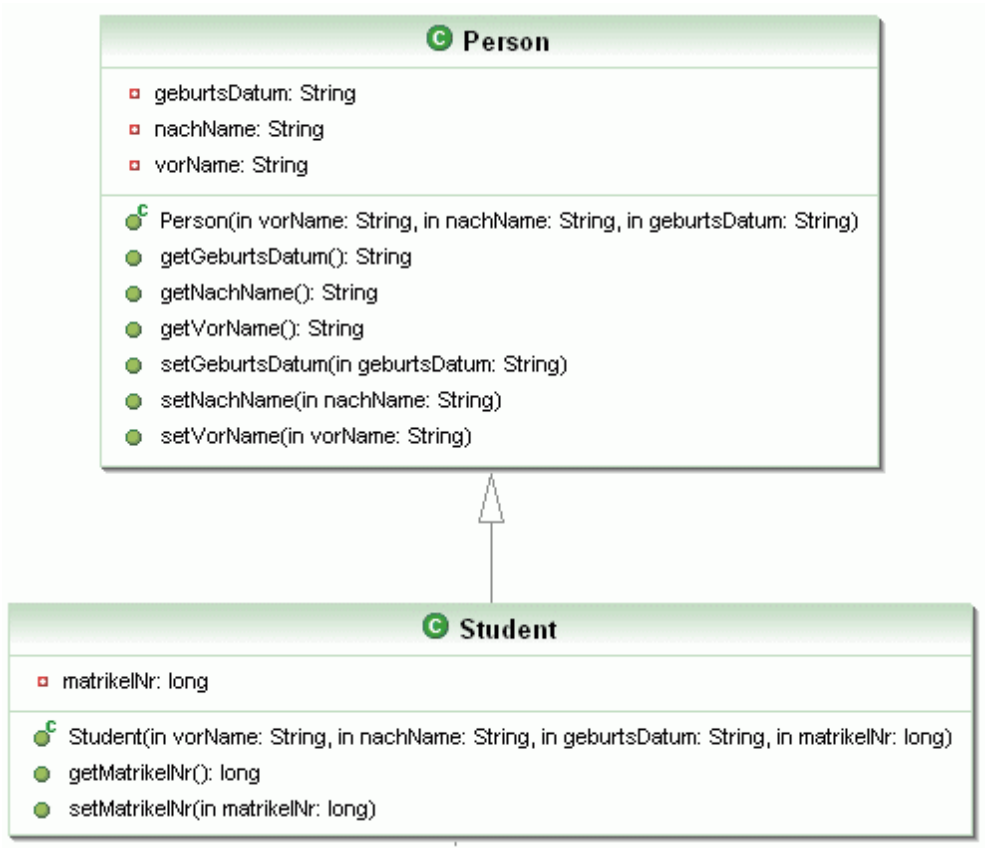
Von der obigen Klasse Person ist z. Bsp. die folgende Klasse Student abgeleitet:

```
package bsp02112;

public class Student extends Person
{
/*
 * Attribute
 */
  private long matrikelNr = 0L;

/*
 * Konstruktoren
 */
  public Student(String vorName, String nachName, String geburtsDatum, long matrikelNr)
  {
// Der Super-Konstruktor muss stets als erste Anweisung aufgerufen werden!
    super(vorName, nachName, geburtsDatum);
```

```
    this.matrikelNr = matrikelNr;
  }

/*
 * get-Methoden
 */
  public long getMatrikelNr()
  {
    return this.matrikelNr;
  }

/*
 * set-Methoden
 */
  public void setMatrikelNr(long matrikelNr)
  {
    this.matrikelNr = matrikelNr;
  }
}
```

Mit den im obigen Beispiel enthaltenen Konstruktoren sowie dem Schlüsselwort `super` befasst sich der nächste Abschnitt. Im UML-Klassendiagramm werden die beiden obigen Klassen wie folgt dargestellt:

- runde (grüne) Punkte zeigen die public get- und set-Methoden
- rote) Quadrate mit einem weißen Pluszeichen die private Attribute
- die Konstruktoren werden mit dem Buchstaben C gekennzeichnet

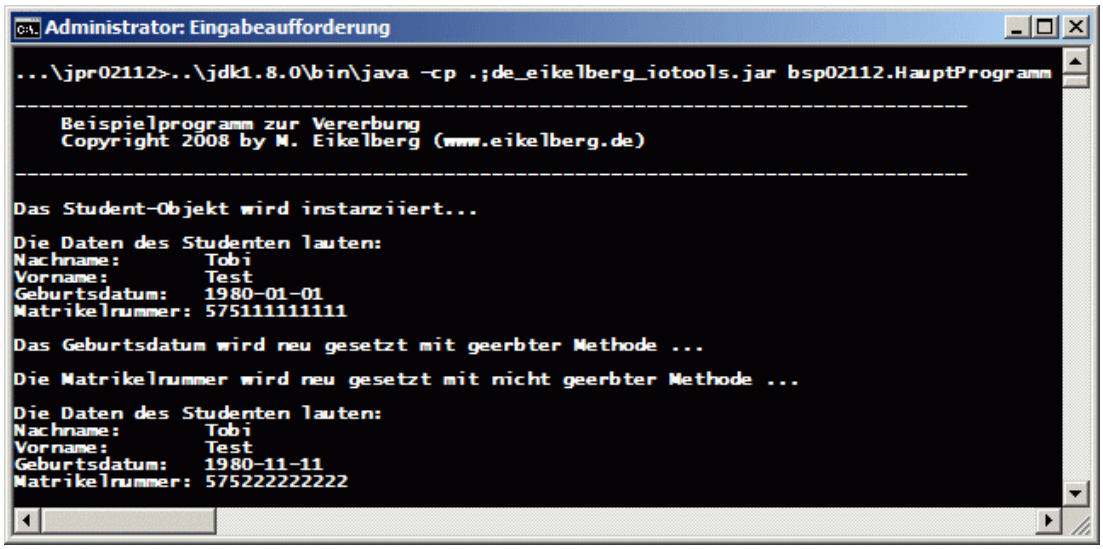


Das nachfolgende Programm zeigt, dass über die get- und set-Methoden der Klasse Person nach Instanziierung eines Objektes der Klasse Student auf die Instanzvariablen der Klasse Person zugegriffen werden kann:

```
package bsp02112;

import de.eikelberg.iotools.*;

public class HauptProgramm
{

  public static void main(String[] args)
  {
    ScreenOut.kStartScreenOut(
      "Beispielprogramm zur Vererbung",
      "M. Eikelberg (www.eikelberg.de)","2008");
```

```
// Instanziieren eines Objektes der Subklasse Student
    Student stud;
    System.out.println("\nDas Student-Objekt wird instanziiert...");
    stud = new Student("Test","Tobi","1980-01-01",575111111111L);

//  Lesen der Daten des Student-Objektes mit geerbten und
//  nicht geerbten Methoden
     System.out.println("\nDie Daten des Studenten lauten:");
     System.out.println("Nachname:       "+stud.getNachName());     // geerbte get-Methode
     System.out.println("Vorname:        "+stud.getVorName());      // geerbte get-Methode
     System.out.println("Geburtsdatum:   "+stud.getGeburtsDatum());// geerbte get-Methode
     System.out.println("Matrikelnummer: "+stud.getMatrikelNr());  // nicht geerbte get-Methode

// Änderung des Geburtsdatums (geerbte Methode)
    System.out.println("\nDas Geburtsdatum wird neu gesetzt mit geerbter Methode ...");
    stud.setGeburtsDatum("1980-11-11");                            // geerbte set-Methode

// Änderung der Matrikelnummer (nicht geerbte Methode)
    System.out.println("\nDie Matrikelnummer wird neu gesetzt mit nicht geerbter Methode ...");
    stud.setMatrikelNr(575222222222L);                             // nicht geerbte set-Methode

// Lesen der Daten des Student-Objektes mit geerbten und
// nicht geerbten Methoden
    System.out.println("\nDie Daten des Studenten lauten:");
    System.out.println("Nachname:       "+stud.getNachName());     // geerbte get-Methode
    System.out.println("Vorname:        "+stud.getVorName());      // geerbte get-Methode
    System.out.println("Geburtsdatum:   "+stud.getGeburtsDatum()); // geerbte get-Methode
    System.out.println("Matrikelnummer: "+stud.getMatrikelNr());   // nicht geerbte get-Methode
  }
}
```

Der nachfolgende Bildschirmausdruck zeigt, dass die Verarbeitung erfolgreich durchgeführt werden kann:



2.9.2.5 Konstruktoren und das Schlüsselwort super

In den obigen Beispielklassen bsp02112.Person und bsp02112.Student sind jeweils ein Konstruktor enthalten. Für Konstruktoren gelten bei Vererbung folgende Regeln:

- Da Konstruktoren jeweils den Namen der zugehörigen Klasse haben müssen, werden Konstruktoren nicht vererbt. Man kann allerdings auf die Konstruktoren der vererbenden Klasse mit dem Methodennamen super(...) zugreifen (s.u.).
- In einem Konstruktor einer untergeordneten Klasse kann ein Konstruktor der übergeorndeten Klasse aufgerufen werden, in dem vor den Namen des Konstruktors der übergeordneten Klasse das Schlüsselwort `super` gesetzt wird. Das Schlüsselwort `super` steht also für eine Konstruktor-Methode. Da bei der Java-Programmierung eine Klasse stets nur von maximal einer anderen Klasse abgeleitet werden kann und da sich die Konstruktoren durch ihre Signatur unterscheiden, lautet der Aufruf eines übergeordneten Konstruktors wie folgt:

  `super(...`*`Parameterliste`*`...);`

  Daraufhin wird der Konstruktor der übergeordneten Klasse aufgerufen, der eine passende Parameterliste hat.
- Der Aufruf eines Konstruktors der übergeordneten Klasse mit super darf nur als erste Anweisung in einem Konstruktor der abgeleiten Klasse erfolgen.
- Bei Aufruf eines Konstruktors der erbenden Klasse wird vor Ausführung der Anweisungen des Konstruktors standardmäßig zunächst

der parameterlose Konstruktor der übergeordneten Klasse ausgeführt.
- In jedem Konstruktor der abgeleiteten Klasse kann jeder Konstruktor der übergeordneten Klasse aufgerufen werden, der nicht private ist. Bei explizitem Aufruf eines super-Konstruktors entfällt der standardmäßige Aufruf des parameterlosen super-Konstruktors.

Im Beispiel jpr02112 sind zwei Konstruktoren mit Parametern enthalten. Der Konstruktor der übergeordneten Klasse Person lautet:

```
public Person(String vorName, String nachName, String geburtsDatum)
{
  this.vorName = vorName;
  this.nachName = nachName;
  this.geburtsDatum = geburtsDatum;
}
```

In dem obigen Konstruktor werden die dem Konstruktor übergebenen Daten in die Instanzvariablen des Objektes umgespeichert. Der nachfolgende Konstruktor der abgeleiteten Klasse Student

- ruft zunächst den "Superkonstruktor" der Klasse Person auf, um die Daten in den Parametern vorName, nachName und geburtsDatum umzuspeichern, denn die zugehörigen Variablen sind Instanzvariablen der Klasse Person, können nicht direkt angesprochen werden und daher alternativ nur über set-Methoden gefüllt werden.
- Die Variable matrikelNummer ist Instanzvariable der Klasse Student und kann anschließend direkt gefüllt werden.

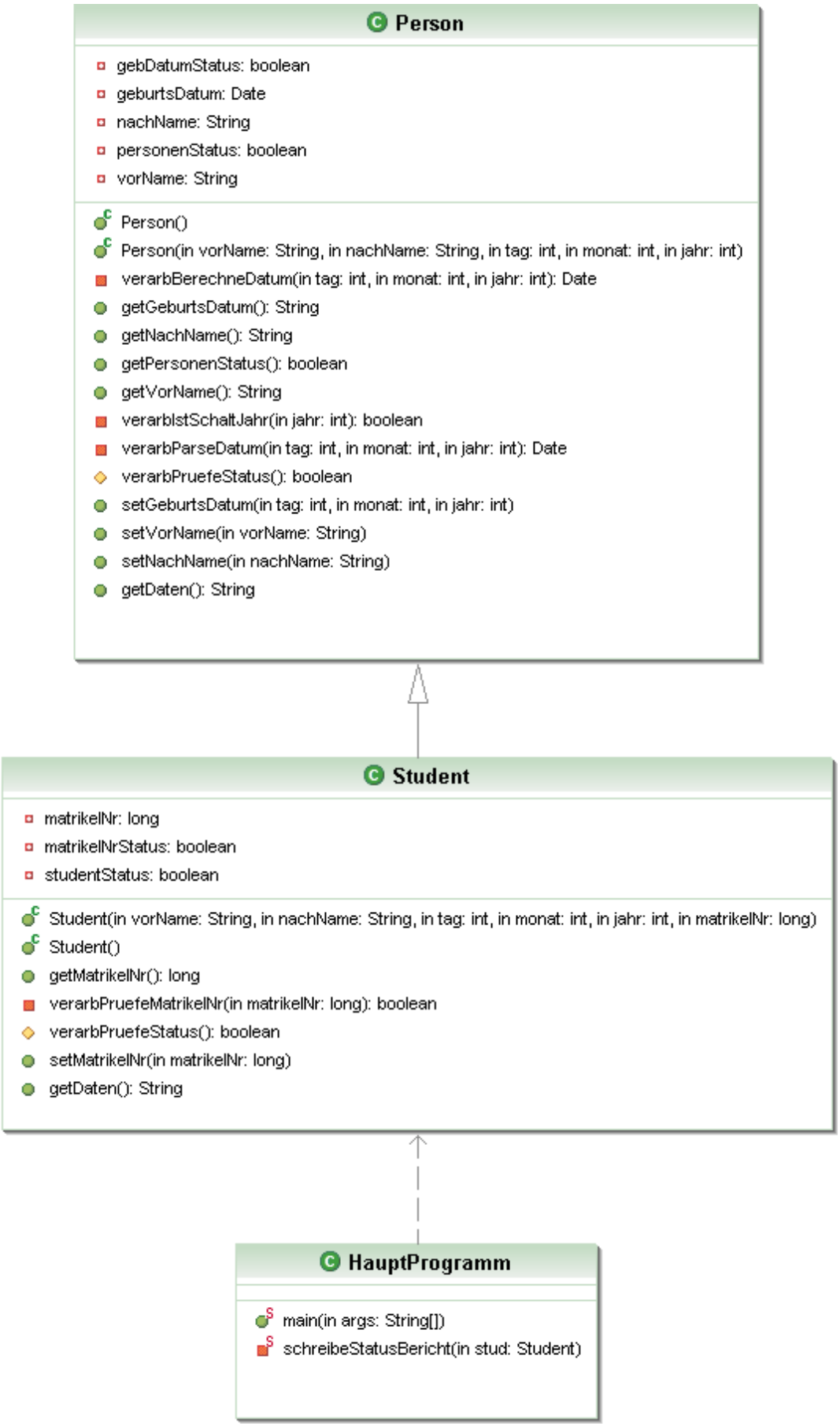
Da der Aufruf eines Superkonstruktors stets als erste Anweisung in einem Konstruktor geschehen muss, geschieht im nachfolgenden Programmcode das Umspeichern der Matrikelnummer erst als zweite Anweisung:

```
  public Student(String vorName, String nachName, String geburtsDatum, long matrikelNr)
  {
// Der Super-Konstruktor muss stets als erste Anweisung aufgerufen werden!
    super(vorName, nachName, geburtsDatum);
    this.matrikelNr = matrikelNr;
  }
```

Beispiel jpr02113 erweitert und verbessert wie folgt das Beispiel jpr02112:

- Das Verhalten von Konstruktoren bei Vererbung wird besser sichtbar (s.u.).
- Die Speicherung des Geburtsdatums wurde verbessert - statt eines Strings wird ein Objekt der Klasse java.util.Date verwendet.
- Das Prinzip "Überschreiben" wurde verwendet, um den Programmcode einfacher zu gestalten (siehe 2.9.2.6)

Um das Beispiel übersichtlich zu halten, werden im Projekt jpr02113 nur noch die Klassen Person und Student betrachtet. Insgesamt enthalten die Klassen des Projektes jpr02113 folgende Attribute und Methoden:



Man sieht folgende Anwendung der oben beschriebenen Prinzipien:

- Alle Attribute sind private gesetzt - markiert mit (roten) Quadraten mit weißem Plus.
- Die Klassen Person und Student verfügen beide über einen parameterlosen sowie einen parameterbehafteten Konstruktor.
- Die Klassen Person und Student verfügen beide über public get- und set-Methoden (beginnen alle mit "set" bzw. "get")
- Die Klassen Person und Student verfügen beide über private bzw. protected verarbeitende Methoden (beginnen alle mit "verarb")

Im Beispiel jpr02113 sind somit folgende Konstruktoren enthalten:

- `public Person()`
- `public Person(String vorName, String nachName, int tag, int monat, int jahr)`
- `public Student()`
- `public Student(String vorName, String nachName, int tag, int monat, int jahr, long matrikelNr)`

Jeder dieser Konstruktoren protokolliert seinen Aufruf mit einer Konsolmeldung "Konstruktor (...) der Klasse ... wird verarbeitet!". In der Klasse bsp02113.HauptProgramm werden die Konstruktoren der Klasse Student wie folgt aufgerufen:

```
// Objekt der Klasse Student mit parameterlosem Konstruktor anlegen
    System.out.println(
        "\nAnlegen eines Objektes der Klasse Student mit Konstruktor (parameterlos) ...");
    Student stud1 = new Student();

...

// Objekt der Klasse Student mit Daten anlegen
    System.out.println(
        "\nAnlegen eines Objektes der Klasse Student mit Konstruktor (mit Parametern) ...");
    Student stud2 = new Student("Test", "Tobias", 29, 2, 2001, 575000111111L);
```

Die beiden Konstruktoren enthalten folgenden Programmcode:

```
  public Student()
  {
    System.out.println("Konstruktor (parameterlos) der Klasse Student wird verarbeitet!");
  }

  public Student(String vorName, String nachName, int tag, int monat, int jahr, long
matrikelNr)
  {
// Der Super-Konstruktor muss stets als erste Anweisung aufgerufen werden!
    super(vorName, nachName, tag, monat,jahr);
    System.out.println("Konstruktor (mit Parametern) der Klasse Student wird verarbeitet!");
...
    this.matrikelNr = matrikelNr;
...
  }
```

Man sieht:

- Im parameterlosen Konstruktor der Klasse Student *erfolgt kein Aufruf* des parameterlosen Konstruktors der vererbenden Klasse, da dieser Aufruf stets automatisch erfolgt.
- Im parameterbehafteten Konstruktor der Klasse Student *erfolgt ein Aufruf* des parameterbehafteten Konstruktors der vererbenden Klasse, da in einem parameterbehafteten Konstruktor kein automatischer Aufruf eines Konstruktors der vererbenden Klasse erfolgt. Der Aufruf des Superkonstruktors erfolgt als erster Verarbeitungsschritt im Konstruktor der vererbenden Klasse.

In der Klasse Person enthalten die Konstruktoren folgenden Programmcode:

```
  public Person()
  {
    System.out.println("Konstruktor (parameterlos) der Klasse Person wird verarbeitet!");
    this.nachName = "";
    this.vorName = "";
    this.geburtsDatum = new Date(0);
...
  }

  public Person(String vorName, String nachName, int tag, int monat, int jahr)
  {
    System.out.println("Konstruktor (mit Parametern) der Klasse Person wird verarbeitet!");
    this.vorName = vorName;
    this.nachName = nachName;
    this.geburtsDatum = this.verarbBerechneDatum(tag, monat, jahr);
...
  }
```

Der Programmlauf zeigt jetzt, dass sowohl für den parameterlosen als auch für den parameterbehafteten Konstruktor eine Verarbeitung eines Konstruktors der vererbenden Klasse erfolgt:

```
Administrator: Eingabeaufforderung
...\jpr02113>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02113.HauptProgramm
--------------------------------------------------------------------------------
    Beispielprogramm zu Vererbung, Ueberschreiben und Zeiger super
    Copyright 2008 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Anlegen eines Objektes der Klasse Student mit Konstruktor (parameterlos) ...
Konstruktor (parameterlos) der Klasse Person wird verarbeitet!
Konstruktor (parameterlos) der Klasse Student wird verarbeitet!

Das Student-Objekt ist NICHT in korrektem Zustand.

Die Personendaten lauten:
-> Nachname:
-> Vorname:
-> Geburtsdatum: Thu Jan 01 01:00:00 CET 1970
-> Matrikelnummer: 0
-> Status: nicht ok

Anlegen eines Objektes der Klasse Student mit Konstruktor (mit Parametern) ...
Konstruktor (mit Parametern) der Klasse Person wird verarbeitet!
Konstruktor (mit Parametern) der Klasse Student wird verarbeitet!

Das Student-Objekt ist NICHT in korrektem Zustand.

Die Personendaten lauten:
-> Nachname: Tobias
-> Vorname: Test
-> Geburtsdatum: Thu Jan 01 01:00:00 CET 1970
-> Matrikelnummer: 575000111111
-> Status: nicht ok

Aendern des Geburtsdatums fuer das zuletzt angelegte Objekt ...

Das Student-Objekt ist in korrektem Zustand.

Die Personendaten lauten:
-> Nachname: Tobias
-> Vorname: Test
-> Geburtsdatum: Tue Feb 29 00:00:00 CET 2000
-> Matrikelnummer: 575000111111
-> Status: ok
```

Die Meldung des Programms im obigen Bildschirmausdruck, dass nach dem zweiten Anlegen eines Person-Objektes mit dem parameterbehafteten Konstruktor das Student-Objekt nicht im korrekten Zustand ist, ist übrigens dadurch verursacht, dass beim Aufruf des Konstruktors das fehlerhafte Geburtsdatum 29.2.2001 angegeben wurde. Dieses wurde nicht gespeichert und stattdessen bleibt der bei der Initialisierung des Objektes gesetzte Standardwert 1.1.1970 im Person-Objekt erhalten.

2.9.2.6 Überschreiben und das Schlüsselwort super

Beispiel jpr02113 zeigt neben der Verarbeitung von Konstruktoren auch die Anwendung des Prinzips "Überschreiben":

Das Prinzip **Überschreiben** besagt, dass in einer untergeordneten Klasse eine Methode der übergeordneten Klasse durch eine Methode

- mit gleicher Signatur (d.h. gleiche Parameter und gleicher Rückgabewert) und
- mindestens gleicher Sichtbarkeit

ersetzt (überschrieben) werden kann. Voraussetzung hierfür ist,

- dass die überschreibende Methode mindestens die gleiche Sichtbarkeit erhält wie die überschriebene Methode

Eine private Methode einer übergeordneten Klasse ist in der abgeleiteten Klasse nicht sichtbar, kann aber auch nicht überschrieben werden.

- Man kann auf die überschriebene Methode durch *Voransetzen* des Schlüsselwortes `super` vor den Methodennamen zugegreifen.
- Dazu muss die überschriebene Methode mindestens die Sichtbarkeit protected haben, denn ansonsten ist sie in der abgeleiteten Klasse nicht sichtbar.
- Da eine überschreibende Methode mindestens die gleiche Sichtbarkeit haben muss wie die überschriebene Methode, muss eine überschreibende Methode, in der das Schlüsselwort `super` verwendet wird, mindestens Sichtbarkeit protected haben.

Das Prinzip Überschreiben wird u.a. dann eingesetzt, wenn in verschiedenen, über Vererbung verbundenen Klassen Methoden mit fachlich äquivalentem Ergebnis codiert werden sollen, die jedoch jeweils eine individuelle und an die jeweilige Klasse angepasste interne Verarbeitung beinhalten müssen (siehe Beispiel unten). Eine weitere Anwendung besteht darin, dass weitestgehend fertig ausprogrammierte Klassen als vererbende Klassen gewählt werden und dass in der erbenden Klasse lediglich eine oder wenige Methoden durch Überschreiben ersetzt werden (vgl. 2.9.2.7)

Im Beispiel jpr02113 überschreibt in der Klasse Student die Methode `verarbPruefeStatus()` die gleichnamige Methode der Klasse Person. Zunächst hier der Quellcode der Klasse Person:

```
package bsp02113;
```

```
import java.util.Date;
...

public class Person
{
/*
 * Attribute
 */
// Personendaten - Protected Variablen werden vererbt, die private
// Variablen nicht. Es wird empfohlen, keine protected Variablen zu deklarieren,
// so wird erzwungen, dass auf die Variablen der vererbenden Klasse stets nur über
// die get- und set-Methoden zugegriffen wird.
  private String vorName = "";
  private String nachName = "";
  ...
// Der personenStatus ist true, wenn ein gültige Personendaten vorliegen
  private boolean personenStatus = true;
// Der gebDatumStatus ist true, wenn ein gültiges Geburtsdatum vorliegt
  private boolean gebDatumStatus = true;

...

// Methode zur Überprüfung, ob gültige Daten im Objekt gespeichert sind
// Methode muss protected sein, da sie ansonsten in einer
// erbenden Klasse nicht mit super aufgerufen werden kann
  protected boolean verarbPruefeStatus()
  {
    if (this.nachName.equals("") | this.vorName.equals("") | !this.gebDatumStatus)
      this.personenStatus = false;
    else
      this.personenStatus = true;
    return this.personenStatus;
  }
}
```

Die nachfolgende Methode `verarbPruefeStatus()` aus der Klasse Student überschreibt die Methode `verarbPruefeStatus()` der Klasse Person. Dabei wird in der Verarbeitung der überschreibenden Methode mit `super` auf die überschriebene Methode zugegriffen:

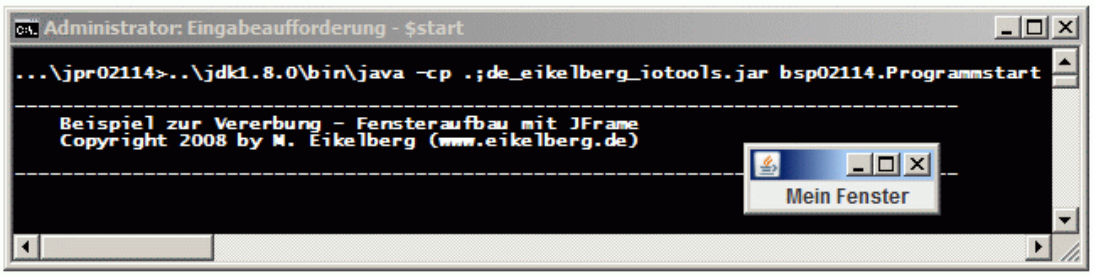
```
package bsp02113;


public class Student extends Person
{
/*
 * Attribute
 */
  ...

  private boolean studentStatus = true;
  private boolean matrikelNrStatus = true;
...

// Methode zur Überprüfung, ob gültige Daten im Objekt gespeichert sind
// Methode muss protected sein, da die überschriebene Methode protected ist und
// die Sichtbarkeit beim Überschreiben nicht reduziert werden kann
  protected boolean verarbPruefeStatus()
  {
    if (!super.verarbPruefeStatus() | !this.matrikelNrStatus)
      this.studentStatus = false;
    else
      this.studentStatus = true;
    return this.studentStatus;
  }
}
```

Mit Hilfe des Prinzips Überschreiben kann also im vorliegenden Beispiel erreicht werden, dass sowohl die vererbende Klasse als auch die erbende Klasse eine Methode `verarbPruefeStatus()` enthalten, die ein in gleichwertiges Ergebnis liefern, die jedoch eine unterschiedliche und an die jeweilige Klasse angepasste Verarbeitung enthalten.

2.9.2.7 Beispiel: Vererbung bei der Programmierung grafischer Benutzeroberflächen

Eine wichtige Anwendung des Prinzips der Vererbung stellt die Programmierung grafischer Benutzeroberflächen dar. Das nachfolgende

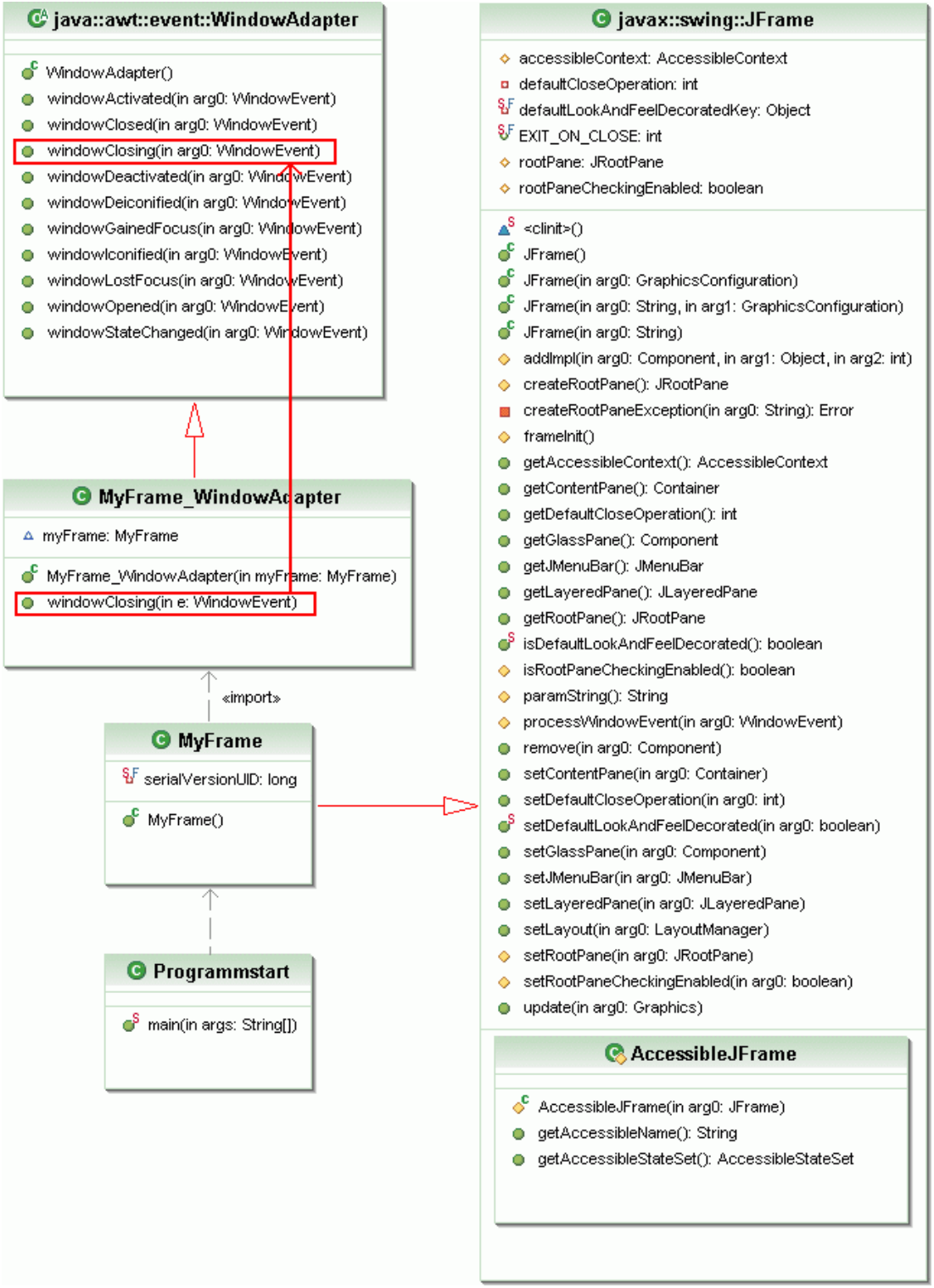
Beispiel jpr02114 zeigt die Programmierung eines einfachen Fensters mit der Klasse JFrame aus dem Swing-Paket des JDK. Gestartet wird das Programm im Projekt jpr02114 mit der main-Methode der Klasse bsp02114.Programmstart. Nach Start des Programms öffnet sich ein Fenster mit der Beschriftung "Mein Fenster":



Im Beispiel jpr02114 wurden folgende der in diesem Kapitel besprochenen Pinzipien eingesetzt:

1. **Programmierung des Fensters**
   - Vererbung: Das Fenster wurde programmiert in einer Klasse MyFrame. Diese ist abgeleitet von javax.swing.JFrame.
2. **Programmierung der Ereignisbehandlung für das Fenster**
   - Vererbung: Die Reaktion des Fensters auf Fensterereignisse (Schließen der Anwendung bei Schließen des Fensters) wurde programmiert in der Klasse MyFrame_WindowAdapter. Dieser Klasse ist abgeleitet von java.awt.event.WindowAdapter.
   - Überschreiben: In der Klasse MyFrame_WindowAdapter wird die Methode `windowClosing(WindowEvent e)` der Superklasse java.awt.event.WindowAdapter überschrieben und durch eine eigene Methode ersetzt, die für das Beenden der gesamten Anwendung sorgt. Alle anderen Methoden der Klasse java.awt.event.WindowAdapter werden über die Vererbung unverändet in die Subklasse MyFrame_WindowAdapter übernommen.

Das folgende Diagramm zeigt diese Beziehungen:



Nach Ableiten der Klasse MyFrame von der Klasse JFrame wurde in MyFrame wie folgt auf die geerbten Methoden zugegriffen, um das Fenster zu konfigurieren:

```
package bsp02114;

import javax.swing.*;
import java.awt.Container;

public class MyFrame extends JFrame
{
  ...

  public MyFrame()
  {
// An dieser Stelle wird zunächst automatisch der parameterlose
// Konstruktor der Klasse JFrame verarbeitet.

// Zugriff auf den Inhaltsbereich des Fensters mit geerbter Methode
```

```
    Container myCP = this.getContentPane();
// Einfügen einer Komponente (Label) in den Inhaltsbereich des Fensters
    myCP.add(new JLabel("       Mein Fenster     "));

// Mit geerbter Methode Adapterklasse für Fensterereignisse hinzufügen.
    MyFrame_WindowAdapter myFWA = new MyFrame_WindowAdapter();
    this.addWindowListener(myFWA);

//  Konfiguration des Fensters MyFrame mit geerbten Methoden
    this.setLocation(300, 100);  // Fensterort
    this.pack();                 // Fenstergröße einstellen
    this.setVisible(true);       // Fenster sichtbar machen
  }
}
```

Es gilt nun:

- In der Klasse MyFrame_WindowAdapter wird die Methode `windowClosing(WindowEvent e)` überschrieben.
- Die Methode `windowClosing(WindowEvent e)` ist bereits in der vererbenden Klasse java.awt.event.WindowAdapter vorhanden, enthält dort jedoch keine Verarbeitung.
- Die über Vererbung erhaltene "leere" Methode wird in der erbenden Klasse durch eine Methode ersetzt, die eine Verarbeitung enthält.

In der vererbenden Klasse ist somit bereits programmiert, dass auf das Ereignis windowClosing(WindowEvent e) reagiert werden soll. Allerdings enthält die Reaktion keinerlei Verarbeitung. In der erbenden Klasse wird jetzt diese verarbeitungslose Reaktion ersetzt durch eine Reaktion mit Verarbeitung:

```
package bsp02114;

import java.awt.event.*;

public class MyFrame_WindowAdapter extends java.awt.event.WindowAdapter
{
// Ereignisbehandlung für das Ereignis windowClosing (Betätigung des
// Schließen-Buttons in der rechten oberen Fensterecke) - überschreibt
// die gleichnamige Methode in java.awt.event.WindowAdapter
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

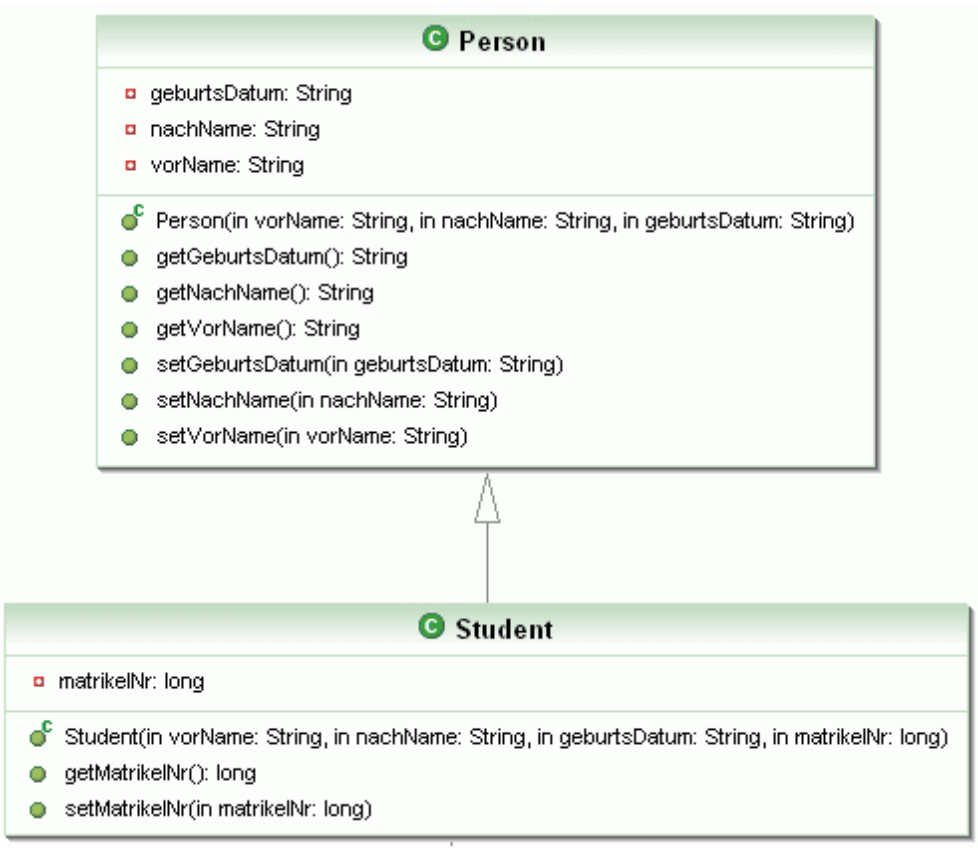
2.9.3 Techniken bei Vererbung

2.9.3.1 Polymorphie

Mit **Polymorphie** bezeichnet man die folgende Eigenschaft von Objektvariablen:

> Java-Methoden können **polymorph** sein: Es ist möglich, dass gleichnamige Methoden mit gleicher Signatur in verschiedenen über Vererbung verbundenen Klassen unterschiedliche Implementierung haben.
>
> Java-Objektvariablen sind **polymorph**: Jeder Objektvariablen einer Klasse `MeineKlasse` kann die Referenz auf ein Objekt einer Klasse zugewiesen werden, die von der Klasse `MeineKlasse` erbt. Beim Zugriff auf dieses Objekt über die Objektvariable der Klasse `MeineKlasse` stehen dann allerdings nur die Variablen und Methoden zur Verfügung, die in der Klasse `MeineKlasse` deklariert wurden und somit durch Vererbung in die erbende Klasse weitergegeben wurden. Es gilt somit:
>
> **Eine Objekt-Referenz kann stets in einer Objektvariablen einer übergeordneten Klasse gespeichert werden.**

Als Beispiel wird wie in Kapitel 2.9.2 auf folgende Klassenhierarchie zurückgegriffen (vgl. jpr02112):



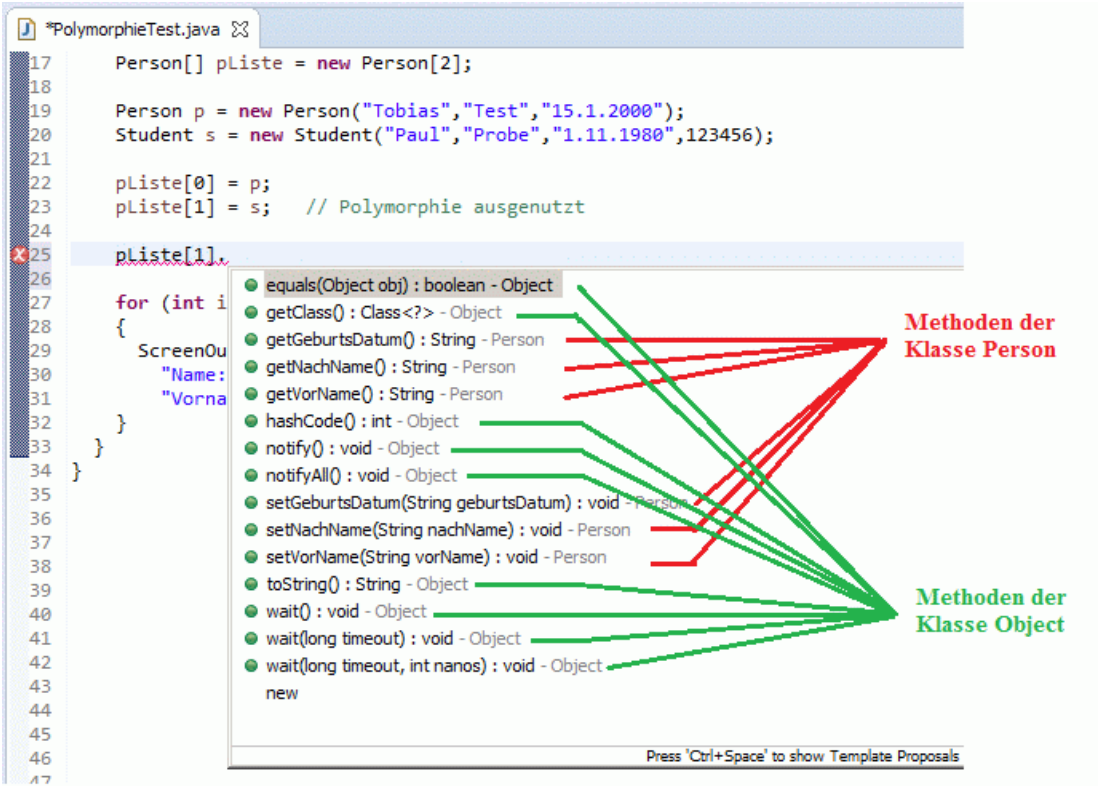
Die Klasse `Student` erbet von der übergeordneten Klasse `Person` u.a. die Methoden `genNachName()` und `getVorName`. Dies wird in folgendem Beispielprogramm verwendet, um über alle eine Namensliste zu erzeugen, die sowohl (allgemeine) Personen als auch Studenten enthält (vgl. Beispiel jpr02115):

```
Person[] pListe = new Person[2];

Person p = new Person("Tobias","Test","15.1.2000");
Student s = new Student("Paul","Probe","1.11.1980",123456);

pListe[0] = p;
pListe[1] = s;   // Polymorphie ausgenutzt

for (int i=0;i<pListe.length;i++)
{
  ScreenOut.kPrintln(
    "Name: " + pListe[i].getNachName() + " " +
    "Vorname: " + pListe[i].getVorName());
}
```

Im obigen Beispiel wird also

- der Objektvariablen pListe[1], die vom Datentyp Person ist,
- ein Objekt der Klasse Student zugewiesen, die von der Klasse Person abgeleitet ist.

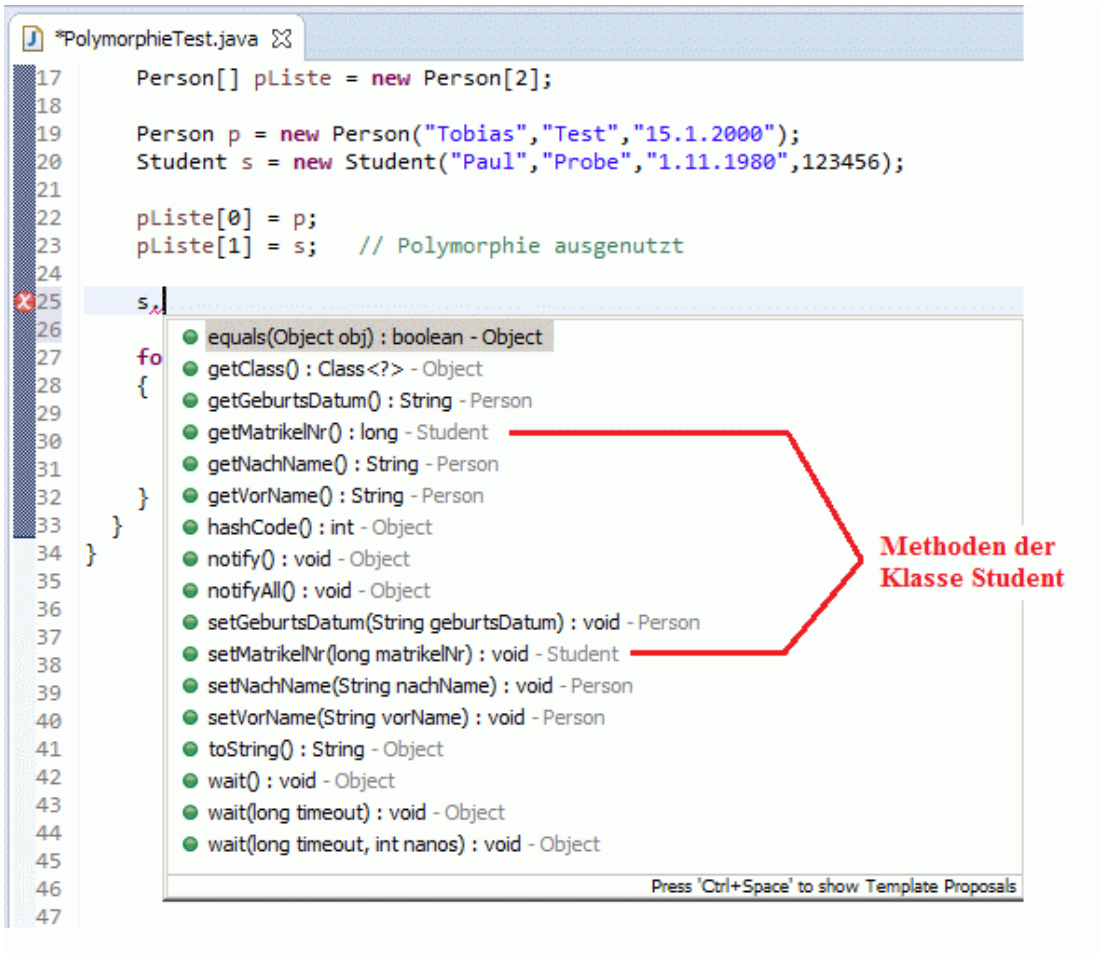
Wird so ein Objekt einer erbenden Klasse einer Variablen einer vererbenden Klasse zugewiesen, so steht danach beim Zugriff auf das Objekt über das Objekt der vererbenden Klasse nur die Variablen und Methoden der vererbenden Klasse zur Verfügung, wie der folgende Bildschirmausdruck zeigt:



In diesem Bildschirmausdruck werden als verfügbare Methoden somit die Methoden der Klasse `Object` als auch der Klasse `Person` angezeigt, denn die komplette Objekthierarchie sieht wie folgt aus:

```
java.lang.Object
  └bsp02112.Person
      └bsp02112.Student
```

Die Variablen und Methoden der Klasse Student werden somit nicht angezeigt, obwohl pListe[1] die Referenz auf eine Objekt der Klasse Student enthält. Diese würden allerdings angezeigt, wenn über die Variable s auf das Objekt zugegriffen würde:



2.9.3.2 Dynamisches Binden

> Bei der Programmierung mit Java erfolgt beim Methodenaufruf in der überwiegenden Anzahl der Fälle eine **dynamische Bindung**: Bei einem Methodenaufruf wird dann der ausführbare Code zur Methode nicht vom Compiler in die class-Datei eingefügt, in der der Methodenaufruf codiert ist. Zwar wird vom Compiler bereits geprüft, ob eine Implementierung der Methode zur Verfügung steht. Diese kann jedoch nach dem Compile ausgetauscht werden, da der ausführbare Code zur Implementierung erst zur Laufzeit von der virtuellen Maschine dynamisch geladen wird.

Mit anderen Worten wird bei dynamischer Bindung wie folgt auf den Quellcode einer aufgerufenen Methode zugegriffen:

- Der Compiler prüft beim Compile, ob es eine Methode gibt, deren Signatur (s.u.) zum Methodenaufruf passt.
- Auf den ausführbaren Code der Methode wird erst zur Laufzeit zugegriffen. Er kann nach dem Compile der Klasse mit dem Methodenaufruf ausgetauscht werden, ohne dass die rufende Klasse erneut compiliert werden muss.

Die **Signatur** einer Methode besteht aus dem Namen der Methode sowie der Liste der Datentypen der Parameter. Gibt es zu einem Methodennamen mehrere Methoden mit unterschiedlicher Signatur (also mit unterschiedlichen Parameterlisten), so bezeichnet man die jeweiligen Methoden als **überladen**: Überladene Methoden unterscheiden sich bezüglich der Signatur also durch unterschiedliche Listen von Datentypen der Parameter.

Die Suche nach der passenden Methode sowie die Festlegung, ob dynamische Bindung erfolgen soll, geschieht in folgenden Schritten (vgl. Horstmann, Cornell. Core. Band 1 Grundlagen. Kapitel 5.1.3):

Gegeben sei

- eine Vater-Klasse `VKlasse` und eine davon abgeleitete Klasse `SKlasse`.
- `s` ein ein Objekt der Klasse `SKlasse`.
- In der Klasse `SKlasse` gebe es mehrere (überladene) Methoden `tue(...)`.
- Betrachtet wird der Methodenaufruf `s.tue(...)`.

Mit diesen Voraussetzung kann das Verfahren zum Binden (suchen eines passenden Methodenaufrufs) wie folgt beschrieben werden:

1. Zunächst werden (vom Compiler bzw. der Virtuellen Maschine) alle Methoden `tue(...)` der Klasse `SKlasse` sowie alle sichtbaren Klassen `tue(...)` der Klasse `VKlasse` aufgelistet. Diese bilden die Kandidaten für die Auflösung des Methodenaufrufs.
2. Es erfolgt die Suche nach einer passenden Methode: Gibt es unter den in Schritt 1 aufgelisteten Methoden eine Methode, deren Parameterliste genau auf den Methodenaufruf passt, so wird diese Methode als passend gewählt. Ansonsten wird versucht, durch

implizite Typumwandlung für die Parameter eine Methode zu finden, deren Parameterliste passt. Ist dies nicht eindeutig möglich, erfolgt eine Fehlermeldung.

3. Ist die passende Methode vom Type private, static oder final (vgl. Kap. 2.9.4), so wird diese Methode für den Methodenaufruf gewählt. In diesem Fall erfolgt *keine dynamische Bindung* , vielmehr generiert der Compiler einen festen Programmcode für den Methodenaufruf. Andernfalls erfolgt *dynamische Bindung* und der Compiler generiert einen Programmcode für die Suche der Methode beim Programmlauf.
4. Bei dynamischer Bindung wird *während des Programmlaufs* wie folgt aus den passenden Methoden die zu verwendende Methode zur Auflösung des Methodenaufrufs `s.tue(...)` ausgewählt: Es wird der tatsächliche Typ des Objektes `s` festgestellt. Anschließend wird zunächst in der Klasse `SKlasse` nach einer passenden Methode gesucht. Wird keine Methode gefunden, so wird sukzessiv auf die jeweils bezüglich Vererbung übergeordnete Klasse zugegriffen und dort nach einer passenden Methode gesucht.

Beispiel: Die dynamische Bindung macht es möglich, ohne erneuten Compile in den Beispielprogrammen dieses Kurses die Datei de_eikelberg_iotools.jar durch eine neuere Version zu ersetzen,

- wenn für die static-Methoden im Paket (statische Bindung) lediglich die Implementierung geändert wurde und
- wenn für die nicht statischen Methoden dafür gesorgt wurde, dass bei dynamischer Bindung auch nach Änderung der Klassen die Suche nach der passenden Methode gemäß Schritt 4 stets zu einem Ergebnis führt.

Diese beiden Punkte werden bei Änderungen des Paketes de.eikelberg.iotools jedoch stets berücksichtigt.

2.9.3.3 Casting

Für den Begriff **Cast** verwendet man auch den deutschen Begriff **Typumwandlung**:

> Bei der Typumwandlung wird der Datentyp der Variablen eines einfachen Datentyps oder einer Objektvariablen geändert. Man unterscheidet:
>
> - **implizite Typumwandlung** - die Typumwandlung wird durch Compiler oder Virtuelle Maschine durchgeführt und ist nicht direkt durch ein Sprachelement sichtbar.
> - **explizite Typumwandlung** - durch Angabe des neuen Datentyps im Sprachelememt Cast.
>
> Das Sprachelement Cast besteht aus dem Namen des neuen Datentyps, umgeben mit runden Klammern und angeordnet direkt vor der entsprechenden Variablen. Typumwandungen für Referenzdatentypen sind nur dann möglich, wenn zwischen den beteiligten Datentypen eine Vererbungsbeziehung besteht.

Eine explizite Typumwandlung erfolgt in der Regel nach einer zuvor vorgenommenen impliziten Typumwandlung mit dem Ziel, wieder auf die durch die implizite Typumwandlung verloren gegangenen Methoden des Objektes zugreifen zu können. Das Beispiel jpr02116 verwendet die Klassen `Person` und `Student` des Paketes bsp02112 und zeigt die Wirkung von Typumwandlungen:

Typumwandlung für einfache Datentypen

```
// Typumwandlungen für einen einfachen Datentyp:
    int zaehler = 1;
// implizite Typumwandlung
    long zaehlerLong = zaehler;
// explizite Typumwandlung mit Cast
    zaehler = (int)zaehlerLong;
```

Im obigen Beispiel wird die in der int-Variablen `zaehler` gespeicherte Zahl 1 zunächst mit einer impliziten Typumwandlung in eine long-Zahl umgewandelt und anschließend wieder mit einer expliziten Typumwandlung zurück in eine int-Zahl.

Typumwandlung für Referenzdatentypen

Im nachfolgenden Programmausschnitt von bsp02116.TypUmwandlung

- wird ein Objekt der Klasse bsp02112.Student angelegt. anschließend.
- Anschließend wird die Referenz auf das angelegte Objekt in einer Objektvariablen der Klasse Person umgespeichert, wobei eine implizite Typumwandlung erfolgt. Diese Typumwandlung ist möglich, weil die Klasse Student von der Klasse Person abgeleitet ist:

```
    Student s = new Student("Vera","Versuch","10.10.1980",123456);
// implizite Typumwandlung
    Person[] pListe = new Person[1];
    pListe[0] = s;
```

Die Eigenschaften des Objektes selber werden von der Typumwandlung nicht verändert. Die Typumwandlung beeinflusst lediglich die Verfügbarkeit von Variablen und Methoden. Nach der Typumwandlung kann bei Zugriff über die neue Variable (nur noch) auf die Variablen und Methoden des neuen Datentyps - also der Klasse Person - zugegriffen werden (obwohl die Variablen und Methoden der alten Klasse noch vorhanden sind):

```
// Nach der Typumwandlung nicht mehr möglicher Aufruf:
//  pListe[0].getMatrikelnummer()
// Der Aufruf ist jedoch über die Objektvariable s noch möglich:
    s.getMatrikelNr();
```

Soll zu einem späteren Zeitpunkt wieder auf die speziellen Variablen und Methoden der erbenden Klasse zugegriffen werden, muss eine explizite Typumwandlung zurück in den ursprünglichen Datentyp erfolgen. Hierbei sollte

- entweder vor der Typumwandlung eine Prüfung erfolgen, ob die explizite Typumwandlung möglich ist
- oder es muss eine gegebenenfalls auftretende Exception (vom Typ ClassCastException) behandelt werden.

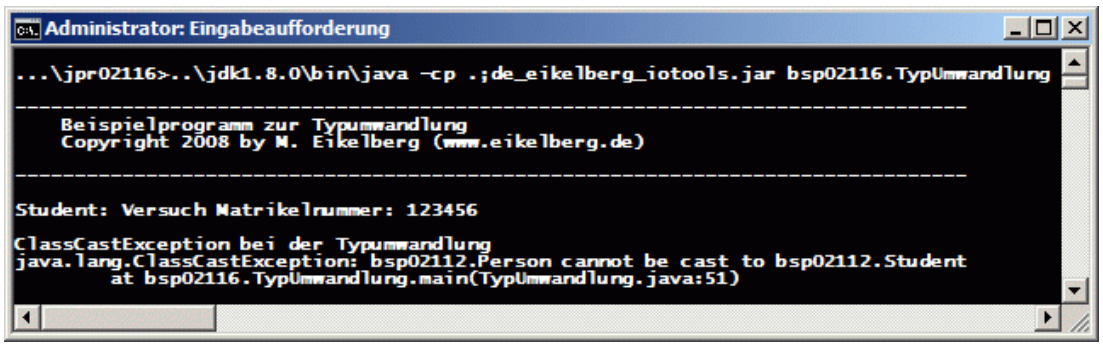
Die Prüfung, ob eine Typumwandlung möglich ist, kann wie folgt durchgeführt werden:

```
    Student s = new Student("Vera","Versuch","10.10.1980",123456);
// implizite Typumwandlung
    Person[] pListe = new Person[1];
    pListe[0] = s;
// Prüfung, ob Cast möglich ist
    if (pListe[0] instanceof Student)
    {
      Student s1 = (Student)pListe[0];            // Cast
      ScreenOut.kPrintln(
        "\nStudent: " + s1.getNachName() + " " + // Methode der Klasse Person
        "Matrikelnummer: " + s1.getMatrikelNr()  // Methode der Klasse Student
      );
    }
```

Alternatv kann wie folgt eine Behandlung möglicherweise auftretender Exceptions erfolgen:

```
    Person p = new Person("Tobi","Test","20.2.1980");
    pListe[0] = p;
// Cast mit Exception-Behandlung
    try
    {
      Student s2 = (Student)pListe[0];            // Cast
      ScreenOut.kPrintln(
        "\nStudent: " + s2.getNachName() + " " + // Methode der Klasse Person
        "Matrikelnummer: " + s2.getMatrikelNr()  // Methode der Klasse Student
      );
    }
    catch (ClassCastException e)
    {
      ScreenOut.kPrintln("\nClassCastException bei der Typumwandlung");
      e.printStackTrace();
    }
```

Da im obigen Beispiel das Objekt ursprünglich ein Objekt der Klasse Person ist und nicht über die Variablen und Methoden der Klasse Student verfügt, tritt bei der Typumwandlung eine ClassCastException auf:



2.9.4 Steuerung von Vererbungsvorgängen

Zur Steuerung des Vererbungsvorgangs stehen folgende Techniken zur Verfüugung:

- Verhinderung von Vererbung durch Kennzeichnung einer Klasse oder einer Methode als `final`
- Erzwingung eines Vererbungsvorgangs durch Kennzeichnung einer Klasse als `abstract`
- Steuerung von Implementierungsvorgängen mit Schnittstellen

2.9.4.1 Finale Klassen und Methoden

Der Überschreibungsvorgang für Methoden sowie der Vererbungsvorgang für Klassen wie folgt mit Hilfe der Eigenschaft `final` gesteuert werden:

- **Vererbung verhindern:** Wird eine Klasse als `final` gekennzeichnet, kann keine weitere Klasse mehr von ihr abgeleitet werden.
- **Überschreiben verhindern:** Wird eine Methode als `final` gekennzeichnet, so kann man in abgeleiteten Klassen diese Methode nicht mehr überschreiben.

Hinweis: Die Kennzeichnung einer Klassen- oder Instanzvariablen als final hat keine Bedeutung im Zusammenhang mit der Vererbung. Vielmehr hat diese Kennzeichnung zur Folge, dass der Wert der Variablen *in Anweisungen der eigenen Klasse* nicht mehr geändert werden kann.

Beispiele zu finalen Klassen

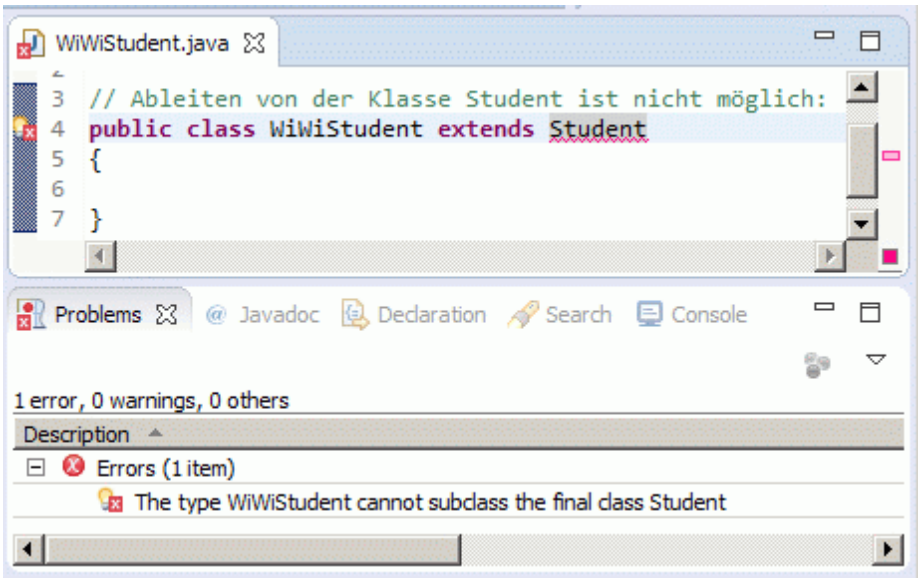
- Die Klasse java.lang.String ist wie folgt als final deklariert:

  ```
  public final class String
  ```

  Daher kann keine Klasse von der String-Klasse abgeleitet werden.
- Im Beispiel jpr02117 ist die Klasse bsp02117.Student als final deklariert:

  ```
  public final class Student extends Person
  ```

  Daher wird der Versuch vom Compiler abgewiesen, von der Klasse Student eine weitere Klasse abzuleiten:



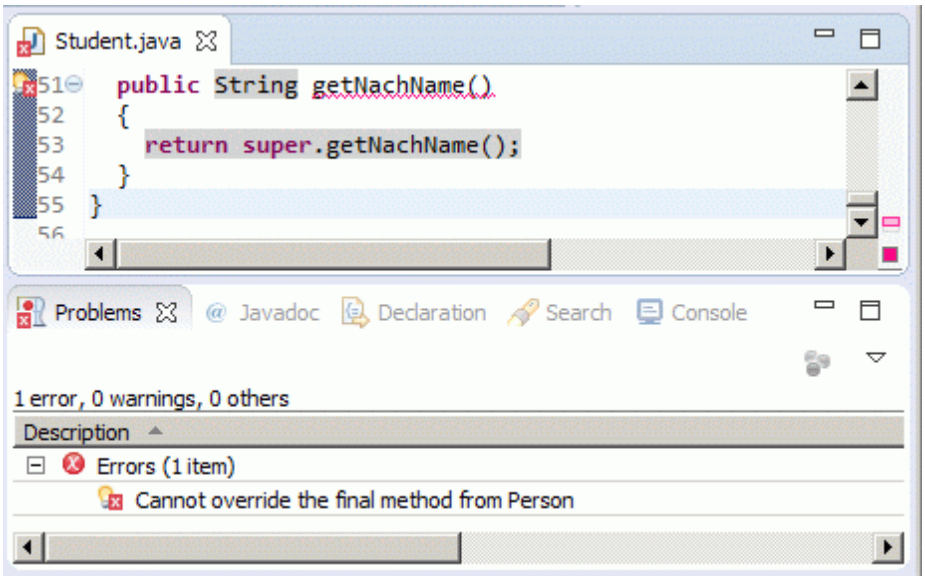
Beispiel zu finalen Methoden

In der Klasse bsp02117.Person sind die Methoden getNachName() und getVorName() als final deklariert:

```
public final String getVorName()
{
  return this.vorName;
}
```

```
  public final String getNachName()
  {
    return this.nachName;
  }
```

Dies führt dazu, dass in der abgeleiteten Klasse `Student` diese beiden Methoden nicht mehr überschrieben werden können:



2.9.4.2 Abstrakte Klassen und Methoden

- Die Kennzeichnung einer Methode mit dem Schlüsselwort `abstract` - also als **abstrakte Methode - ermöglicht, in einer Klasse eine Methode ohne Implementierung zu codieren.**
- **Klassen, die eine abstrakte Methode enthalten, müssen ebenfalls als abstrakt gekennzeichnet werden.**
- **Es ist nicht möglich, von einer abstrakten Klasse Objekte zu bilden. Allerdings können Variablen abstrakter Klassen *deklariert* werden, denen nach Typumwandlung Objektreferenzen abgeleiterer Klassen zugewiesen werden können.**
- **In einer von einer abstrakten Klasse erbenden Klasse kann das Überschreiben der abstrakten Methode unterbleiben. Dann muss allerdings die erbende Klasse ebenfalls als abstrakt gekennzeichnet werden.**

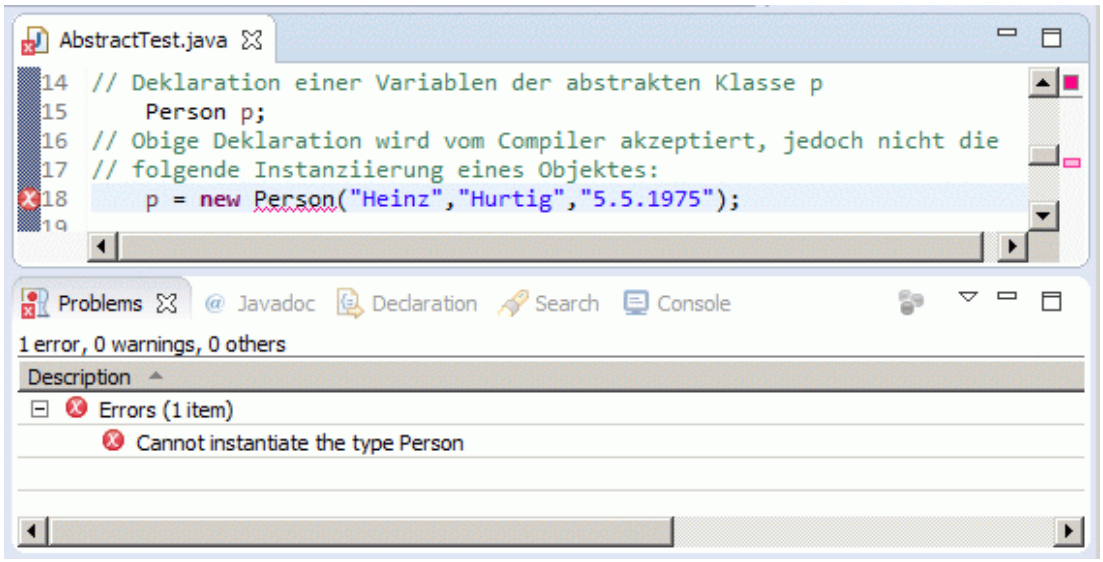
Mit anderen Worten kann eine Objektbildung erst dann erfolgen, wenn von der abstrakten Klasse - gegebenenfalls nach mehreren Vererbungsvorgängen - durch Vererbung eine nicht abstrakte Klasse abgeleitet worden ist, in der alle abstrakten Methoden implementiert worden sind.

Beispiel

Die Klasse bsp02118.Person ist als abstrakt deklariert, da sie eine abstrakte Methode enthält:

```
package bsp02118;

public abstract class Person
{
  ...

  public abstract String getBeschreibung();

  ...
}
```

Man beachte im obigen Quellcode das Semikolon direkt nach der Deklaration der Methode. Damit steht fest, dass für diese Methode kein Block mit einer Implementierung bereitgestellt wird. Wegen der fehlenden Implementierung der Methode kann von der Klasse Person jetzt keine Instanz gebildet werden - allerdings ist die Deklaration eines Objektes möglich (vgl. bsp02118.AbstractTest):



Da die Klasse Person abstrakt ist, ist zur Objektbildung zunächst ein Vererbungsvorgang erforderlich, wobei in der erbenden Klasse die Implementierung der Methode getBeschreibung erfolgen muss, damit eine Objektbildung möglich ist. In Beispiel jpr02118 erfolgt dies in der Klasse bsp02118.Student:

```
package bsp02118;

public class Student extends Person
{
  ...

  public String getBeschreibung()
  {
    String s;
      s =  "Student "+this.getNachName()+
           " hat Matrikelnummer "+this.getMatrikelNr();
    return s;
  }
}
```

Diese Klasse braucht nicht mehr als abstract gekennzeichnet werden, da die Klasse eine Implementierung der Methode getBerschreibung() bereitstellt. Entsprechend kann nun von dieser Klasse ein Objekt gebildet werden (vgl. bsp02118.AbstractTest):

```
package bsp02118;

...

public class AbstractTest
{

  public static void main(String[] args)
  {
    ...
// Deklaration einer Variablen der abstrakten Klasse p
    Person p;
// Instanziierung eines Objektes der erbenden Klasse Student:
    Student s = new Student("Paul","Probe","4.4.1984",444444);
// Kopieren der Referenz in eine Variable der abstrakten Klasse Person
    p = s;
// Zugriff auf eine Methode über die Variable der abstrakten Klasse
    ScreenOut.kPrintln(p.getBeschreibung());
  }
}
```

Man beachte folgende **Besonderheit**: Die Anweisung

```
    p = s;
```

stellt eine implizite Typumwandlung von der Klasse Student in die Klasse Person dar. Damit gehen beim Methodenaufruf über den Objektnamen p zunächst alle Methoden verloren, über die die Klasse `Student` zusätzlich zu den Methoden der Klasse `Person` verfügt. Da die Methode `getBeschreibung()` jedoch als abstrakte Methode in der Klasse `Person` enhalten ist, schafft es die virtuelle Maschine, beim Methodenaufruf mit dynamischer Bindung über eine als Objekt der Klasse `Person` deklarierte Variable auf die Methode `getBeschreibung()` der Klasse `Student` zuzugreifen:

```
Administrator: Eingabeaufforderung
...\jpr02118>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02118.AbstractTest
--------------------------------------------------------------------------------
    Testprogramm zu abstrakten Methoden
    Copyright 2008 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Student Probe hat Matrikelnummer 444444
```

Hinweis: Kommt man beim objektorientierten Entwurf zu dem Ergebnis, dass eine abstrakte Klasse ohne eine einzige implementierte Methode vorliegt, kann man besser statt einer abstrakten Klasse eine sogenannte Schnittstelle programmieren (vgl. 2.9.4.3)

2.9.4.3 Schnittstellen

Die Schnittstelle (Interface) stellt ein sehr wichtiges methodisches Hilfsmittel bei der Java-Programmierung dar. Wichtige Anwendung von Schnittstellen finden sich bei folgenden Themen:

- **Datenbankprogrammierung mit Java Database Connectivity (JDBC)** - die vom Programmierer bei der Datenbankprogrammierung zu verwendenden Methodennamen sind in den JDBC-Interfaces festgelegt. Die einzelnen Hersteller der Datenbankserver müssen dann die JDBC-Methoden bereitstellen.
- **Client-Server-Programmierung mit Java Remote Method Invocation (RMI)** - der Programmierer des Servers deklariert die Methodenaufrufe, die die Clients nutzen sollen, in einem Interface. Anschließend implementiert er das Interface auf dem Server. Die Client-Programmierer erhalten nur das Interface und kennen so die Methoden-Implementierung. Erst beim Programmlauf werden dann die Methodenaufrufen aufgelöst.

- Eine Schnittstelle (Interface) ist keine Klasse, sondern ein Zusammenstellung von Deklarationen von Methoden. Schnittstellen enthalten keine Variablen und keine Implementierungen von Methoden.
- Es ist nicht möglich, von Objekte von einer Schnittstelle zu instanziieren. Eine Deklaration einer Variablen vom Typ einer Schnittstelle ist jedoch möglich.
- Eine Schnittstelle wird implementiert, d.h. es wird eine Klasse programmiert, die zu jeder in der Schnittstelle genannten Methode einen Block mit den ausführbaren Anweisungen - die sogenannte **Implementierung** - enthält.
- Eine Klasse kann mehrere Schnittstellen implemtieren.
- Alle Methoden einer Schnittstelle sind public, auch wenn dieses Schlüsselwort fehlt.

Man beachte insbesondere: Eine Klasse kann mehrere Schnittstellen implementieren. Dies steht im Gegensatz zu der Tatsache, dass eine Klasse nur von genau einer anderen Klasse erben kann. Eine Klasse kann also auch nur von einer abstrakten Klasse erben, jedoch kann sie mehrere Schnitstellen implementieren. Dies ist der wichtigste Grund, warum Java neben abstrakten Klassen zusätzlich auch noch Schnittstellen bereitstellt.

- Man kann Schnittstellen auch als "Aufgabenliste" ansehen, die von einem anderen Programmierer erfüllt werden muss (z. Bsp. JDBC).
- Andersherum kann eine Schnittstelle aber auch ein "Versprechen" interpretiert, das ein Programmierer abgibt - nämlich dass er die in der Schnittstelle angegebenen Methoden zur Laufzeit bereitstellt (z. Bsp. JMI).

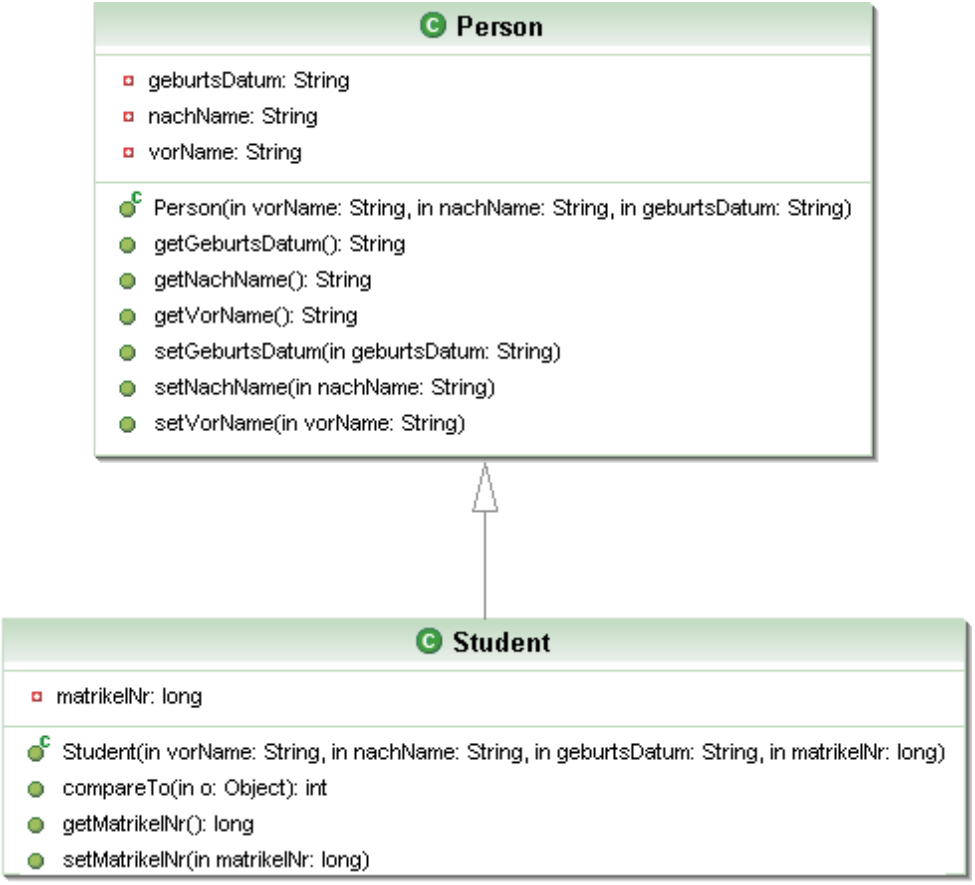
Beispiel: Sortieren von Arrays

Eine wichtiges Beispiel ist das Sortieren von Arrays. Hierzu stellt die Klasse `java.util.Arrays` die Methode `Arrays.sort` bereit, die die Elemente des Arrays *im Array selber* sortiert. Voraussetzung hierzu ist jedoch, dass die Elemente des Arrays die Schnittstelle `java.lang.Comparable<T>` implementieren. Diese Schnittstelle ist generisch (vgl. Kapitel 2.11) und enthält u. a. folgenden Quellcode:

```
public abstract interface java.lang.Comparable<T>
{
  public abstract int compareTo(T o);
}
```

Das Schlüsselwort `abstract` ist im Interface Comparable codiert, es kann jedoch im Allgemeinen auch fehlen und ist für ein Interface nicht zwingend notwendig (vgl. 2.9.4.4). Implementierungen des Interface Comparable müssen somit eine Methode `compareTo(T o)` mit der angegebenen Signatur bereitstellen. Dabei ist T der generische Typ des Interface.

Im Beispiel jpr02119 soll ein Array aus Objekten der Klasse bsp02119.Student sortiert werden. Die Klasse bsp02119.Student ist abgeleitet von der Klasse bsp02119.Person:



Die Klasse bsp02119.Person enthält folgenden Quellcode, der u.a. auch eine Implementierung der Methode `compareTo(...)` enthält:

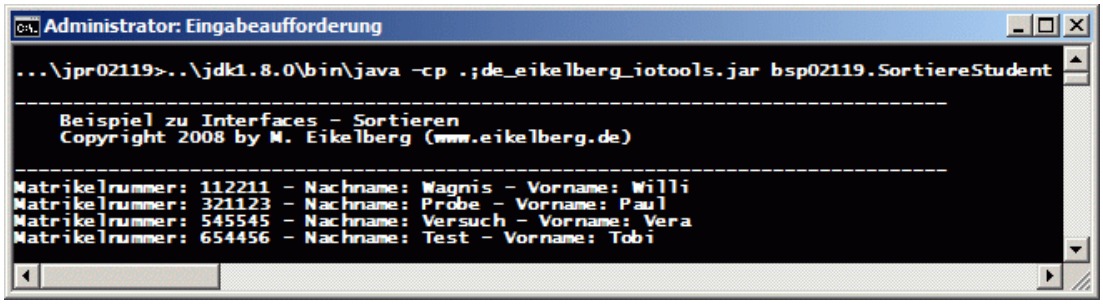
```
package bsp02119;

public class Student extends Person implements Comparable<Student>
{
/*
 * Attribute
 */
  private long matrikelNr = 0L;

/*
 * Konstruktoren
 */
  public Student(String vorName, String nachName, String geburtsDatum, long matrikelNr)
  {
// Der Super-Konstruktor muss stets als erste Anweisung aufgerufen werden!
    super(vorName, nachName, geburtsDatum);
    this.matrikelNr = matrikelNr;
  }

/*
 * get-Methoden
 */
  public long getMatrikelNr()
  {
    return this.matrikelNr;
  }

/*
 * set-Methoden
 */
  public void setMatrikelNr(long matrikelNr)
  {
    this.matrikelNr = matrikelNr;
  }

// Implementierung der Methode compareTo(T o), gefordert im
// Interface Comparable, ermöglicht das Sortieren der Student-Objekte
  public int compareTo(Student s)
```

```
  {
    int comp = 0;
// Aufsteigend Sortieren nach der Matrikelnummer
    if (s.getMatrikelNr() > this.getMatrikelNr())
      comp = -1;   // Matrikelnummer des aktellen Objekte ist "kleiner"
    else if (s.getMatrikelNr() < this.getMatrikelNr())
      comp = 1;    // Matrikelnummer des aktellen Objekte ist "größer"
    return comp;
  }

}
```

Damit enthält die Klasse bsp02119.Student die von dem Interface java.lang.Comparable geforderte Methode und implementiert damit dieses Interface. Deshalb ist es jetzt möglich, mit der Methode `Arrays.sort(...)` einen Array aus Objekten der Klasse Student zu sortieren:

```
package bsp02119;

import de.eikelberg.iotools.*;
import java.util.Arrays;

public class SortiereStudent
{

  public static void main(String[] args)
  {
    ScreenOut.kStartScreenOut(
      "Beispiel zu Interfaces - Sortieren",
      "M. Eikelberg (www.eikelberg.de)","2008");

// Anlegen eines Arrays aus Student-Objekten
    Student[] sListe = new Student[4];
    sListe[0] = new Student("Tobi","Test","1.1.197",    654456);
    sListe[1] = new Student("Paul","Probe","2.2.1972",  321123);
    sListe[2] = new Student("Vera","Versuch","3.3.1973",545545);
    sListe[3] = new Student("Willi","Wagnis","4.4.1974",112211);

// Sortieren mit der Methode Arrays.sort() - funktioniert immer dann,
// wenn die zu sortierenden Objekte das Interface Comparable implementieren
    Arrays.sort(sListe);

// Ausgabe der sortieren Liste
    for (int i=0;i<sListe.length;i++)
      ScreenOut.kPrintln(
        "Matrikelnummer: "+sListe[i].getMatrikelNr()+" - "+
        "Nachname: "      +sListe[i].getNachName()  +" - "+
        "Vorname: "       +sListe[i].getVorName()
      );
  }
}
```

In der Tat erfolgt beim Programmlauf eine sortierte Ausgabe des Arrays, das Sortieren erfolgte somit im Array selbst:



2.9.4.4 Exkurs: Anwendung von Schnittstellen bei der Client-Server-Programmierung mit Java RMI

Für die Client-Server-Programmierung gibt es das etablierte Verfahren CORBA. Dieses ist jedoch relativ komplex und aufwändig. Alternativ kann bei der Programmierung mit Java stattdessen die Methode der **Java Remote Method Invocation(RMI)** verwendet werden. In diesem Abschnitt wird ein einfaches Beispiel zur RMI-Programmierung vorgestellt, denn bei der RMI-Programmierung werden vom Programmierer erstellte Interfaces verwendet. Damit stellt dieses Beispiel auch ein Beispiel zur Verwendung von Schnittstellen dar.

Bei der RMI-Programmierung können der Server und der Client von unterschiedlichen Programmierern erstellt werden. Damit der Serveraufruf gelingt, wird wie folgt zwischen den Programmierern des Servers und des Clients eine Vereinbarung getroffen:

> Der Programmierer des RMI-Serverdienstes *verspricht* dem Programmierer des Client-Programms, gewisse Methoden für Methodenaufrufe über das Netzwerk bereitzustellen. Dieses Versprechen geschieht in Form eines Interfaces, in dem die Methodendeklaration festgehalten wird. Der Programmierer des RMI-Serverdienstes muss die Implementierung dieser Methoden auf dem Server bereitstellen. Der Programmierer des Client-Programms kennt die Implementierung nicht, ihm genügt die Kenntnis der Syntax beim Methodenaufruf.

Im Beispiel jpr02120 stellt der RMI-Serverdienst (das sogenannte *Remote Object*) die folgende im Interface bsp02120.rmiserver.RmiMethodenInterface genannte Methode bereit. Zunächst wird dieses Interface betrachtet, das die Absprache zwischen den Programmierern von Server und Client regelt:

```
package bsp02120.rmiserver;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RmiMethodenInterface extends Remote
{
  public String getServerZeit() throws RemoteException;
}
```

Man sieht:

- `RmiMethodenInterface` ist keine Klasse, sondern ein Interface - gekennzeichnet mit dem Schlüsselwort `interface`.
- Das Interface `RmiMethodenInterface` ist abgeleitet von dem Interface `java.rmi.Remote`. Jedes Objekt, das Methodenaufrufe über das Netzwerk bereitstellt, muss von dem Interface `java.rmi.Remote` abgeleitet sein. Das Ableiten der Klasse von der Methode Remote bewirkt jedoch, dass bei der Client-Programmierung die Adresse des RMI Dienstes (des Remote Object) in einer Objekt-Variablen vom Typ Remote bereitgestellt wird (s.u.).
- Im Interface `RmiMethodenInterface` ist eine einzige Methode deklariert, die Methode getServerZeit(), public, mit String Rückgabewert und eine `RemoteException` werfend.
- Man beachte: **Alle im Interface deklarierten Methoden müssen in der Deklaration mit der Klausel** `throws RemoteException` **codiert werden, auch wenn dies vom Compiler nicht gefordert wird. Fehlt diese Klausel, so erhält man zur Laufzeit eine java.rmi.server.ExportException mit der Meldung** "`illegal remote method encountered`".

Auf der Serverseite wurde das im Interface gegeben Versprechen wie folgt in der implementierenden Klasse `RmiMethodenImplementierung` eingelöst:

```
package bsp02120.rmiserver;

import java.rmi.*;
import java.rmi.server.*;
...

public class RmiMethodenImplementierung
  extends UnicastRemoteObject implements RmiMethodenInterface
{

  private static final long serialVersionUID = 710L;

// Der Konstruktor erzeugt ein Remote Objekt, das auf Port 15550 horcht
  public RmiMethodenImplementierung() throws RemoteException
  {
    super(15550);
  }

  public String getServerZeit() throws RemoteException
  {
    String zeit = this.getZeit();
    System.out.println("\nZeitabfrage von Client - aktuelle Zeit: "+zeit);
    return zeit;
  }

  private String getZeit()
  {
// Uhrzeit feststellen:
   ...
// Uhrzeit umformatieren
   ...
  }
}
```

Der Programmcode der Implementierung bsp02120.rmiserver.RmiMethodenImplementierung enthält folgende Abschnitte:

- In der Klassendeklaration wird festgelegt, dass die Klasse `RmiMethodenImplementierung` von der Klasse `java.rmi.server.UnicastRemoteObject` abgeleitet ist, wodurch das beim Serverstart gebildete Objekt der Klasse `RmiMethodenImplementierung` zu einem Remote Object wird. Ferner wird festgelegt, dass die Klasse `RmiMethodenImplementierung` das Interface `RmiMethodenInterface` implementiert, wodurch das vom Programmierer des Servers im Interface gegebene *Versprechen* eingelöst wird.
- Im Konstruktor der Klasse `RmiMethodenImplementierung` wird der Superkonstruktor, also der Konstruktor der Klasse `UnicastRemoteObject` aufgerufen, um ein Remote Object zu erzeugen, das im vorliegenden Beisiel auf TCP Port 15550 auf eingehende Methodenaufrufe horcht.
- Die Klasse `RmiMethodenImplementierung` stellt eine Implementierung der im Interface genannten Methode getServerZeit() bereit. Diese gibt eine Meldung auf dem Server aus und schickt die auf dem Server festgestellte und formatierte Zeit an den Client.

Mit den beiden obigen Klassen ist die Programmierung des RMI-Servers bereits weitgehend abgeschlossen. Es fehlt die main-Methode für den Serverstart sowie eine Datei mit den Datei- und Netzwerkberechtigungen, die der Server haben soll (das sind also Java-eigene Firewall-Regeln). Diese Regeln werden in eine Datei eingetragen (rmipolicy_server.txt) und lauten wie folgt:

```
grant
{
  // Permissions für den RMI-Server:
  permission java.net.SocketPermission "127.0.0.1:1-65535", "accept,connect,listen,resolve";

  // Permissions für eingehende Client-Abfragen
  permission java.net.SocketPermission "127.0.0.1:15550", "accept,connect,listen,resolve";

  // Permissions für die RMI Registry:
  permission java.net.SocketPermission "127.0.0.1:11099", "accept,connect,listen,resolve";

  // Permissions für die Verarbeitung
  permission java.lang.RuntimePermission "createSecurityManager";
  permission java.lang.RuntimePermission "setSecurityManager";
};
```

Der RMI-Server benötigt eine SocketPermission für Netzwerkverbindungen vom Client, eingehend auf Port 15550 sowie mit sich selber kommunizierend auf beliebigen Ports. Ferner muss der RMI-Server sich bei der sogeannten RMI Registrierung anmelden. Die RMI-Registrierung ist das Programm rmiregistry.exe des JDK, das vor dem Start des Servers auf dem Server gestartet werden muss und das im vorliegenden Fall so gestartet wurde, dass es auf TCP Port 11099 horcht. Der Standard-Port 1099 ist auf Servern in der Regel nicht geeignet, da er nach dem Serverstart oft bereits dynamisch vergeben wurde und damit für die RMI Registrierung nicht zur Verfügung steht.

Mit dem Interface, der Implementierung sowie den Permissions kann jetzt wie folgt einfach der Server programmiert werden:

```
package bsp02120.rmiserver;

import java.rmi.Naming;

public class Server {

  public static void main(String[] args)
  {
    ...

    try
    {
// Security Manager konfigurieren und setzen
      String policypath="bsp02120/rmiserver/rmipolicy_server.txt";
      System.setProperty("java.security.policy",policypath);
      System.setProperty("java.rmi.server.hostname","127.0.0.1");
      if (System.getSecurityManager() == null)
        System.setSecurityManager(new SecurityManager());

// Remote Object instanziieren und als Dienst bei der RMI Registrierung anmelden
      RmiMethodenImplementierung myRmiMethImpl = new RmiMethodenImplementierung();
      String rmiUrl = "rmi://127.0.0.1:11099/rmidienst";
      System.out.println("\nDie URL des RMI-Dienstes lautet: "+rmiUrl);
      Naming.rebind(rmiUrl,myRmiMethImpl);
      System.out.println(
        "\nRMI-Dienst ist angemeldet, Server wartet auf Client-Zugriffe...\n");
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
```

}

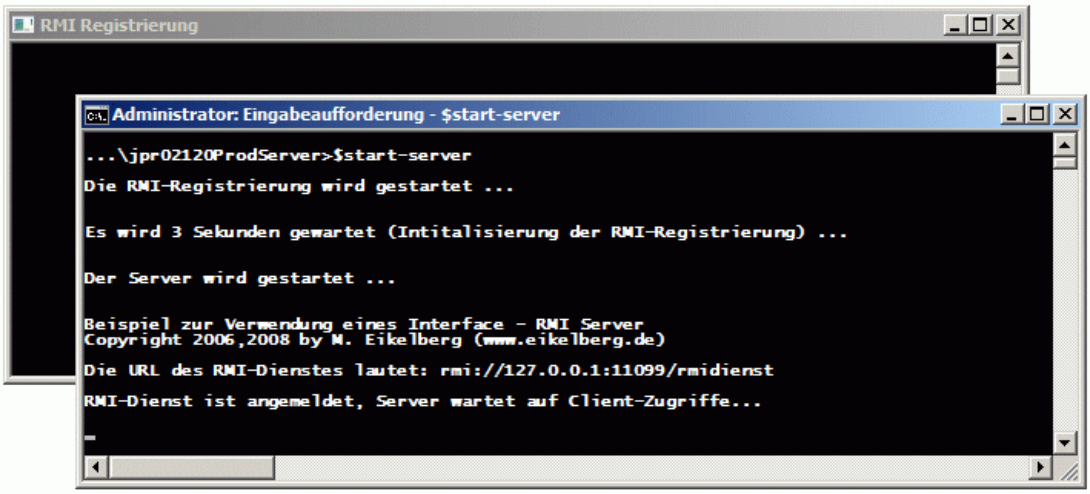
Die das Remote Objekt betreffenden wesenlichen Schritte sind wie folgt

1. Es wird ein Objekt der Implementierung des Interface angelegt.
2. Es wird die URL des Remote Objektes festgelegt - im vorliegenden Fall erhält das Objekt den Namen "rmidienst".
3. Mit der Methode rebind(...) der Klasse Naming werden die Implementierung und die URL miteinander verbunden, so dass das Remote Objekt über das Netzwerk erreicht werden kann.

Wie bereits oben erwähnt, ist für den Start des Servers das Programm rmiregistry.exe (RMI Registrierung) erforderlich. Zwischen dem Start des Programms rmiregistry.exe und dem RMI-Server sollte mindestens ca. 3 Sekunden gewartet werden, damit sich die RMI Registrierung initialisieren kann. Dieses Warten wird im vorliegenden Fall vom Programm bsp02120.timer.Timer erledigt. Insgesamt kann der Serverstart also mit folgender Batch-Datei erfolgen:

```
@Echo off

rem
rem ggf. belegte Systemvariablen löschen:
rem

set classpath=.
set sourcepath=.
set bootclasspath=.
set extdirds=.

Echo.
Echo Die RMI-Registrierung wird gestartet ...
Echo.
start "RMI Registrierung" ..\jdk1.8.0\bin\rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
11099

Echo.
Echo Es wird 3 Sekunden gewartet (Intitalisierung der RMI-Registrierung) ...
Echo.

..\jdk1.8.0\bin\java bsp02120.timer.Timer 3

Echo.
Echo Der Server wird gestartet ...
Echo.

rem aktuellen Pfad ermitteln, anschließend \ durch / ersetzen
set pfad=%~d0%~p0
set pfad=%pfad:\=/%

..\jdk1.8.0\bin\java -cp . -Djava.rmi.server.codebase=file:%pfad%
-Djava.rmi.server.useCodebaseOnly=false bsp02120.rmiserver.Server

Echo.

pause
```

Zunächst startet die RMI Registrierung in einer separaten Konsolbox, nach der Wartezeit dann der RMI Server:

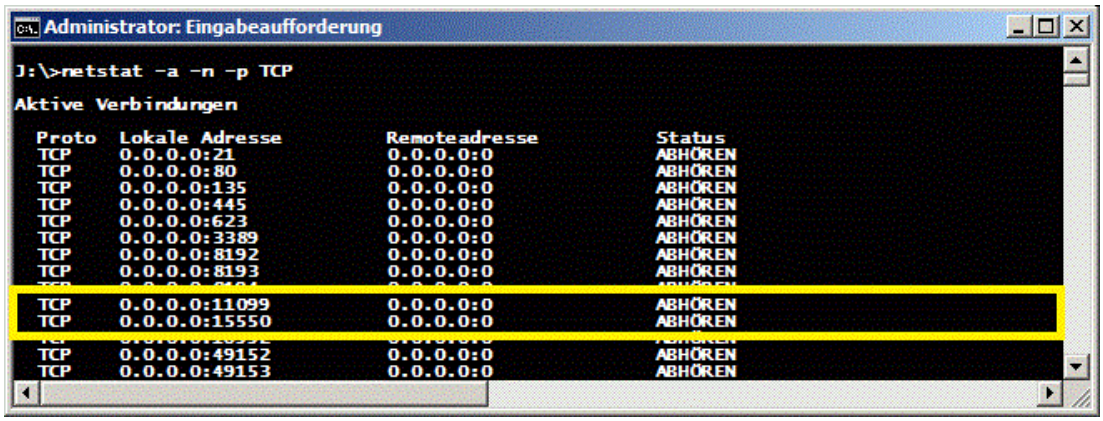


Bei Eingabe des Befehls

```
netstat -a -n -p TCP
```

werden die bestehenden Verbindungen des Servers angezeigt:

- Die RMI-Registrierung horcht auf Port 11099
- Das Remote Object horcht auf Port 15550.



Der Server wartet nun auf die Zugriffe der Clients. Diese benötigen das Recht, auf die beiden horchenden Ports des Servers zuzugreifen. Diese Rechte werden wieder in einer Datei eingetragen (rmipolicy_client.txt):

```
grant
{
  // Permissions für den RMI-Server:
  permission java.net.SocketPermission "127.0.0.1:15550", "accept,connect,listen,resolve";

  // Permissions für die RMI Registry:
  permission java.net.SocketPermission "127.0.0.1:11099", "accept,connect,listen,resolve";
};
```

Das Clientprogramm greift jetzt auf das Interface `RmiMethodenInterface` zu und kann wie folgt in einfacher Weise programmiert werden:

```
package bsp02120.rmiclient;

import java.rmi.Remote;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.NotBoundException;

import bsp02120.rmiserver.RmiMethodenInterface;

public class Client
{

  public static void main(String[] args)
  {
    ...

    try
    {
// Security Manager konfigurieren und setzen
      String policypath="bsp02120/rmiclient/rmipolicy_client.txt";
      System.setProperty("java.security.policy",policypath);
      if (System.getSecurityManager() == null)
        System.setSecurityManager(new SecurityManager());

// Remote Object suchen
      String rmiurl = "rmi://127.0.0.1:11099/rmidienst";
      System.out.println("\nEs wird folgender RMI-Dienst gesucht: "+rmiurl);
      Remote myRemoteObj = Naming.lookup(rmiurl);

// Cast auf das Remote Object absetzen, damit es vom Typ des Interface wird
      RmiMethodenInterface myRmiMethInterf;
      myRmiMethInterf= (RmiMethodenInterface) myRemoteObj;
      System.out.println("\nRMI-Dienst "+rmiurl+" wurde gefunden.");

// Methodenaufruf auf dem Server
      String zeit = myRmiMethInterf.getServerZeit();

      System.out.println("\nZeit-Abfrage auf Server ergab folgende Zeit: "+zeit);
```

```
    }
    catch (RemoteException e)
    {
      System.out.println("\nEine RemoteException ist aufgetreten.");
      e.printStackTrace();
    }
    catch (NotBoundException e)
    {
      System.out.println(
        "\nDer Dienst \"rmidienst\" ist noch nicht an die Registry gebunden.");
      e.printStackTrace();
    }
    catch (Exception e)
    {
      System.out.println("\nFehler beim Verbindungsaufbau zum Server.");
      e.printStackTrace();
    }
  }
}
```

Die wesentlichen Schritte dieser Verarbeitung sind wie folgt:

1. Die Methode lookup(...) der Klasse Naming holt sich beim RMI-Dienst (Port 11099) die Adresse des Remote Objektes mit dem Namen "rmidienst". Das Remote Objekt ist ein Objekt einer Klasse, die von der Klasse java.rmi.Remote abgeleitet ist. Die Methode lookup(...) liefert entsprechend die Adresse auf ein Objekt der Klasse Remote.
2. Damit die in dem Interface genannte Methode verfügbar wird, muss ein Cast auf das Interface durchgeführt werden. Dabei wird die Adresse des Remote Objektes von der Objekt-Variablen der Klasse Remote in eine Objekt-Variable des Interface kopiert.
3. Unter Verwendung der neuen Objekt-Variablen des Interface wird der Methodenaufruf der Servermethode durchgeführt.

Die Batch-Datei für den Start des Clients in Beispiel jpr02120 ist wie folgt programmiert:

```
@Echo off

rem
rem ggf. belegte Systemvariablen löschen:
rem

set classpath=.
set sourcepath=.
set bootclasspath=.
set extdirds=.

..\jdk1.8.0\bin\java -cp . bsp02120.rmiclient.Client

echo.
pause
```

Bei Ausführung dieser Batch-Datei startet der Client und ruft die vereinbarte Servermethode auf. Auf dem Server wird die Uhrzeit ausgegeben und an den Client zurückgeschickt, wo sie anschließend ebenfalls ausgegeben wird:

RMI Registrierung

Administrator: Eingabeaufforderung - $start-server

```
...\jpr02120ProdServer>$start-server
Die RMI-Registrierung wird gestartet ...

Es wird 3 Sekunden gewartet (Intitalisierung der RMI-Registrierung) ...

Der Server wird gestartet ...

Beispiel zur Verwendung eines Interface - RMI Server
Copyright 2006,2008 by W. Eikelberg (www.eikelberg.de)
Die URL des RMI-Dienstes lautet: rmi://127.0.0.1:11099/rmidienst
RMI-Dienst ist angemeldet, Server wartet auf Client-Zugriffe...

Zeitabfrage von Client - aktuelle Zeit: Wed 28 Sep 2016 12:49:03
```

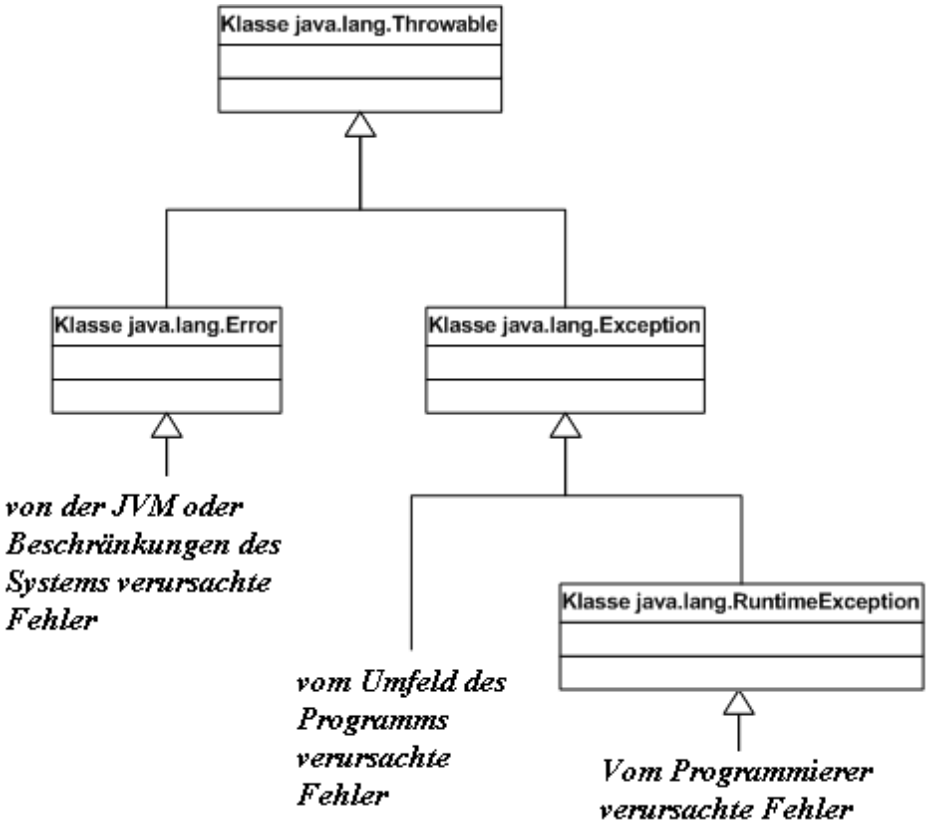
Administrator: Eingabeaufforderung - $start-client

```
...\jpr02120ProdClient>$start-client
Beispiel zur Verwendung eines Interface - RMI Client
Copyright 2006,2008 by W. Eikelberg (www.eikelberg.de)
Es wird folgender RMI-Dienst gesucht: rmi://127.0.0.1:11099/rmidienst
RMI-Dienst rmi://127.0.0.1:11099/rmidienst wurde gefunden.
Zeit-Abfrage auf Server ergab folgende Zeit: Wed 28 Sep 2016 12:49:03
Drücken Sie eine beliebige Taste . . .
```

2.9.5 Exceptions

2.9.5.1 Hierarchie der Fehler

Jeder Fehler zur Laufzeit eines Java-Programms ergibt ein Objekt einer Klasse, die den Fehler beschreibt. Diese Klassen sind hierarchisch geordnet. Die oberste Stufe der Hierarchie gibt zunächst an, ob es sich um einen Fehler des Programms handelt und ob dieser Fehler vom Programmierer verursacht wurde:



- Die Klasse `Throwable` ist die Superklasse aller Fehler-Klassen von Java, ihrerseits abgeleitet von der obresten Java-Klasse `java.lang.Object`. Jedes Objekt einer von Throwable abgeleiteten Klasse kann geworfen (`throw`, s.u.) bzw. gefangen (`catch`, s.u.) werden. Die Klasse Throwable hat genau zwei Subklassen, nämlich die Klassen `Error` und `Exception`.
- Objekte der Klasse `Error` bzw. Objekte von Klassen, die von `Error` abgeleitet sind, zeigen Fehler der Java Laufzeitumgebung (Virtuelle Maschine JVM) an. Bei diesen Fehlern handelt es sich um Mängel bzw. Fehler aufgrund von Einschränkungen der JVM. Objekte der Klasse `Error` bzw. Objekte von Klassen, die von `Error` abgeleitet sind, sollten nicht vom Programmierer gefangen werden, da sie nicht durch Programmierfehler ausgelöst werden.
- Objekte der Klasse `Exception` stellen Fehler des Anwendungsprogramms dar. Sie können durch falsche Benutzung des Programms sowie durch Programmierfehler verursacht worden sein. Die Menge der von Exception abgeleiteten Klassen ist zweigeteilt:
    - Von der Klasse `RuntimeException` treten im Rahmen der ordnungsgemäßen Verarbeitung der JVM auf. Sie beruhen in der Regel auf einem Programmierfehler.
    - Nicht von der Klasse `RuntimeException`, aber von der Klasse `Exception` abgeleitete Fehler beruhen auf Fehlersituationen im Umfeld des Programms.

Beispiel zu einer von Error abgeleiteten Fehlerklasse:

Das Programm im Beispiel jpr03062 ist ein Beispiel zur Thread-Programmierung. Es ist in der Lage, parallel ablaufende Prozesse (Threads) zu starten. Diese Threads müssen auch von der JVM verwaltet werden, weshalb die Anzahl der Threads beschränkt wird. Wenn die Anzahl der Threads zu hoch wird, kann es zu einem `java.lang.OutOfMemoryError` kommen (vgl. Kapitel 3.9.8). Dieser Fehler ist nicht vom Programmierer verursacht, sondern durch die Beschränktheit der Rechnerressourcen.

Beispiel zu einer von Exception - aber nicht von RuntimeException - abgeleiteten Fehlerklasse:

Soll während des Programmlaufs eine Datei geöffnet werden und wird diese Datei nicht gefunden, so tritt eine `java.io.IOException` auf. Diese Klasse ist direkt von der Klasse `Exception` abgeleitet und betrifft entsprechend eine Fehlersituation im Umfeld des Programms, die keinen Programmierfehler darstellt.

Beispiel zu einer von RuntimeException abgeleiteten Fehlerklasse:

Versucht ein Programmierer, auf ein Feld eines Arrays zuzugreifen, das es nicht gibt, weil der Index beim zugriff zu hoch oder zu niedrig gewählt wurde, wirft das Programm eine `java.lang.IndexOutOfBoundsException`. Die IndexOutOfBoundsException ist abgeleitet von der Klasse RuntimeException und stellt entsprechend einen Programmierfehler dar.

2.9.5.2 Fangen von Exceptions

Grundsätzlich gilt, dass bei Auftreten einer Exception die Bearbeitung des Programms abgebrochen wird und dass das entsprechende Programm abgebrochen wird. Diesen Programmabbruch kann man vermeiden, wenn man die Exception mit Hilfe von geeignetem Programmcode "fängt". Dies geschieht im Prinzip wie folgt:

- Man umgibt den Programmcode, bei dessen Abarbeitung die Exception auftreten kann, mit einem Block und codiert vor diesem Block das Schlüsselwort **try** (versuche).
- Hinter dem try-Block codiert man mindestens einen weiteren Block, vor den man das Schlüsselwort **catch** (fange) einfügt. Ferner muss man nach dem Schlüsselwort catch angeben, welche Exception man fangen will. Gibt man eine Exception an, so fängt man mit dem entsprechenden Block alle Exceptions dieses Typs sowie alle von dieser Exception abgeleiteten Exceptions.

Aus Kapitel 2.9.5.1 geht hervor, dass sich der Programmierer nicht um die von `Error` abgeleiteten Fehlerklassen kümmern muss, sondern nur um die von `Exception` abgeleiteten Klassen. Daher wird im Allgemeinen stets nur das Auslösen ("Werfen" vgl. 2.9.5.3) und Fangen von Exceptions betrachtet. Für das Fangen und Behandeln von Fehlersituationen gibt es in Java die folgenden beiden Schlüsselwörter:

- **catch** - ermöglicht das Behandeln (Fangen) von Exceptions
- **finally** - ermöglicht eine Verarbeitung des Programms unabhängig davon, ob im vorhergehenden try-Block eine Exception aufgetreten ist und ob diese Exception gefangen wurde.

Das Fangen von Exceptions ist in diesem Kurs bereits mehrfach behandelt worden. Ein Beispiel mit mehreren catch-Klauseln zeigt das Projekt jpr02121. In dem Beispielprogramm bsp02121.UrlSuche wird eine Web-Seite geöffnet und aus dem Quellcode der Web-Seite die erste Zeile gelesen:

```
package bsp02121;

import java.net.URL;
import java.net.MalformedURLException;
import java.io.*;
...

public class UrlSuche
{

  public static void main(String[] args)
  {
    ...
// Lesen der ersten Zeile aus einer Webseite
    URL url = null;
    try
    {
//    url = new URL("file:webseite//aufg0002.html");   // läuft fehlerfrei
      url = new URL("fyle:webseite//aufg0002.html");   // MalformedURLException
//    url = new URL("file:webseite//aufg0003.html");   // FileNotFoundException

      InputStream in = url.openStream();
      InputStreamReader isr = new InputStreamReader(in);
      BufferedReader ib = new BufferedReader(isr);

      String ein = ib.readLine();
      System.out.println("\nDie erste Zeile der Webseite lautet: \n"+ein);

      ib.close();
      isr.close();
      in.close();

    }
// Behandlung möglicher Exceptions
    catch (MalformedURLException e)
    {
      System.out.println("\nURL fehlerhaft angegeben!");
      e.printStackTrace();
    }
    catch (IOException e)
    {
```

```
      System.out.println("\nKein Zugriff auf die angegebene URL moeglich!");
      e.printStackTrace();
    }
    finally
    {
      System.out.println("\nDas Programm wird jetzt beendet.");
    }
  }
}
```

Beim Programmlauf können an folgenden Stellen Exceptions auftreten:

- Der Konstruktor `URL(...)` kann eine MalformedURLException werfen.
- Die Methode `openStream()` kann eine IOException werfen. Die von IOException abgeleitete FileNotFoundException tritt auf, wenn die angegebene Datei nicht existiert.
- Die Methode `readLine()` kann eine IOException werfen.

Nan beachte dabei, dass die Klasse java.net.MalformedURLException von der Klasse java.io.IOException abgeleitet ist. Es tritt also stets eine Exception vom Typ java.io.IOException oder einer von IOException abgeleiten Klasse auf. Der Programmcode enthält für den Spezialfall der MalformedURLException eine besondere Fehlerbehandlung.

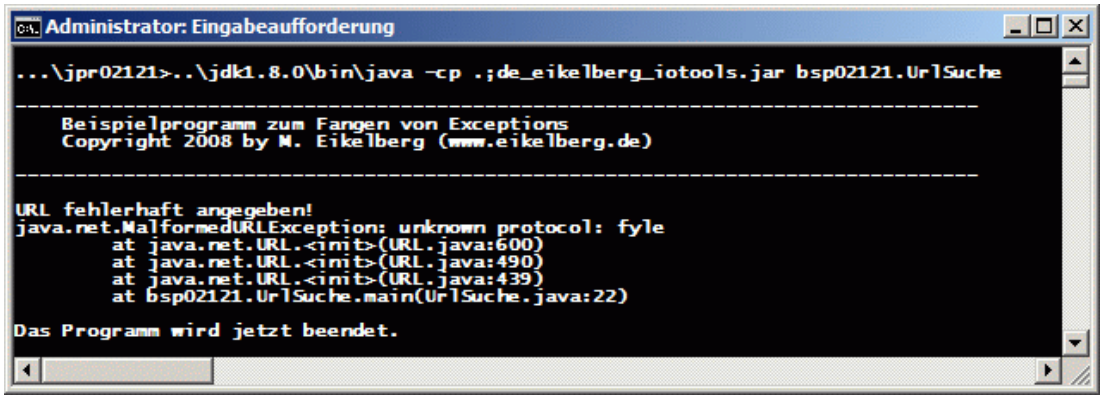
Beim Auftreten einer dieser Exceptions wird jetzt wie folgt vorgegangen:

- Die Bearbeitung der Anweisungen des try-Blocks wird abgebrochen.
- Es wird der `erste` catch-Block gesucht, in dessen catch-Anweisung die aufgetretene Exception *oder eine Klasse, von der die Exception direkt oder indirekt abgeleitet ist* behandelt. Die Anweisung des Blocks nach dieser catch-Anweisung werden ausgeführt.
- Die Anweisungen in dem Block nach der finally-Anweisung werden abgearbeitet.

Für den Fall, dass eine fehlerhafte URL angegeben wurde, also dass folgender Programmcode aktiv war

```
    url = new URL("fyle:webseite//aufg0002.html");   // MalformedURLException
```

ergibt sich eine MalformedURLException, die mit dem entsprechenden catch-Block behandelt behandelt wird, wonach dann noch der finally-Block abgearbeitet wird:



Grundsätzlich gilt für die Behandlung von Exceptions die Regel

catch or throw

Dies bedeutet, dass jede Exception vom Programmierer mit catch "gefangen" oder mit "geworfen" (vgl. 2.9.5.3) werden muss. Im Beispiel jpr02122 werden nicht alle auftretenden Exceptions gefangen, sondern lediglich die MalformedURLException. Mit der Klausel

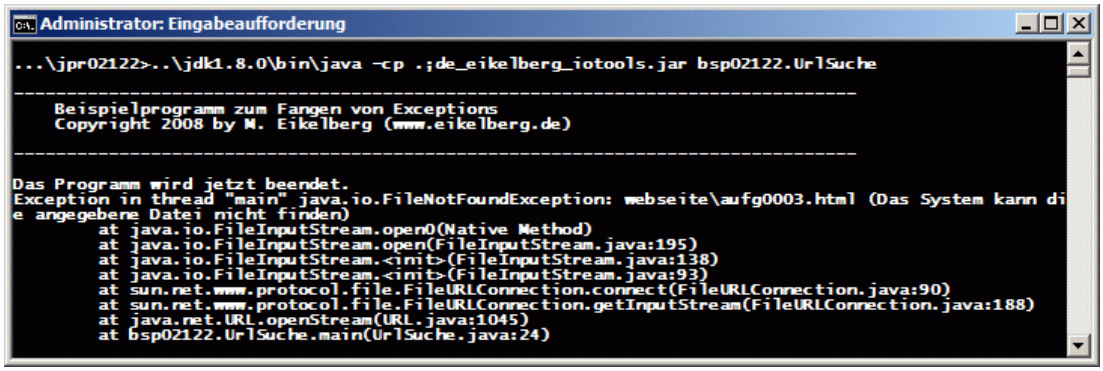
```
  throws java.io.IOException
```

im Kopf der main-Methode wird stattdessen angezeigt, dass (nicht gefangene) Exceptions vom Typ IOException auftreten können:

```
package bsp02122;

import ...

public class UrlSuche
{

  public static void main(String[] args)
  throws java.io.IOException
  {
    ...

// Lesen der ersten Zeile aus einer Webseite
    URL url = null;
    try
```

```
    {
      url = new URL("file:webseite//aufg0003.html");   // FileNotFoundException

      ...
    }
// Behandlung möglicher Exceptions
    catch (MalformedURLException e)
    {
      System.out.println("\nURL fehlerhaft angegeben!");
      e.printStackTrace();
    }
    finally
    {
      System.out.println("\nDas Programm wird jetzt beendet.");
    }
  }
}
```

Durch diesen Programmcode werden alle Exceptions, die nicht vom Typ MalformedURLException sind bzw. die nicht vom Typ einer von MalformedURLException abgeleiteten Klasse sind, geworfen und müssen von dem Programm behandelt werden, dass die entsprechende Methode ruft. Da es sich hier um die main-Methode handelt, müssen diese Exceptions von der JVM behandelt werden. In diesem Fall wird die Bearbeitung des Programms abgebrochen und die JVM gibt den StackTrace zur Exception auf. *Man beachte, dass trotz Programmabbruch noch der finally-Block abgearbeitet wird, wie der folgende Bildschirmausdruck zeigt:*



Man sieht, dass zunächst die Konsolausgabe des finally-Blocks erfolgt und anschließend der StackTrace von der JVM in der Konsole ausgegeben wird.

2.9.5.3 Werfen von Exceptions

Ein Programmierer kann

- auftretende Exceptions nicht behandeln und stattdessen werfen sowie
- selber Exceptions erzeugen und werfen.

In beiden Fällen muss jeder Programmierer, der die ensprechende Methode ruft, wieder entweder die jeweilige Exception fangen oder werfen.

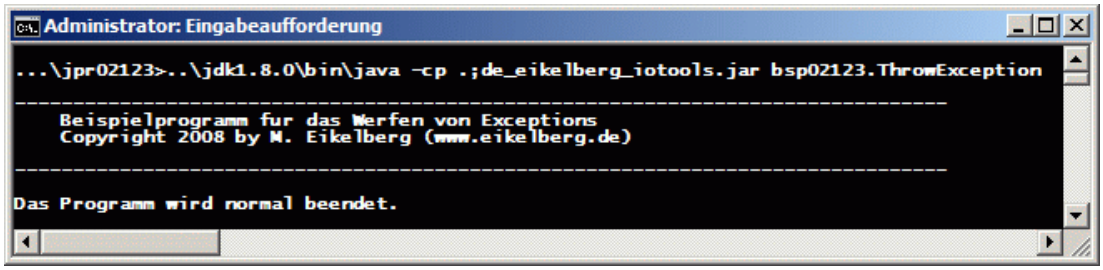
Erzeugen eines Exception-Objektes einer vorhandenen Exception-Klasse

Beispiel jpr02123 zeigt das Erzeugen eines Exception-Objektes einer vorhandenen Exception-Klasse - im vorliegenden Fall der Exception-Klasse `java.lang.Exception`:

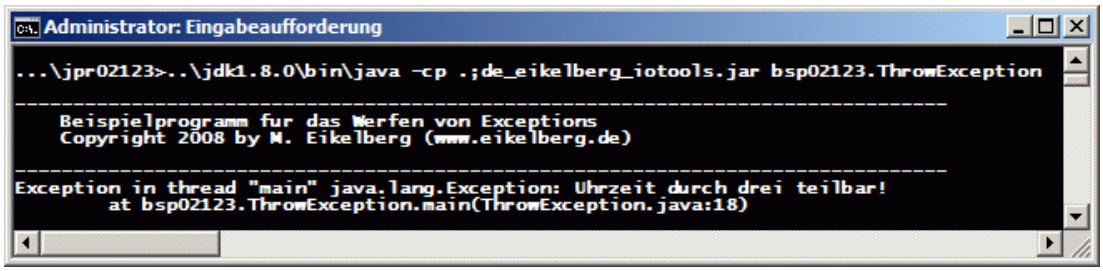
```
package bsp02123;

...

public class ThrowException
{

  public static void main(String[] args)
  throws Exception
  {
    ...

// Uhrzeit (msec. nach 1.1.1970) feststellen und Exception werfen,
// wenn die Uhrzeit durch 3 Teilbar ist
    if (System.currentTimeMillis() % 3 == 0)
      throw new Exception("Uhrzeit durch drei teilbar!");
```

```
// Schlussmeldung
    System.out.println("\nDas Programm wird normal beendet.");
  }
}
```

Tritt eine durch drei teilbare Uhrzeit auf, so wird das Programm mit einer Exception abgebrochen und die Schlussmeldung des Programms wird nicht ausgegeben:



Tritt dagegen keine durch 3 teilbare Uhrzeit auf, wird das Programm ohne Exception mit der Schlussmeldung beendet:



Erzeugen einer Exception und fangen der erzeugten Exception in der rufenden Methode

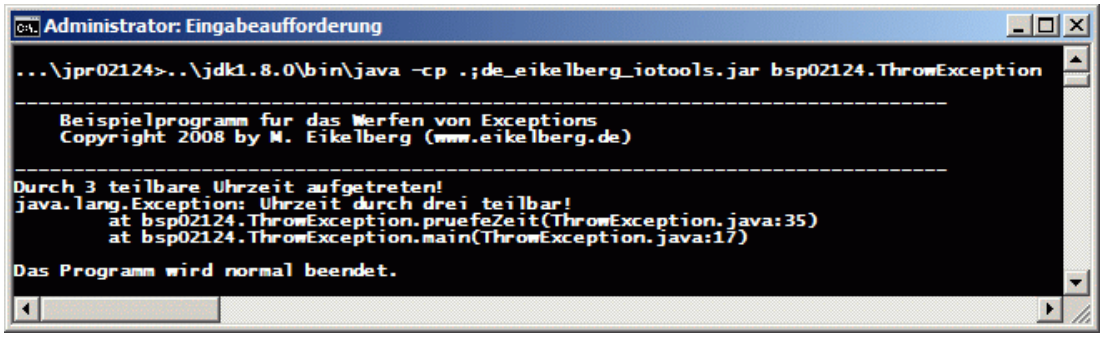
Wird die Exception nicht in der main-Methode geworfen, sondern in einer untergeordneten Methode, so kann diese Exception in der übergeordneten Methode gefangen werden, wie das Beispiel jpr02124 zeigt:

```
package bsp02124;

...

public class ThrowException
{

  public static void main(String[] args)
  throws Exception
  {
    ...

    try
    {
      pruefeZeit();
    }
    catch (Exception e)
    {
      System.out.println("Durch 3 teilbare Uhrzeit aufgetreten!");
      e.printStackTrace();
    }

//  Schlussmeldung
    System.out.println("\nDas Programm wird normal beendet.");
  }

  private static void pruefeZeit()
  throws Exception
  {
//  Uhrzeit (msec. nach 1.1.1970) feststellen und Exception werfen,
//  wenn die Uhrzeit durch 3 Teilbar ist
     if (System.currentTimeMillis() % 3 == 0)
       throw new Exception("Uhrzeit durch drei teilbar!");
  }
}
```

Die Methode pruefeZeit() wirft gegebenenfalls eine Exception, weshalb im Kopf der Methode die Klausel

```
  throws Exception
```

codiert wurde. Hierdurch weis der Compiler, dass in der rufenden Methode - hier die main-Methode - eine Exception vom Typ `java.lang.Exception` entweder gefangen oder geworfen werden muss. Bei Auftreten der Exception gibt das Programm dann die im catch-Block codierte Meldung sowie den StackTrace. Im StackTrace wird jetzt auch verzeichnet, dass die Exception in der gerufenen Methode pruefeZeit() aufgetreten ist:



Codieren individueller Exception-Klassen

Zur individuellen Kennzeichnung selber geworfener Exceptions ist es möglich, eigene Exception-Klassen zu codieren. Dies ist in sehr einfacher Weise möglich, wie das Beispiel jpr02125 zeigt. In der Klasse bsp02125.ThrowException ist nicht nur die gleichnamige Klasse codiert, sondern auch die individuelle von java.lang.Exception abgeleitete Klasse bsp02125.ZeitException:

```
package bsp02125;

...

public class ThrowException
{

  public static void main(String[] args)
  throws Exception
  {
    ...

    try
    {
      pruefeZeit();
    }
    catch (ZeitException e)
    {
      System.out.println("Durch 3 teilbare Uhrzeit aufgetreten!");
      e.printStackTrace();
    }

//  Schlussmeldung
    System.out.println("\nDas Programm wird normal beendet.");
  }

  private static void pruefeZeit()
  throws ZeitException
  {
//  Uhrzeit (msec. nach 1.1.1970) feststellen und Exception werfen,
//  wenn die Uhrzeit durch 3 Teilbar ist
     if (System.currentTimeMillis() % 3 == 0)
       throw new ZeitException("Uhrzeit durch drei teilbar!");
  }
}

class ZeitException extends Exception
{
  private static final long serialVersionUID = 710L;

  public ZeitException() { }
  public ZeitException(String meldung)
  {
    super(meldung);
  }
}
```

In diesem Programmcode wird die geworfene Exception als individuell programmierte Exception vom Typ `bsp02125.ZeitException` gekennzeichnet, die von der Klasse `java.lang.Exception` abgeleitet ist. Hierdurch kann sie bei der Fehlerbehandlung durch catch und throw genauer behandelt werden:

- Die Klasse ZeitException stellt einen parameterlosen Konstruktor bereit, der - wie stets automatisch bei parameterlosen Konstruktoren - den super-Konstruktor der Klasse Exception aufruft (vgl. 2.9.2.5). Ferner stellt die Klasse ZeitException eine parameterbehafteten Konstruktor bereit, der den mit dem gleichen Paramter behafteten Konstruktor der übergeordneten Klasse Exception aufruft.
- Beim Werfen der Exception wird ein Objekt der Klasse `ZeitException` erzeugt.
- Im Kopf der Methode pruefeZeit() wird angezeigt, das gegebenenfalls ein Objekt der Klasse `ZeitException` geworfen wird.
- Beim Methodenaufruf wird das Fangen von Exceptions vom Typ `ZeitException` durchgeführt. Hierdurch können die selber geworfenen Exceptions sicher von anderen gegebenenfalls auftretenden Exceptions unterschieden werden.

Beim Programmlauf wird jetzt sichtbar, dass eine individuell codierte Exception vom Typ `bsp02125.ZeitException` geworfen wurde, die in einer Methode ausgelöst wurde und in der main-Methode gefangen wurde. Da die Exception gefangen wurde, wird das Programm normal beendet:

```
Administrator: Eingabeaufforderung
...\jpr02125>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02125.ThrowException
--------------------------------------------------------------------------------
    Beispielprogramm fur das Werfen von Exceptions
    Copyright 2008 by M. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------
Durch 3 teilbare Uhrzeit aufgetreten!
bsp02125.ZeitException: Uhrzeit durch drei teilbar!
        at bsp02125.ThrowException.pruefeZeit(ThrowException.java:35)
        at bsp02125.ThrowException.main(ThrowException.java:17)

Das Programm wird normal beendet.
```

2.9.5.4 Logging

Zur Fehleranalyse ist es in der Regel notwendig, Zwischenstände der Programmabarbeitung zu protokollieren und gefangene Exceptions mit einem Kommentar versehen auszugeben. Ein solches Vorgehen nennt man **Logging**. Logging kann in einfacher Weise realisiert werden durch Einfügen von Anweisungen zu Ausgabe von Meldungen in die Konsolbox oder eine (Log-)Datei. Es gibt jedoch auch eine elegante Methode zur Implementierung von Logging in ein Programm, bereitgestellt von der Klasse **java.util.logging.Logger**. Diese Klasse stellt ein umfangreiches System zur Verfügung, das ein sehr filigranes Logging ermöglicht.

- Der Standard-Logger **Logger.global** ist ein Objekt der Klasse Logger, der für einfaches Logging verwendet werden kann. Er wird initialisiert durch den Methodenaufruf

  Logger myLogger = Logger.getLogger("global")

  und gibt seine Meldungen in der Konsolbox aus.
- Jeder Logger kennt folgende Stufen der Wichtigkeit von Meldungen, die als Klassenvariablen der Klasse **java.util.logging.Level** deklariert sind:
  - **Level.ON** - Logging wird ausgeschaltet
  - **Level.OFF** - Logging wird eingeschaltet und standardmäßig ab Stufe INFO durchgeführt
  - **Level.FINEST**
  - **Level.FINER**
  - **Level.FINE**
  - **Level.CONFIG**
  - **Level.INFO**
  - **Level.WARNING**
  - **Level.SEVERE**
- Mit der Methode **myLogger.setLevel(...)** kann die Stufe eingestellt werden, ab der Log-Meldungen ausgegeben werden sollen.
- Logmeldungen der o.g. Level werden von folgenden Instanzmethoden der Klasse Logger ausgegeben:
  - **myLogger.finest(String)**
  - **myLogger.finer(String)**
  - **myLogger.fine(String)**
  - **myLogger.config(String)**
  - **myLogger.info(String)**
  - **myLogger.warning(String)**
  - **myLogger.severe(String)**
- Auch wenn eine Stufe feiner als Level.INFO eingestellt wird, erfolgt das Logging standardmäßig nur ab Level.INFO aufwärts. Andererseits werden die Stufen Level.FINEST bis Level.CONFIG für die Log-Meldungen empfohlen, die für die Diagnose während der Programmentwicklung interssant sind, jedoch nicht für den Anwender des Programms. Dieser sollte nur Meldungen der Stufen Level.INFO bis Level.SEVERE erhalten.
- Durch folgende Methode kann das Logging für die Stufen Level.FINEST bis Level.CONFIG erreicht werden:

- Man kopiere aus dem JDK die Datei `jre/lib/logging.properties` an einen geeigneten Ort im Projektverzeichnis zum Paket, also zum Beispiel nach `config/logging.properties`
- Man gebe beim Start des Programmms folgende Option an:

  ```
  -Djava.util.logging.config.file="config/logging.properties"
  ```

- Man ändere die nachfolgenden beiden Einträge in der Datei config/logging.properties so ab, dass der jeweils eingestellte minimale Level statt INFO jetzt FINEST wird:

  ```
  ...
  .level= FINEST
  ...
  java.util.logging.ConsoleHandler.level = FINEST
  ...
  ```

  Hierdurch wird sowohl die Standard-Einstellung für das Logging (.level) als auch die spezielle Einstellung für das Logging des Standdard-Loggers in die Konsole auf mindeststufe Level.FINEST gesetzt.

- Logging für Exceptions ist mit Hilfe der folgenden Instanzmethode der Klasse Logger möglich:
  - **void log(Level l, String message, Throwable t)**
- Logger können - mit einer Namensstruktur wie Paketnamen - wie folgt benannt werden:

  Logger eikLogger = Logger.getLogger("de.eikelberg.logging")

  Für einen solchen benannten Logger kann die Mindstestufe für das Logging in der Konfigurationsdatei logging.properties durch Einfügen einer Zeile eingestellt werden:

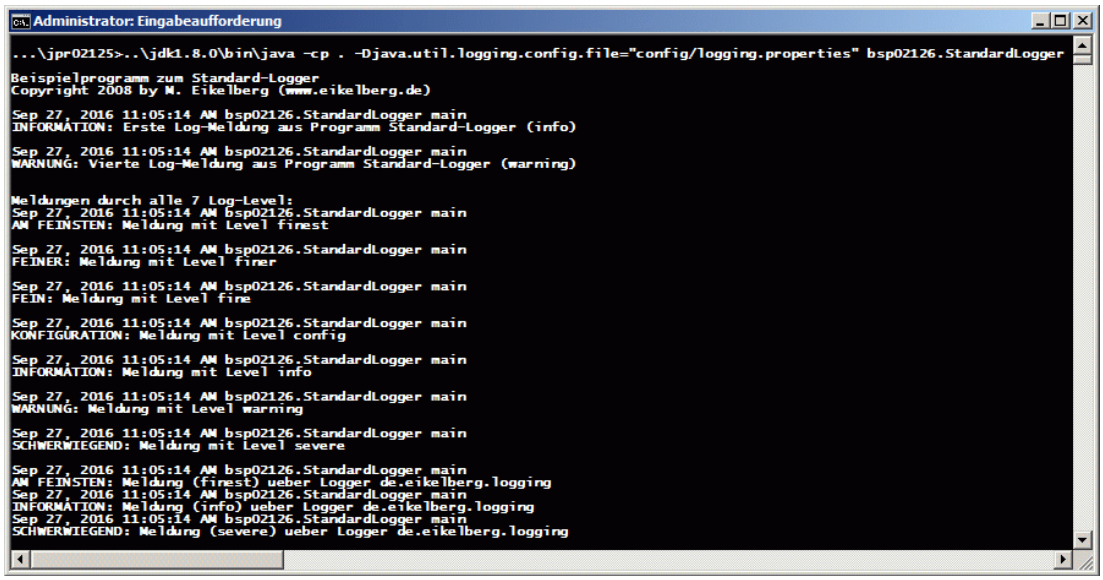
  ```
  de.eikelberg.logging.level=FINEST
  ```

Beispiel jpr02126 zeigt die Benutzung des Loggers bei Ausgabe in die Konsole:

```
package bsp02126;

import java.util.logging.Logger;
import java.util.logging.Level;
...

public class StandardLogger
{

  public static void main(String[] args)
  throws Exception
  {
    ....

// Logger initialisieren
    Logger myLogger = Logger.getLogger("global");

// Log-Meldung ausgeben
    myLogger.info(
        "Erste Log-Meldung aus Programm Standard-Logger (info)\n");

// Logging deaktivieren
    myLogger.setLevel(Level.OFF);

//  Log-Meldung ausgeben
    myLogger.info(
        "Zweite Log-Meldung aus Programm Standard-Logger (info)\n");

//  Logging ab Log-Level INFO aktivieren
    myLogger.setLevel(Level.WARNING);

//  Log-Meldung ausgeben, mit Fehlerstufe WARNING
    myLogger.info(
        "Dritte Log-Meldung aus Programm Standard-Logger (info)\n");

//  Log-Meldung ausgeben, mit Fehlerstufe WARNING
    myLogger.warning(
        "Vierte Log-Meldung aus Programm Standard-Logger (warning)\n");

    if (System.currentTimeMillis() % 3 == 0)
    {
//    Log-Meldung ausgeben, mit Fehlerstufe SEVERE
      myLogger.log(
          Level.SEVERE,
          "Fuenfte Log-Meldung aus Programm Standard-Logger (log/severe)\n",
```

```
            new Exception("Absturz von StandardLogger.java")
        );
    }

// Logging durch alle Stufen
    System.out.println("\nMeldungen durch alle 7 Log-Level:");
    myLogger.setLevel(Level.FINEST);
    myLogger.finest("Meldung mit Level finest\n");
    myLogger.finer("Meldung mit Level finer\n");
    myLogger.fine("Meldung mit Level fine\n");
    myLogger.config("Meldung mit Level config\n");
    myLogger.info("Meldung mit Level info\n");
    myLogger.warning("Meldung mit Level warning\n");
    myLogger.severe("Meldung mit Level severe\n");

    ...
  }
}
```

Beim Programmlauf erscheinen die zweite und dritte Log-Meldung nicht, da das Logging ausgeschaltet ist bzw. erst ab Level.WARNING erfolgt. Die Meldungen fünf erscheint je nachdem, ob die Zeit in msec. seit dem 1.1.1970 durch 3 teilbar ist oder nicht:



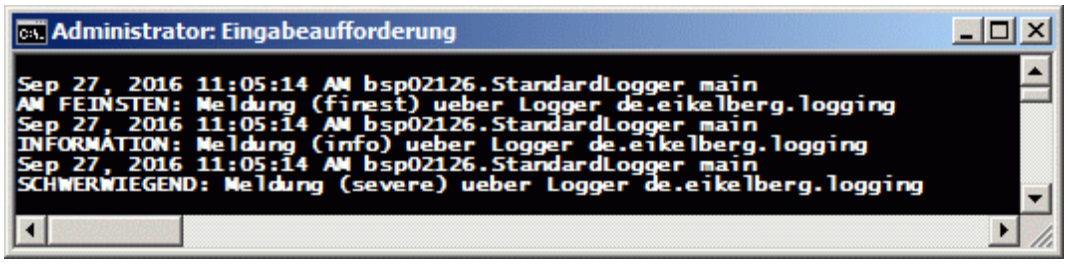
Standardmäßig erfolgt die Ausgabe des Loggings in die Konsolbox. Zusätzlich kann jedoch auch eine Protokollierung in Log-Dateien erfolgen. Hierzu muss beim Logger ein `FileHandler` registriert werden, der für die Ausgabe der Log-Meldungen in die Log-Datei sorgt. Der FileHandler hat standardmäßig wieder den minimalen Log-Level Level.INFO eingestellt. Diese Standard-Konfiguration muss gegebenenfalls angepasst werden. Der folgende Abschnitt aus dem Programm bsp02126.StandardLogger zeigt das Vorgehen für ein Logging in Log-Dateien (zusätzlich zum Logging in der Konsolbox)

```
// Benannten Logger mit File Handler anlegen
    Logger eikLogger = Logger.getLogger("de.eikelberg.logging");
// FileHandler anlegen mit Pattern für Dateienamen, je Datei max 100 Zeilen,
// 2 routierende Dateien
    FileHandler logDateiHandler = new FileHandler("logs/StandardLogger.log",100,2);
// Log-Level setzen
    logDateiHandler.setLevel(Level.FINEST);
// FileHandler beim Logger registrieren
    eikLogger.addHandler(logDateiHandler);
// Log-Meldung ausgeben
    eikLogger.finest("Meldung (finest) ueber Logger de.eikelberg.logging");
    eikLogger.info("Meldung (info) ueber Logger de.eikelberg.logging");
    eikLogger.severe("Meldung (severe) ueber Logger de.eikelberg.logging");
// Log-Datei schließen
    logDateiHandler.flush();
    logDateiHandler.close();
```

In der Datei `logging.properties` wurde für den Logger die Stufe Level.FINEST eingetragen.

```
de.eikelberg.logging.level=FINEST
```

Entsprechend erscheinen alle drei Log-Meldungen in der Konsolbox:



In einer Log-Datei `logs/StandardLogger.log` sind ferner alle drei Log-Meldungen in XML-Code gespeichert, der folgende Anzeige in einem Browser ergibt:

```
2016-09-27T11:10:08
1474967408672
10
de.eikelberg.logging
FINEST
bsp02126.StandardLogger
main
1
Meldung (finest) ueber Logger de.eikelberg.logging


2016-09-27T11:10:08
1474967408672
11
de.eikelberg.logging
INFO
bsp02126.StandardLogger
main
1
Meldung (info) ueber Logger de.eikelberg.logging


2016-09-27T11:10:08
1474967408672
12
de.eikelberg.logging
SEVERE
bsp02126.StandardLogger
main
1
Meldung (severe) ueber Logger de.eikelberg.logging
```

2.10 Stream-Konzept

2.10.1 Streams, Reader, Writer und Buffer bei Konsol-IO

2.10.1.1 Der InputStream System.in

Alle Ein- und Ausgabevorgänge (IO-Vorgang) erfolgen gemäß dem Java Stream-Konzept. Dies bedeutet, dass jeder Ein- und Ausgabevorgang in einer unstrukturierten Folge von Schriftzeichen erfolgt, von denen einige besondere Bedeutung haben wie zum Beispiel Zeilenwechsel oder Dateiende. Das Java Stream-Konzept hat Vor- und Nachteile:

- Jeder Ein- und Ausgabevorgang erfolgt nach dem gleichen Schema unabhängig, ob ein Zugriff auf Tastatur und Konsole, Datei, URL im Internet oder TCP Socket durchgeführt wird.
- Jeder Stream kann mit jeder Art von Ein- oder Ausgabevorgang verknüpft werden.
- Da der Zugriff auf einen Stream ein Vorgang programmiertechnisch niedriger Ebene ist, ist er aufwändig und wenig komfortabel.
- Da der Zugriff auf einen Stream ein Vorgang programmiertechnisch niedriger Ebene ist, ist er flexibel und universell sowie fein steuerbar.

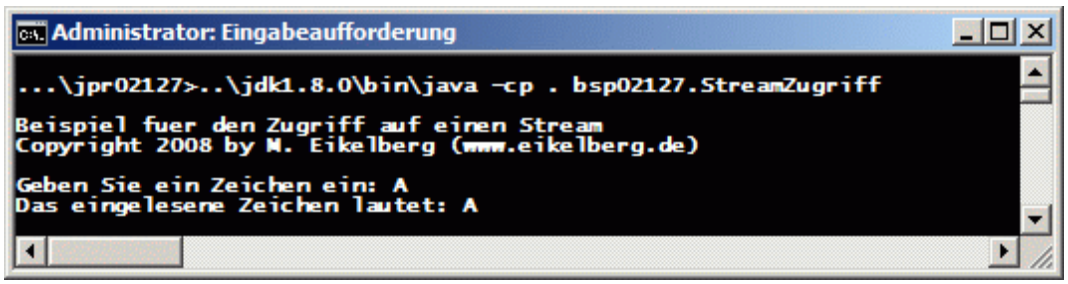
Ein lesender oder schreibender Zugriff auf einen Ein- oder Ausgabestrom erfolgt standardmäßig über Methoden mit den Namen

- **int read()** - liefert die Dezimalzahl der Verschlüsselung des gelesenen Zeichens gemäß Codierung des Betriebssytems zum jeweiligen Einlesevorgang
- **void write(int c)** - übernimmt als Parameter die Dezimalzahl der Verschlüsselung des gelesenen Zeichens gemäß Codierung des Betriebssytems zum jeweiligen Ausgabevorgang und schreibt das entsprechende Zeichen in den Ausgabestrom

Ansatzpunkt für die Codierung eines IO-Vorgangs gemäß dem Stream-Konzept ist der Zugriff auf den passenden Ein- oder Ausgabestrom. Beispiel jpr02129 zeigt den Zugriff auf einen InputStream am Beispiel des Eingabestroms `System.in`:

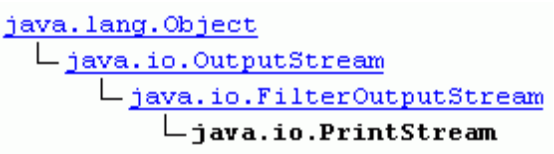
```
package bsp02127;

import java.io.*;

public class StreamZugriff
{

  public static void main(String[] args)
  {
    ...

    InputStream tastaturEin = System.in;
//  Ein PrintStream ist abgeleitet von OutputStream, liefert insbesondere
//  keien Exceptions und flusht automatisch den Output
    PrintStream konsolAus = System.out;

    ...

// Einlesevorgang gemäß Stream-Konzept
    int zeichen = 0;
    try
    {
// einzulesendes Zeichen holen
      zeichen = tastaturEin.read();
// Return und Newline lesen und wegwerfen
      tastaturEin.read();
      tastaturEin.read();
    }
    catch (IOException e)
    {
      konsolAus.println("Fehler beim Einlesevorgang!");
      e.printStackTrace();
    }

// Ausgabevorgang gemäß Stream-Konzept, jedoch ohne try und catch, da
// System.out ein PrintStream ist:
    ...
    konsolAus.write(zeichen);
    ...
  }
}
```

Beim Programmlauf wird genau ein Zeichen eingelesen und als int-Zahl gespeichert. Diese int-Zahl wird beim Ausgabevorgang mit der Methode write(int) wieder ausgegeben und dabei automatisch gemäß der Codierungstabelle des Systems zum angegebenen Vorgang in ein Schriftzeichen umgesetzt:



2.10.1.2 Der PrintStream System.out

Beispiel jpr02127 zeigt auch, dass System.out kein Objekt der Klasse OutputStream ist, sondern Objekt der von OutputStream abgeleiteten Klasse PrintStream. Diese Klasse ist ein erstes Beispiel für Klassen, die von InputStream oder OutputStream abgeleitet sind und die es ermöglichen, die Ein- und Ausgabe in vereinfachter Form durchzuführen.



Die Klasse PrintStream erweitert die Funktionalität der Klasse OutputStream in folgender Art und Weise:

- Ein PrintStream ist in der Lage, verschiedene Datentypen (mit den überladenen Methoden print(...) und println(...) komfortabel auszugeben. System.out.println(...) ist eine solche Menge von Methoden.
- Methoden der Klasse PrintStream werfen keine Exceptions, was die Codierung vereinfacht. Stattdessen kann mit der Methode `boolean checkError()` geprüft werden, ob ein Fehler erfolgt ist.
- Bei der Ausgabe in einen OutputStream werden alle Zeichen zunächst in einen Ausgabe-Puffer geschrieben, der in der Regel erst dann an das Ausgabemedium weitergegeben wird, wenn der Puffer voll ist oder wenn der Programmierer dies mit der Methode `flush()` anfordert. Die print-Methoden eines PrintStream führen das Entleeren des Puffers (den Flush) automatisch nach jedem Schreib-Befehl durch.
- Die Methoden println(...) fügen an jede Ausgabe das Zeichen Newline ('\n') an, so dass zum Abschluss der Ausgabe ein Zeilenwechsel erfolgt.

Im Beispielprogramm jpr02127 wird genutzt, dass System.out ein Objekt der Klasse PrintStream ist und damit sowohl die API der Klasse OutputStream - Methode `write(int)` - als auch die zusätzliche API der Klasse PrintStream bereitstellt:

```
package bsp02127;

import java.io.*;

public class StreamZugriff
{

  public static void main(String[] args)
  {
    ...

//  Ein PrintStream ist abgeleitet von OutputStream, liefert insbesondere
//  keien Exceptions und flusht automatisch den Output
    PrintStream konsolAus = System.out;

    konsolAus.print("Geben Sie ein Zeichen ein: ");

    ...

// Ausgabevorgang gemäß Stream-Konzept, jedoch ohne try und catch, da
// System.out ein PrintStream ist. Methoden print(..) aus der API von
// PrintStream und write geerbt aus der API von OutputStream:
    konsolAus.print("Das eingelesene Zeichen lautet: ");
    konsolAus.write(zeichen);
    konsolAus.println();
  }
}
```

Hinweis: Abgesehen von den Streams System.in und System.out muss jeder Stream möglichst früh wieder mit der Methode close() geschlossen werden. Dies ist zwar für System.in und System.out auch möglich, in der Regeln aber nicht sinvoll und auch nicht notwendig.

2.10.1.3 InputStreamReader und OutputStreamWriter

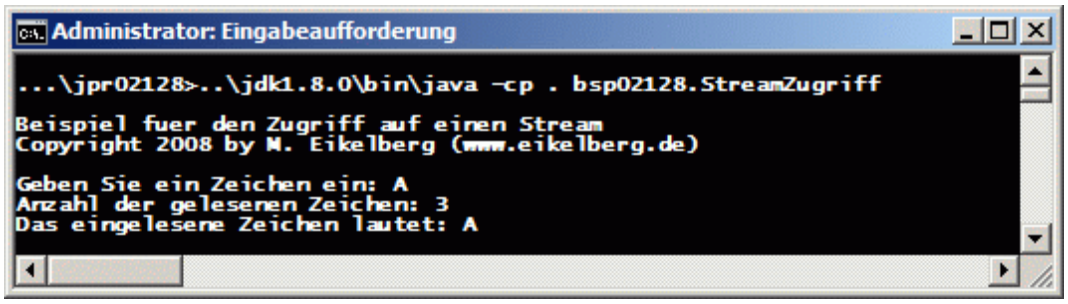
Die Klassen InputStreamReader und OutputStreamWriter sorgen für die Umwandlung der von den Klassen InputStream gelesenen bzw. OutputStream zu schreibenden int-Daten in Schriftzeichen gemäß der Codierung des verwendeten Betriebssystems. Die lesende Methode read(char[],int,int) sowie die schreibende Methode write(char[],int,int) der Klassen InputStreamReader und OutputStreamWriter verwenden entsprechend einen char-Array zur Speicherung bzw. beim Lesen von gespeicherten Eingaben. Erweitert man Beispiel jpr02127 um die Klassen InputStreamReader und OutputStreamWriter, so erhält man das folgende Programm bsp02128.StreamZugriff, das jetzt den eingelesenen Buchstaben in einem char-Array speichert:

```
package bsp02128;

import java.io.*;

public class StreamZugriff
{

  public static void main(String[] args)
  {
    ...

    InputStream tastaturEin = System.in;
    InputStreamReader tastaturLeser = new InputStreamReader(tastaturEin);
//  Ein PrintStream ist abgeleitet von OutputStream und wird hier - für das
//  Beispiel nach OutputStream gecastest.
    OutputStream konsolAus = System.out;
    OutputStreamWriter konsolSchreiber = new OutputStreamWriter(konsolAus);

    System.out.print("Geben Sie ein Zeichen ein: ");

// Einlesevorgang gemäß Stream-Konzept
    char[] zeichen = new char[3];
    int anzahl = 0;
    try
    {
// Einzulesendes Zeichen holen und in den char-Array zeichen speichern.
// Dabei Return und Newline mit lesen
      anzahl = tastaturLeser.read(zeichen,0,3);
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Einlesevorgang!");
      e.printStackTrace();
    }
    System.out.println("Anzahl der gelesenen Zeichen: "+anzahl);

// Ausgabevorgang gemäß Stream-Konzept, jedoch ohne try und catch, da
// System.out ein PrintStream ist. Methoden print(..) aus der API von
// PrintStream und write geerbt aus der API von OutputStream:
    System.out.print("Das eingelesene Zeichen lautet: ");
    try
    {
      konsolSchreiber.write(zeichen,0,1);
      konsolSchreiber.flush();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Ausgabevorgang!");
      e.printStackTrace();
    }
    System.out.println();

// Schließen von Reader und Writer
    try
    {
      tastaturLeser.close();
      konsolSchreiber.close();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Schließen eines Readers oder Writers!");
      e.printStackTrace();
    }
```

```
  }
}
```

Hinweis: Jeder InputStreamReader und OutputStreamWriter muss mit der Methode close() geschlossen werden!

Man sieht, dass das Flushen des Writers immer noch erforderlich ist. Ansonsten kann es vorkommen, dass die Ausgabe des eingelesenen Buchstabens beim Programmlauf fehlt. Der Screenshot zeigt, dass dieser Effekt beim Durchlauf des Programms bsp02128.StreamZugriff nicht auftritt:



2.10.1.4 BufferedReader und BufferedWriter

Die Arbeit mit den char-Arrays sowie die Behandlung von Newline und Return bei Verwendung der Klassen InputStreamReader und OutputStreamWriter sind etwas umständlich, so dass es empfehlenswert ist, die Klassen InputStreamReader und OutputStreamWriter mit einem BufferedReader bzw. BufferedWriter zu umhüllen. Hierdurch wird es möglich, die Ein- und Ausgabe zeilenweise und im String-Format durchzuführen, wie das Beispiel jpr02129 zeigt:

```
package bsp02129;

import java.io.*;

public class StreamZugriff
{

  public static void main(String[] args)
  {
    ...

    InputStream tastaturEin = System.in;
    InputStreamReader tastaturLeser = new InputStreamReader(tastaturEin);
    BufferedReader tastaturPuffer = new BufferedReader(tastaturLeser);

// Ein PrintStream ist abgeleitet von OutputStream und wird hier - für das
// Beispiel nach OutputStream gecastest.
    OutputStream konsolAus = System.out;
    OutputStreamWriter konsolSchreiber = new OutputStreamWriter(konsolAus);
    BufferedWriter konsolPuffer = new BufferedWriter(konsolSchreiber);

    System.out.print("Geben Sie eine Zeile ein: ");

// Lesen einer Zeile in einen String
    String zeile = "";
    try
    {
      zeile = tastaturPuffer.readLine();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Einlesevorgang!");
      e.printStackTrace();
    }

// Ausgabe eines Strings, eines Zeilenwechsel sowie Flushen des Puffers
    System.out.print("Das eingelesene Zeile lautet: ");
    try
    {
      konsolPuffer.write(zeile);
      konsolPuffer.newLine();       // Zeilenwechsel
      konsolPuffer.flush();
    }
    catch (IOException e)
```

```
    {
      System.out.println("Fehler beim Ausgabevorgang!");
      e.printStackTrace();
    }

// Schließen von Reader und Writer sowie beiden Puffern
    try
    {
      tastaturPuffer.close();
      tastaturLeser.close();
      konsolPuffer.close();
      konsolSchreiber.close();
    }
    catch (IOException e)
    {
      System.out.println(
        "Fehler beim Schließen Readers, Writers oder Puffers!");
      e.printStackTrace();
    }
  }
}
```

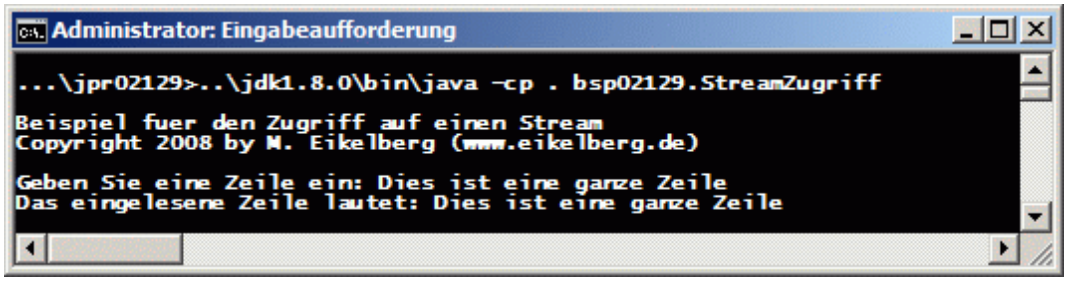
Man sieht, dass es auch bei der Verwendung eines Puffers für die Ausgabe notwendig ist, diesen zu Flushen. Ferner zeigt das Beispiel, dass Puffer ebenfalls mit der Methode close() geschlossen werden müssen.

> **Hinweis:** Die Ein- und Ausgabe über das String-Format ist fundamental fü die Java-Programmierung. Sie ist sowohl beim Lesen und Schreiben eines Konsolprogramms als auch bei Programmen mit grafischer Benutzeroberfläche sowie bei der Dateiverarbeitung die optimale von standard Java-Klassen unterstützte Methode!

Die Methoden der Klassen InputStream, InputStreamReader, OutputStream und OutputStreamReader sind also in der Regel für den Programmierer eines Anwendungsprogramms nicht interessant. Vielmehr erfolgt der Ein- und Ausgabevorgang normalerweise mit den folgenden Methoden der Klassen BufferedReader und BufferedWriter:

- **readLine() - Instanzmethode der Klasse BufferedReader** - liest eine Eingabe bis einschließlich abschließendem Return-Zeichen und Newline-Zeichen. Gibt die Eingabe in einem String zurück, in dem das Return-Zeichen und Newline-Zeichen jedoch nicht gespeichert sind.
- **write(String) - Instanzmethode der Klasse BufferedWriter** - schreibt den Inhalt der als Parameter übergebenen String-Variablen *ohne abschließenden Zeilenwechsel* .
- **newLine() - Instanzmethode der Klasse BufferedWriter** - schreibt einen Zeilenwechsel in dem für das jeweilige Betriebssystem gültigen Format.
- **flush() - Instanzmethode der Klasse BufferedWriter** - entleert den Ausgabepuffer und schreibt die noch im Puffer vorhandenen Zeichen in das Medium zum Ausgabestrom.

Das Programm bsp02129.StreamZugriff ist anders als die Beispiele jpr02127 und jpr02128 jetzt in der Lage, Zeichenketten unbekannter Länge - die auch Leerzeichen enthalten können - einzulesen und auszugeben:



2.10.2 Dateizugriff

Die in Kapitel 2.10.1 erklärte Methode für den Zugriff auf einen Ein- oder Ausgabestrom über einen BufferedReader oder BufferedWriter gilt uneingeschränkt auch für einen Dateizugriff. Der einzige Unterschied zur Ein- und Ausgabe beim Konsolzugriff besteht in der Instanziierung eines Objektes zu den Eingabeströmen aus einer Datei bzw. in eine Datei:

- Den Eingabestrom für das Lesen einer Datei erhält man als Objekt der Klasse `FileInputStream`
- Den Ausgabestrom für das Schreiben in eine Datei erhält man als Objekt der Klasse `FileOutputStream`

Man beachte ferner, dass bei Dateiverarbeitung Ein- und Ausgabeströme geschlossen werden müssen!

Entsprechend ist das folgende Beispiel jpr02130 aus dem abschließenden Beispiel jpr02129 zum Konsol-IO durch Änderung weniger Zeilen entstanden, die den Dateizugriff durchführen und dabei den entsprechenden Stream bereitstellen. Da beim Instanziieren der Stream-Objekte zu den Ein- und Ausgabedateien Exceptions auftreten können, wurden die nachfolgenden Zeilen des Beispielprogramms etwas umgestaltet. Die in wesentlicher Weise geänderten Zeilen sind im nachfolgenden Programmcode von bsp02130.StreamZugriff markiert:

```
package bsp02130;

import java.io.*;

public class StreamZugriff
{

  public static void main(String[] args)
  {
    ...

    FileInputStream dateiEin = null;
    try
    {
      dateiEin = new FileInputStream("daten/eingabe.txt");
    }
    catch (FileNotFoundException e)
    {
      System.out.println("Datei daten/eingabe.txt konnte nicht gefunden werden!");
    }
    InputStreamReader dateiLeser = null;
    BufferedReader dateiLesePuffer = null;
    if (dateiEin != null)
    {
      dateiLeser = new InputStreamReader(dateiEin);
      dateiLesePuffer = new BufferedReader(dateiLeser);
    }

    FileOutputStream dateiAus = null;
    try
    {
      dateiAus = new FileOutputStream("daten/ausgabe.txt");
    }
    catch (FileNotFoundException e)
    {
      System.out.println("Datei daten/ausgabe.txt konnte nicht gefunden werden!");
    }
    OutputStreamWriter dateiSchreiber = null;
    BufferedWriter dateiSchreibPuffer = null;
    if (dateiAus != null)
    {
      dateiSchreiber = new OutputStreamWriter(dateiAus);
      dateiSchreibPuffer = new BufferedWriter(dateiSchreiber);
    }

// Lesen einer Zeile in einen String
    String zeile = "";
    try
    {
      zeile = dateiLesePuffer.readLine();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Einlesevorgang!");
```

```
      e.printStackTrace();
    }

// Ausgabe eines Strings, eines Zeilenwechsel sowie Flushen des Puffers
    try
    {
      dateiSchreibPuffer.write(zeile);
      dateiSchreibPuffer.newLine();       // Zeilenwechsel
      dateiSchreibPuffer.flush();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Ausgabevorgang!");
      e.printStackTrace();
    }

// Schließen von Reader und Writer sowie beiden Puffern
    try
    {
      dateiLesePuffer.close();
      dateiLeser.close();
      dateiEin.close();
      dateiSchreibPuffer.close();
      dateiSchreiber.close();
      dateiAus.close();
    }
    catch (IOException e)
    {
      System.out.println(
        "Fehler beim Schließen Readers, Writers, Puffers oder Streams!");
      e.printStackTrace();
    }
  }
}
```

Das Beispiel jpr02130 zeigt:

> Abgesehen von der Instanziierung der Objekte zu den Ein- und Ausgabeströmen zu Dateien sowie der damit verbundenen Behandlung von Exceptions gibt es keinen Unterschied bei der Programmierung der Ein- und Ausgabevorgänge für Dateien und der Verarbeitung der IO-Vorgänge zur Konsolein- und ausgabe!

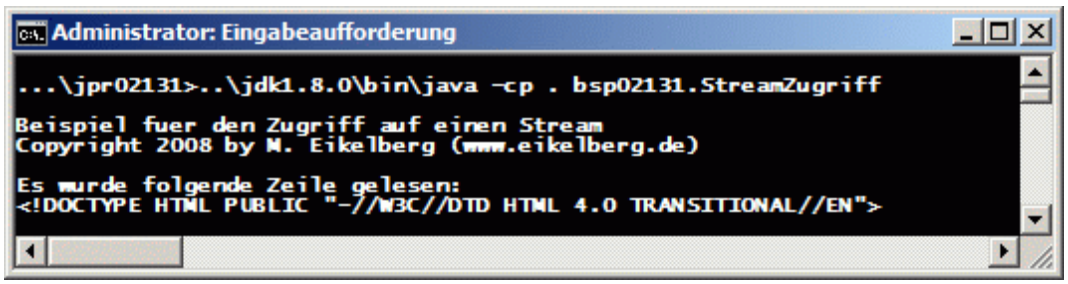
2.10.3 Lesen von einer URL

Auch beim Lesen von einer URL findet die in Kapitel 2.10.1 erklärte Methode für den Zugriff auf einen Ein- oder Ausgabestrom über einen BufferedReader oder BufferedWriter in der beschriebenen Weise Anwendung. Beim Lesen von der URL muss zunächst ein Objekt der Klasse `java.net.URL` zu der URL angelegt werden, von der gelesen werden soll. Anschließend liefert die Instanzmethode `openStream` den InputStream für die in der URL angegebene Datei als Objekt der Klasse `InputStream`. Die weitere Verarbeitung erfolgt dann wie in Kapitel 2.10.1 bzw. 2.10.2, wie das Beispiel jpr02131 zeigt:

```
package bsp02131;

import java.io.*;
import java.net.MalformedURLException;
import java.net.URL;

public class StreamZugriff
{

  public static void main(String[] args)
  {
    ...

    URL url = null;
    InputStream urlEin = null;
    try
    {
      url = new URL("file:webseite//aufg0002.html"); // MalformedURLException
      urlEin = url.openStream();                     // IOException
    }
    catch (MalformedURLException e)
    {
      System.out.println("\nURL fehlerhaft angegeben!");
      e.printStackTrace();
    }
    catch (IOException e)
    {
      System.out.println("\nDatei nicht gefunden!");
      e.printStackTrace();
    }
    InputStreamReader urlLeser = null;
    BufferedReader urlLesePuffer = null;
    if (url != null)
    {
      urlLeser = new InputStreamReader(urlEin);
      urlLesePuffer = new BufferedReader(urlLeser);
    }

// Lesen einer Zeile in einen String
    String zeile = "";
    try
    {
      zeile = urlLesePuffer.readLine();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Einlesevorgang!");
      e.printStackTrace();
    }

// Ausgabe des gelesenen Strings in der Konsole
    System.out.println("Es wurde folgende Zeile gelesen:");
    System.out.println(zeile);
    System.out.println();

// Schließen von Reader und Writer sowie beiden Puffern
    try
    {
      urlLesePuffer.close();
      urlLeser.close();
      urlEin.close();
    }
    catch (IOException e)
```

```
      {
        System.out.println(
          "Fehler beim Schließen Readers oder Puffers!");
        e.printStackTrace();
      }
    }
}
```

Wird die Datei unter der angegebenen URL gefunden, liest das Programm bsp02131.StreamZugriff die erste Zeile dieser Datei:



Man beachte wieder, dass beim Lesen von einer URL der Eingabestrom geschlossen werden muss!

2.10.4 Lesen und Schreiben über einen Socket

Die in Kapitel 2.10.1 beschriebene Technik zum Lesen und Schreiben mit Puffern findet auch bei lesenden und schreibenden (Netzwerk-)Zugriffen über einen TCP-Port Anwendung. In der Sprache der Programmierer heißt ein solcher TCP-Port **Socket**. Beispiel jpr02132 zeigt die Ein- und Ausgabe über einen TCP-Port. Das Programm bsp02132.StreamZugriff

1. öffnet nach dem Start einen TCP-Port für eingehende und ausgehende Verbindungen (ServerSocket).
2. Sobald eine Verbindung eingeht, wird die Verbindung akzeptiert - man erhält einen Socket - und der Eingabestrom zu diesem Socket wird geöffnet.
3. Wie in Kapitel 2.10.1 beschrieben wird der Eingabestrom mit einem BufferedReader verbunden.
4. Neben dem Eingabestrom wird auch der Ausgabestrom zum Socket geöffnet
5. und mit einem BufferedReader verbunden.
6. Es erfolgen Ein- und Ausgabevorgänge über die Puffer, wobei nicht mehr sichtbar ist, dass diese Vorgänge über einen Socket erfolgen.
7. Wie in Kapitel 2.10.1 beschrieben werden die Puffer, Reader, Writer geschlossen.
8. Anders als in Kapitel 2.10.1 aber wie in 2.10.2 und 2.10.3 werden auch die Streams wieder geschlossen.
9. Zusätzlich werden der Socket und der ServerSocket mit einer Methode close() geschlossen.

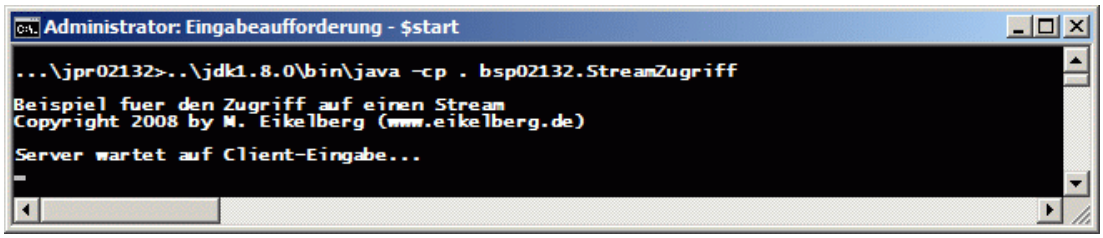
Die für den Ein- und Ausgabevorgang über einen TCP-Port wesentlichen Vorgänge sind im nachfolgenden Listing des Programms bsp02132.StreamZugriff hervorgehoben:

```
package bsp02132;

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

public class StreamZugriff
{

  public static void main(String[] args)
  {
    ...

// Zugriff auf TCP Port
    ServerSocket serverSock = null;
    Socket s = null;
    InputStream socketInStream = null;
    try
    {
// Socket einrichten, der auch Verbindungen annimmt
      serverSock = new ServerSocket(15000);
      System.out.println("Server wartet auf Client-Eingabe...");
// Eine eingehende Netzwerkverbindung akzeptieren
      s = serverSock.accept();
// Referenz auf Eingabestrom zur Netzwerkverbindung holen
      socketInStream = s.getInputStream();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Zugriff auf einen TCP Port!");
    }
    InputStreamReader sockLeser = null;
    BufferedReader sockLesePuffer = null;
    if (socketInStream != null)
    {
      sockLeser = new InputStreamReader(socketInStream);
      sockLesePuffer = new BufferedReader(sockLeser);
    }

    OutputStream socketOutStream = null;
    try
    {
// Referenz auf Ausgabestrom zur Netzwerkverbindung holen
      socketOutStream = s.getOutputStream();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Zugriff auf einen TCP Port!");
    }
    OutputStreamWriter sockSchreiber = null;
```

```
    BufferedWriter sockSchreibPuffer = null;
    if (socketOutStream != null)
    {
      sockSchreiber = new OutputStreamWriter(socketOutStream);
      sockSchreibPuffer = new BufferedWriter(sockSchreiber);
    }

// Begrüßungsmeldung an den Client schicken
    try
    {
      sockSchreibPuffer.newLine();
      sockSchreibPuffer.write("Zugriff auf den Server erfolgreich!");
      sockSchreibPuffer.newLine();
      sockSchreibPuffer.write("Server wartet auf Client-Eingabe...");
      sockSchreibPuffer.newLine();
      sockSchreibPuffer.newLine();
      sockSchreibPuffer.flush();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Ausgabevorgang!");
      e.printStackTrace();
    }

// Lesen einer Zeile in einen String
    String zeile = "";
    try
    {
      zeile = sockLesePuffer.readLine();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Einlesevorgang!");
      e.printStackTrace();
    }

    System.out.println("\nVom TCP Port gelesener Text:\n"+zeile);

// Ausgabe eines Strings, eines Zeilenwechsel sowie Flushen des Puffers
    try
    {
      sockSchreibPuffer.newLine();
      sockSchreibPuffer.write("Beim Server eingangene Eingabe:");
      sockSchreibPuffer.newLine();
      sockSchreibPuffer.write(zeile);
      sockSchreibPuffer.newLine();
      sockSchreibPuffer.flush();
    }
    catch (IOException e)
    {
      System.out.println("Fehler beim Ausgabevorgang!");
      e.printStackTrace();
    }

// Schließen von Reader und Writer sowie beiden Puffern
    try
    {
      sockLesePuffer.close();
      sockLeser.close();
      socketInStream.close();
      sockSchreibPuffer.close();
      sockSchreiber.close();
      socketOutStream.close();
      s.close();
      serverSock.close();
    }
    catch (IOException e)
    {
      System.out.println(
        "Fehler beim Schließen Readers, Writers, Puffers, Streams oder Sockets!");
      e.printStackTrace();
    }
  }
```

}

Beim Start öffnet das Program bsp02132.StreamZugriff den TCP Port 15000 und wartet auf eine eingehende Client-Verbindung:



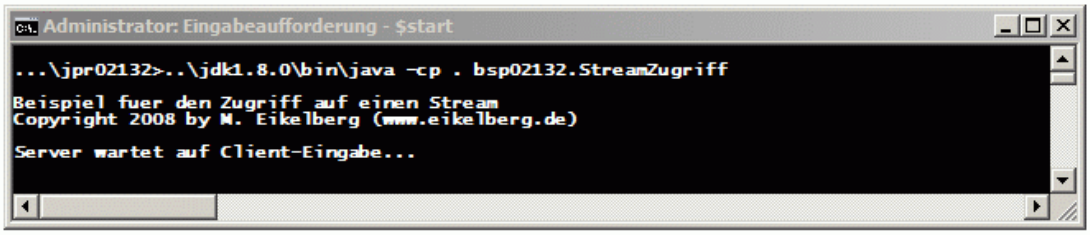
Eine solche Client-Verbindung kann mit Hilfe des Programms telnet hergestellt werden. telnet kann durch Eingabe des Befehls telnet in einer Konsolbox gestartet werden.

Unter Windows 7 muss das Programm telnet gegebenenfalls erst wie folgt freigegeben werden:

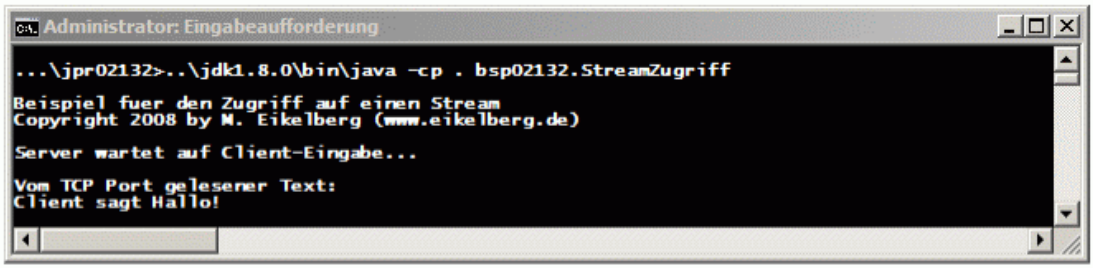
- Öffnen Sie die Anwendung "Programme und Funktionen" der Systemsteuerung.
- Klicken Sie im linken Bereich der Anwendung "Programme und Funktionen" auf den Link "Windows Funktionen aktivieren oder deaktivieren". Es öffnet sich ein Fenster mit der Beschriftung "Windows Funktionen".
- Suchen Sie in dem im Fenster "Windows Funktionen" angezeigten Baum aus Windows Funktionen den Eintrag "Telnet-Client"
- Aktivieren Sie den Telnet-Client, in dem sie einen Haken in der Checkbox vor dem Eintrag zum Telnet-Client setzen. Ist bereits ein Haken gesetzt, so brauchen Sie nichts zu tun - der Telnet-Client ist in diesem Fall bereits aktiviert.

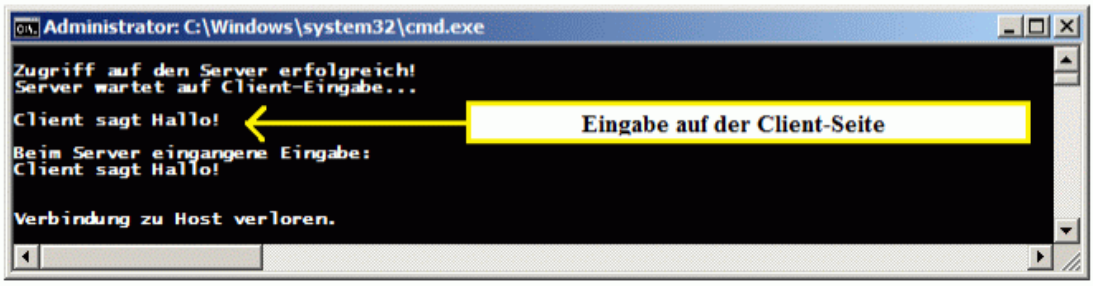
Nach Absetzen des Befehls telnet mit einer geeigneten Hostadresse sowie dem Port, auf dem das Serverprogramm horcht, versucht das Programm telnet einen Verbindungsaufbau:

Gegebenenfalls muss nach Start des Programms Telnet auf Server und Client in der Firewall eine Freigabe der verwendeten Ports erfolgen. Nach Freigabe der Ports ist die Verbindung aufgebaut:



Erfolgt auf dem Client eine Eingabe, so wird diese vom Server gelesen, verarbeitet und an den Client zurückgeschickt. Anschließend beendet der Server die Netzwerkverbindung und der Client meldet, dass er die Serververbindung verloren hat:


Administrator: Eingabeaufforderung
...\jpr02132>..\jdk1.8.0\bin\java -cp . bsp02132.StreamZugriff
Beispiel fuer den Zugriff auf einen Stream
Copyright 2008 by M. Eikelberg (www.eikelberg.de)
Server wartet auf Client-Eingabe...
Vom TCP Port gelesener Text:
Client sagt Hallo!


Administrator: C:\Windows\system32\cmd.exe
Zugriff auf den Server erfolgreich!
Server wartet auf Client-Eingabe...
Client sagt Hallo!
Eingabe auf der Client-Seite
Beim Server eingangene Eingabe:
Client sagt Hallo!
Verbindung zu Host verloren.

2.11 Generics

2.11.1 Generische Klassen

2.11.1.1 Begriffe generische Klasse und Typparameter

Generische Klassen sowie die in dem Zusammenhang mit generischen Klassen verwendete Reflection-Technik stellen ein wesentliches Handwerkszeug fortgeschrittener Java-Programmierung dar. Die Anwendungsmöglichkeiten für generische Klassen und Reflection sind vielfältig, eine Anwendung dieser Techniken kann z. Bsp.

- bei der Implementierung von Datenstrukturen
- bei der Treiberverwaltung für den Datenbankzugriff mit JDBC

erfolgen. Wichtiges Ziel bei der Verwendung generischer Klassen ist die Verbesserung der Typsicherheit und die Vereinfachung bei der Codierung.

Eine **generische Klasse** ist eine Klasse, für die der Typ einer oder mehrer Instanzvariablen nicht festgelegt ist. Vielmehr ist dieser Typ generisch und wird erst bei der Instanziierung eines Objektes festgelegt. Man beachte, dass der Typ einer Klassenvariablen nicht generisch gesetzt werden kann.

P> Ein sehr einfaches Beispiel für eine generische Klasse ist die Klasse bsp02160.GenKlasse:

```
package bsp02160;

public class GenKlasse<T>
{
  private T datenObjekt;

  public GenKlasse(T datenObjekt)
  {
    this.datenObjekt = datenObjekt;
  }

  public T getDatenObjekt()
  {
    return this.datenObjekt;
  }
  public void setDatenObjekt(T datenObjekt)
  {
    this.datenObjekt = datenObjekt;
  }
}
```

In der Klasse bsp02160.GenKlasse ist der Datentyp der (einzigen) Instanzvariablen T nicht festgelegt. Dies wird im Klassennamen dadurch angezeigt, dass der Datentyp nach dem Klassennamen genannt wird.

- Da der Typ von T erst beim Anlegen von Objekten festgelegt wird, nennt man T einen **Typparameter**.
- Der Compiler erstellt aus der obigen Klasse eine einzige Klasse GenClasse.class.
- Die Java Virtuelle Maschine ist zur Laufzeit in der Lage, Objekte anzulegen, für die der Datentyp von T genauer festgelegt wird,
- Compiler und JVM sind in der Lage, Typprüfungen aufgrund der jeweiligen Spezifikation von T durchzuführen.

Mit generischen Klassen kann man also Klassen programmieren, die zur Laufzeit auf viele Datentypen anwendbar sind. Um eine solche flexible Anwendung zu erleichtern, gibt es in der Java API hierzu Klassen und Methoden, die zur Laufzeit die Untersuchung von Objekten in Bezug auf ihren Datentyp sowie ihre Methoden etc. ermöglichen. Diese Technik zur Untersuchung von Objekten nennt man **Reflection** (s.u.).

Die Klasse bsp02160.GenKlasse kann jetzt in einfacher Weise verwendet werden, um Objekte verschiedener Datentypen zu speichern:

```
package bsp02160;

...

public class GenKlasseVerwendung
{
  public static void main(String[] args)
  {
    ...

// Speichern eines String-Objektes in einem Objekt der generischen Klasse:
    String sEin = "Hallo Welt";
```

```
    GenKlasse<String> sGK = new GenKlasse<String>(sEin);   // (1)
    String sRueck = sGK.getDatenObjekt();                     // (2)
    System.out.println(
      "\nString - aus dem Objekt in der generischen Klasse zurueck: "+sRueck);

// Speichern eines Integer-Objektes in einem Objekt der generischen Klasse:
    Integer iEin = new Integer(123);

    GenKlasse<Integer> iGK = new GenKlasse<Integer>(iEin); // (3)
    Integer iRueck = iGK.getDatenObjekt();                    // (4)
    System.out.println(
      "\nZahl - aus dem Integer-Objekt in der generischen Klasse zurueck: "+iRueck.intValue());
  }
}
```

Man beachte:

1. Es wird ein Objekt der generischen Klasse GenKlasse angelegt. Dabei wird als Datentyp für den Typparameter T die Klasse String angegeben. Dies muss sowohl bei der Deklaration als auch bei der Instanziierung - also beim Aufruf des Konstruktors erfolgen.
   - Der Konstruktor wurde in der Klasse GenKlasse ohne Angabe des Typparameters T codiert.
   - Der Aufruf des Konstruktors erfolgt immer mit Angabe der jeweils gewählten Klasse für den Typparameter.
2. Da beim Anlegen des Objektes für den Typparameter die Klasse String festgelegt wurde, liefert jetzt auch die Methode `public T getDatenObjekt()` für das aktuelle Objekt den Datentyp String zurück. Daher erfolgt die Zuweisung der zurückgegebenen Referenz auf das Objekt in eine Variable vom Typ String.
3. Jetzt wird ein Objekt der generischen Klasse angelegt, wobei als Datentyp für den Typparameter T die Klasse Integer angegeben wird - also die Hüllklasse des einfachen Datentyps int.
4. Entsprchend liefert jetzt die Methode `public T getDatenObjekt()` eine Referenz auf ein Integer-Objekt, die in einer Variablen vom Typ Integer gespeichert werden kann.

2.11.1.2 Verwendung generischer Klassen aus der Java API

In der Regel ergibt sich der erste Kontakt eines Programmieres mit generischen Klassen beim Versuch, aufwändigere Datenstrukturen der Java API zu verwenden. Eine dieser Datenstrukturen, auf die man oft schon relativ früh stößt ist die generische Klasse **java.util.Vector<E>**. Für diese Klasse hat man als Buchstaben für den Typparameter den großen Buchstaben E gewählt um anzuzeigen, die Klasse Vector eine Implementierung des Interface **java.util.Collection<E<**. Für die Verwendung der Klasse in Bezug auf ihre Eingeschaft, generische Klasse zu sein, hat dies jedoch keine Bedeutung.

Die Klasse **java.util.Vector<E>** kann dazu verwendet werden, eine unbestimmte Anzahl von Objekten der gleichen Klasse zu speichern. Sie wird also oft an Stellen gewählt, wo eigentlich ein Array eine geeignete Wahl gewesen wäre, wo jedoch keine maximale Anzahl von Objekten festgelegt werden kann. Eine entsprechende Verwendung der Klasse **java.util.Vector<E>** erfolgt in Kapitel 3.10 dieses Kurses. Im einfachsten Fall kann in einem Objekt der Klases **java.util.Vector<E>** eine Liste von Zeichenketten unbestimmter Länge gespeichert werden, wie das folgende Beispiel jpr02161 zeigt:

```
package bsp02161;

import java.util.Vector;
import java.util.Enumeration;
...

public class StringVector
{
  public static void main(String[] args)
  {
    ...

    Vector<String> namensListe = new Vector<String>();

// Einfügen in die Liste
    namensListe.add("Heinz");
    namensListe.add("Lieschen");
    namensListe.add("Fritz");
    namensListe.add("Hilda");

// Ausgabe der gesamten Liste kann über das generische
// Interface java.util.Enueration erfolgen:
// 1. Feststellen der Anzahl der Elemente
   Enumeration<String> namensEnum = namensListe.elements();
   int zaehler = 0;
   while (namensEnum.hasMoreElements())
   {
     namensEnum.nextElement();
```

```
      zaehler++;
    }
// 2. Umkopieren in einen String-Array
// Enumeration erneut erstellen, da durchgearbeitet:
    namensEnum = namensListe.elements();
    String[] namensArray = new String[zaehler];
    zaehler = 0;
    while (namensEnum.hasMoreElements())
    {
      namensArray[zaehler] = namensEnum.nextElement();
      zaehler++;
    }

// Ausgabe der Liste:
    System.out.println();
    for (int i=0; i<namensArray.length;i++)
      System.out.println("Element "+i+" hat Inhalt: "+namensArray[i]);

  }
}
```

In obigem Quellcode sind folgende Punkte bedeutsam:

- Bei der Deklaration sowie bei der Instanziierung - also dem Aufruf des Konstruktors - muss der Datentyp für den Typparameter für die generische Klasse java.util.Vector<E> angegeben werden.
- Ebenso muss bei der Deklaration der Variablen des generischen Interface java.util.Enumeration<E> der Datentyp für den Typparameter angegeben werden.
- Bei den Methodenaufrufen muss weder für die Klasse java.util.Vector<E> noch für das Interface java.util.Enumeration<E> der Datentyp für den Typparameter angegeben werden. Dieser Datentyp ist dem Compiler ja bereits aus der Deklaration bekannt. Entsprechend reicht diese Angabe auch für eine Typprüfung für den Compiler aus.

2.11.1.3 Vereinfachung der Programmierung durch Codierung generischer Klassen

Durch Codierung generischer Klassen kann der Programmieraufwand wesentlich gesenkt und auch der Aufwand zur Programmierung erheblich verringert werden. Ein Beispiel hierfür ist die nachfolgende Implementierung einer sogenannten "Warteschlange". Eine Warteschlange ist eine standard Datenstruktur der Informatik zur Speicherung einer unbestimmten Anzahl von Elementen. Dabei gilt das Prinzip **first in first out**, d.h. das zuerst in die Warteschlange eingefügte Element verlässt die Warteschlage auch als erstes Element. Im Allgemeinen ist als das Element, das die Warteschlange verlässt immer das Element, das sich am längsten in der Warteschlange befindet.

Die Implementierung der Warteschlange erfolgt unter Verwendung eines Arrays in der Klasse bsp02162.Queue. Für diese Klasse ist der Datentyp der in ihren Objekten zu speichernden Daten nicht festgelegt, sondern mit Hilfe eines Typparameters variabel gehalten. Dies ermöglicht, die Klasse bsp02162.Queue für Warteschlangen aus beliebigen Objekten zu verwenden:

```
package bsp02162;

public class Queue<T>
{
  private T[] schlange = null;
  private int anfang;   // Index erstes belegtes Feld des Arrays
  private int ende;     // Index leeres Feld am Ende der Warteschlange

// Konstruktor
  @SuppressWarnings("unchecked")
  public Queue(int laenge)
  {
// Arrays vom generischen Typ T sind nicht möglich! Daher muss ein Object-Array
// gewählt werden, der in einem unchecked cast in einen Array vom Typ T umgewandelt
// werden muss. Die entsprechenden Compilerwarnungen können mit der Annotation
// qSuppressWarnings(...) unterdrückt werden:
    this.schlange = (T[]) new Object[laenge];

// Warteschlange als leer kennzeichnen - anfang gleich ende setzen
    this.anfang = 0;
    this.ende = 0;
  }

// Einfügen am  Ende der Warteschlange
  public boolean enqueue(T t)
  {
    boolean ok = false;
    if (
// Das Ende liegt direkt vor dem Anfang der Warteschlange
```

```
     this.anfang-this.ende == 1 |
// Anfang liegt am Anfang des Arrays und Ende am Ende des Arrays
     this.anfang-this.ende == -this.schlange.length+1
    )
    {
// Fehler, denn die Warteschlange ist voll
      System.out.println("\n! Einfuegen fehlgeschlagen - Warteschlange ist voll!");
    }
    else
// Einfügen des Objektes und Verschieben des Endes der Warteschlange
    {
// Anfügen am Ende der Schlange
      this.schlange[this.ende] = t;
      ok = true;
      System.out.println("\n+ Eingefuegt wurde Element mit Index: "+this.ende);
// Verschieben des Ende der Schlange
      if (this.ende == this.schlange.length-1)
// Wenn man am Ende des Arrays angekommen ist muss man wieder nach vorne gehen
        this.ende = 0;
      else
        this.ende = this.ende + 1;
    }
    return ok;
  }

// Entfernen am Anfang der Warteschlange
  public T dequeue()
  {
    T t = null;
    if (this.anfang == this.ende)
    {
// Fehler, denn die Schlange ist leer
      System.out.println("\n! Entfernen fehlgeschlagen - Warteschlange ist leer!");
    }
    else
    {
// auf das erste Element der Warteschlange zugreifen
      t = this.schlange[this.anfang];
      System.out.println("\n- Entfernt wurde Element mit Index: "+this.anfang);
// Anfang der Schlange verschieben
      if (this.anfang == this.schlange.length-1)
// Wenn man am Ende des Arrays angekommen ist, wieder an den Anfang des Arrays gehen
        this.anfang = 0;
      else
// ansonsten den Anfang im Array nach hinten verschieben
        this.anfang = this.anfang + 1;
    }
    return t;
  }
}
```

Der Quellcode zeigt eine sehr wichtige Einschränkung bei der Verwendung von Typparametern:

Es ist nicht möglich, einen Array vom einem Typparameter zu instanziieren.

Beispiel jpr02162 zeigt zugleich die - mit Schönheitsfehlern verbundene - Umgehung dieser Einschränkung:

- Die Deklaration des Arrays kann unter Verwendung des Typparameters T erfolgen:

```
private T[] schlange = null;
```

- Die Instanziierung des Arrays muss unter Verwendung der Klasse Object sowie eines Casts erfolgen. Dies ergibt Compilerwarnungen wegen einer unchecked conversion, die mit der Annotation @SuppressWarnings("unchecked") unterdrückt werden:

```
@SuppressWarnings("unchecked")
public Queue(int laenge)
{
  ...
  this.schlange = (T[]) new Object[laenge];
  ...
  this.anfang = 0;
  this.ende = 0;
}
```

- Die Methoden zum Einfügen (enqueue) und Entfernen (dequeue) aus der Warteschlange können mit Hilfe des Typparameters codiert

werden:

```
public boolean enqueue(T t)
{
  ...
  this.schlange[this.ende] = t;
  ...
}
```

Ferner:

```
public T dequeue()
{
  T t = null;
  ...
  t = this.schlange[this.anfang];
  ...
  return t;
}
```

Die Verwendung der Klasse bsp02162.Queue für eine Warteschlange aus String-Objekten zeigt die Klasse bsp02162.WarteschlangeTest:

```
package bsp02162;

...

public class WarteschlangeTest {

  public static void main(String[] args)
  {
    ...

//  Warteschlange für vier Elemente erzeugen, mit String als Typparameter
    Queue<String> q = new Queue<String>(5); // Warteschlange mit max. 5-1 Elementen

//  drei Elemente in die Warteschlange einfügen
    for (int i=0;i<3;i++)
    {
      String s = new String("E_0"+i);
      boolean ok = q.enqueue(s);
      if (ok)
        System.out.println("\nEingefuegt wurde das Element: '"+s+"'");
      else
        System.out.println("\nEinfuegen des Elements : '"+s+"' fehlgeschlagen!");
    }

//  ein Element aus der Warteschlange entfernen
    String s = q.dequeue();
    System.out.println("\nEntfernt wurde das Element: '"+s+"'");
  }
}
```

Das Programm bsp02162.WarteschlangeTest enthält in Bezug auf das Thema generische Klassen drei wichtige Schritte:

- Die Warteschlange wird instanziiert, wobei die Klasse String für den Typparameter angegeben wird.
- Die Methode enqueue(...) nimmt als Parameter ein String-Objekt an.
- Die Methode dequeue() ergibt als Rückgabewert ein String-Objekt.

Der nachfolgende Bildschirmausdruck zeigt das Verhalten des Programms bsp02162.WarteschlangeTest:

```
Administrator: Eingabeaufforderung
...\jpr02162>..\jdk1.8.0\bin\java -cp .;de_eikelberg_iotools.jar bsp02162.WarteschlangeTest
--------------------------------------------------------------------------------
    Beispielprogramm zur Warteschlange, implementiert mit Typparameter
    Copyright 2011 by by N. Eikelberg (www.eikelberg.de)
--------------------------------------------------------------------------------

+ Eingefuegt wurde Element mit Index: 0

Eingefuegt wurde das Element: 'E_00'

+ Eingefuegt wurde Element mit Index: 1

Eingefuegt wurde das Element: 'E_01'

+ Eingefuegt wurde Element mit Index: 2

Eingefuegt wurde das Element: 'E_02'

- Entfernt wurde Element mit Index: 0

Entfernt wurde das Element: 'E_00'
```

2.11.2 Kovarianz und Kontravarianz

2.11.2.1 Typparameter Fragezeichen als Wildcard bei der Instanziierung von Objekten generischer Klasse

Zusätzlich zu der in Kapitel 2.11.1 erklärten Möglichkeit, bei der Deklaration und Instanzziierung von Objekten generischer Klassen den Typ für den Typparameter festzulegen gibt es auch die Möglichkeit, diesen Typ *nicht* festlegen.

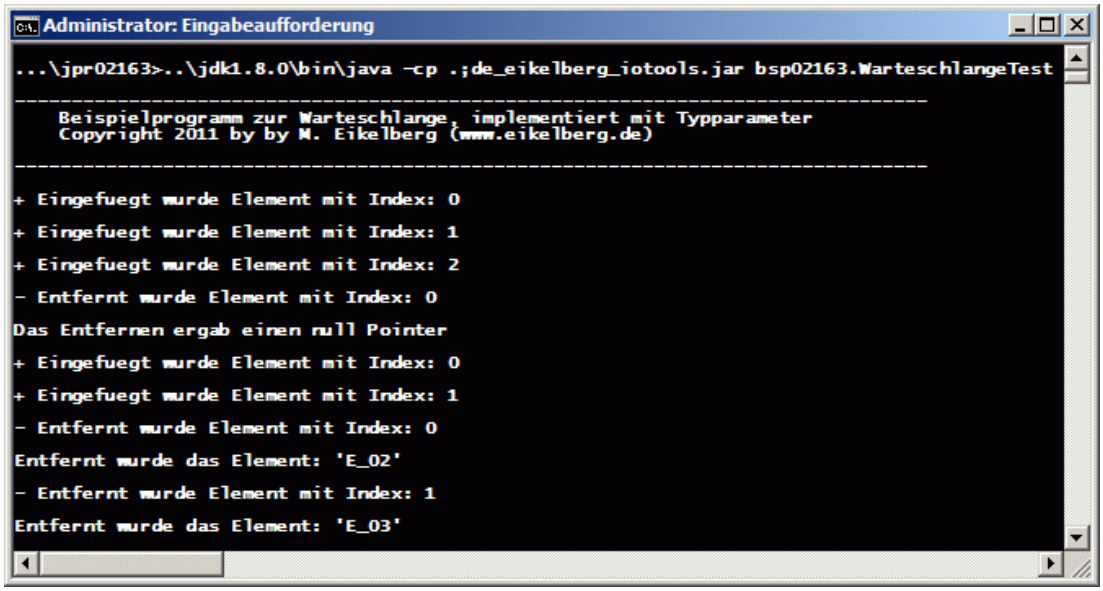
- Hierzu codiert man bei Deklaration und Instanziierung statt des zu wählenden Datentyps - als sogenanntes Wildcard-Zeichen - ein Fragezeichen.
- In diesem Fall wird bei der Instanziierung als Datentyp für den Typparameter die Klasse Object gewählt, sofern der Programmierer in der generischen Klasse keine zusätzlichen Vorkehrungen für diesen Fall getroffen hat (vgl. 2.11.2.2).

Als Beispiel für ein solches Vorgehen wird im Beispiel jpr02163. In diesem Projekt wird die generischen Klasse bsp02162.Queue aus Kapitel 2.11.1 unverändert weitergenutzt. In einem Beispielprogramm bsp02163.WarteschlangeTest wird jetzt jedoch untersucht, welche Operationen bei einer Instanziierung der generischen Klasse bsp02162.Queue mit dem Fragezeichen als Wildcard möglich sind:

```
package bsp02163;

import bsp02162.Queue;
...

public class WarteschlangeTest {

  @SuppressWarnings("rawtypes")
  public static void main(String[] args)
  {
    ...

//  Warteschlange für vier Elemente erzeugen, ohne Angabe eines Typparameters
//  Man beachte, dass beim Konstruktor kein Parameter steht! Dies ergibt eine
//  Compilerwarnung, die mit der Annotation @SuppressWarnings("rawtypes")
//  unterdrückt wird
    Queue<?> q1 = new Queue(5); // Warteschlange mit max. 5-1 Elementen

/*
 *  Einfügen in die Warteschlange q1 ist nicht möglich:
 *
 *   Object o1 = new String("E_01");
 *  boolean ok = q1.enqueue(o1);       // Compilerfehler!
 */

// Ein Einfügen von null Pointern ist aber möglich:
    q1.enqueue(null);
    q1.enqueue(null);
    q1.enqueue(null);

//  ein Element aus der Warteschlange entfernen
    Object o1 = q1.dequeue();
    if (o1 == null)
      System.out.println("\nDas Entfernen ergab einen null Pointer");
    else
      System.out.println("\nEntfernt wurde das Element: '"+o1.toString()+"'");

//  Weitere Warteschlange erzeugen, für die das Objekt selber den Datentyp für
//  den Typparameter kennt:
    Queue<String> q2 = new Queue(5);
    Queue<?>      q3 = q2;           // nur Zuweisung der Referenz

    String s2 = new String("E_02");
    String s3 = new String("E_03");

    q2.enqueue(s2);
/*
 *  Einfügen in die Warteschlange q1 ist nicht möglich:
 *
 *  q3.enqueue(s3);                  // Compilerwarnung!
 */
    q2.enqueue(s3);

    String s4 = q2.dequeue();
    System.out.println("\nEntfernt wurde das Element: '"+s4+"'");
```

```
    Object o5 = q3.dequeue();          // Dequeue als Objekt!
    System.out.println("\nEntfernt wurde das Element: '"+o5.toString()+"'");
  }
}
```

Dieses Programm führt zu folgender Bildschirmausgabe:



Der Quellcode von bsp02163.WarteschlangeTest enthält mehrere bemerkenswerte Anweisungen:

- Unter Angabe des Fragezeichens als Typparameter wird eine erste Warteschlange q1 erfolgreich instanziiert. Diese ist jedoch ein sogenannter "roher Typ" (rawtype), weshalb der Compiler eine Warnung ausgibt. Diese Warnung wird im obigen Beispielprogramm mit der Annotation `@SuppressWarnings("rawtypes")` unterdrückt. Entsprechend der Tatsache, dass die Warteschlange als roher Typ angelegt wird, wird sie mit Hilfe eines Konstruktors instanziiert, für den kein Typparameter angegeben wird:

  ```
  @SuppressWarnings("rawtypes")
  Queue<?> q1 = new Queue(5);
  ```

- Für diese Warteschlange ist ein Einfügen von Objekten nicht möglich, da der Typparameter nicht angegeben wurde:

  ```
  /*
   *  boolean ok = q1.enqueue(o1);        // Compilerwarnung!
   '/
  ```

- Das Einfügen von null Pointern ist aber möglich und wird von der Klasse bsp02162.Queue korrekt verarbeitet, wie der obige Bildschirmausdruck zeigt:

  ```
  q1.enqueue(null);
  ```

- Das Entfernen von Objekten aus der Warteschlange ist ebenfalls möglich, beim Entfernen erhält man wegen der Angabe des Fragezeichens als Typparameter jetzt allerdings den Datentyp Object:

  ```
  Object o1 = q1.dequeue();
  ```

Wird die Warteschlange zunächst unter Angabe eines Typparameters instanziiert und ihre Referenz erst anschließend mit einer Zuweisung auf eine Variable kopiert, für die kein Typparameter angegeben wurde, so verhält sich das Objekt anders, wie die folgenden Schritte des Programms zeigen:

- Instanziieren der Warteschlange mit Typparameter String und Umkopieren der Referenz auf die Warteschlange in eine Variable ohne Typparameter:

  ```
  Queue<String> q2 = new Queue(5);
  Queue<?>      q3 = q2;            // nur Zuweisung der Referenz
  ```

- Ein Einfügen von Objekten in die Warteschlange ist nur über die Variable q2 möglich:

  ```
  q2.enqueue(s2);
  /*
   *  Einfügen in die Warteschlange q1 ist nicht möglich:
   *
   *  q3.enqueue(s3);                  // Compilerwarnung!
   */
  q2.enqueue(s3);
  ```

- Das Entfernen eines Objektes aus der Warteschlange über die Variable q2 ergibt ein Objekt vom Typ String:

  ```
  String s4 = q2.dequeue();
  ```

- Das Entfernen eines Objektes aus der Warteschlange über die Variable q3 ergibt ein Objekt vom Typ Object, da kein Typparameter angegeben wurde:

```
Object o5 = q3.dequeue();          // Dequeue als Objekt!
```

Das Beispiel zeigt somit, dass bei der Verwendung des Fragezeichens als Wildcard für den Typparameter

- ein Aufruf von Methoden einer generischen Klasse mit Typparametern als Parametern nur mit Angabe des null Pointers als Parameter möglich ist sowie
- Rückgabewerte von Methoden einer generischen Klasse mit Typparameter als Rückgabewert stets vom allgemeinsten Datentyp Object sind.

2.11.2.2 Kovarianz und Kontravarianz bei der Instanziierung von Objekten generischer Klasse

Einschränkend zu dem Vorgehen in 2.11.2.1 ist es möglich, bei der Instanziierung

1. Klassen zuzulassen, die von einer bestimmten Klasse abgeleitet sind bzw.
2. Klassen zuzulassen, von der eine bestimmte Klasse abgeleitet ist.

Bezogen auf die Vererbungshierarchie bedeutet dies folgendes:

1. Es werden alle Klassen zugelassen, die gleich der angegebenen Klasse oder in der Vererbungshierarchie unterhalb der angegebenen Klasse angegeben sind.
2. Es werden alle Klassen zugelassen, die gleich der angegebenen Klasse oder in der Vererbungshierarchie oberhalb der angegebenen Klasse angegeben sind.

Man bezeichnet dieses Zulassen als

1. Kontravarianz - Schlüsselwort `super`
2. Kovarianz - Schlüsselwort `extends`

Die Angabe eines eindeutigen Typparameters wie in Kapitel 2.11.1 bezeichnet man in diesem Fall als **Invarianz**.

Die Verwendung des Schlüsselwortes super - also die Codierung von Kontravarianz - ist eher selten. Beispiel jpr02164 zeigt das entsprechende Vorgehen und verwendet erneut die unveränderte Klasse bsp02162.Queue:

```
package bsp02164;

import bsp02162.Queue;
...

public class WarteschlangeTest {

  public static void main(String[] args)
  {
    ...

//  Warteschlange für vier Elemente erzeugen. Beim Konstruktor wird als oberster
//  zulässiger Datentyp die Klasse Object angegeben
    Queue<? super Integer> q = new Queue<Object>(5);

//  Einfügen in die Warteschlange q1:
     Integer i1 = new Integer(1);
     boolean ok = q.enqueue(i1);
     if (ok)
       System.out.println("\nEinfuegen eines Integer-Objektes '"+i1.intValue()+"'
erfolgreich.");

     Object o1 = new Integer(2);
//   ok = q.enqueue(o1);         //Compilerfehler: Einfügen ist nur für Objekte vom Typ Integer
möglich
     System.out.println("\nEinfuegen des Objektes '"+((Integer)o1).intValue()+"' nicht
moeglich.");

//  Entfernen aus der Warteschlange
     Integer i2 = (Integer)q.dequeue();  // Entfernen erfordert anschließenden Cast nach
Integer!
     System.out.println("\nEntfernen des Objektes '"+i2.intValue()+"' erfolgreich.");

     Integer i3 = new Integer(3);
     ok = q.enqueue(i3);
     if (ok)
       System.out.println("\nErneutes Einfuegen eines Integer-Objektes '"+i3.intValue()+"'
erfolgreich.");
```

```
      Object n2 = q.dequeue();            // Entfernen ergibt nur Datentyp Object
      Integer i4 = (Integer)n2;
      System.out.println("\nEntfernen des Objektes '"+i4.intValue()+"' erfolgreich.");
   }
}
```

Die Quintessenz aus Beispiel jpr02164 ist, ist dass bei Einschränkung des Datentyps für den Typparameter mit dem Schlüsselwort super bei der Instanziierung eines Objekts - also bei **Kontravarianz** -

- ein Aufruf von Methoden mit Typparametern als Parametern die Angabe von aktuellen Parametern von dem beim Einschränken angegebenen Datentyp (hier Integer) erfordert sowie
- Rückgabewerte von Methoden mit Typparameter als Rückgabewert stets vom allgemeinsten Datentyp Object sind.

Die Situation bei Codierung von Kovarianz zeigt Beispiel jpr02165. Es besteht wie Beispiel jpr02164 wieder aus der unveränderten Klasse bsp02162.Queue sowie einer individuell codierten Klasse bsp02165.WarteschlangeTest:

```
package bsp02165;

import bsp02162.Queue;
...

public class WarteschlangeTest {

  @SuppressWarnings("unused") // Warnung wg. ungenutzter Variablen unterdrücken
  public static void main(String[] args)
  {
    ,,,

//  Warteschlange für vier Elemente erzeugen.
    Queue<Integer> q1 = new Queue<Integer>(5);
    Queue<? extends Number> q2 = q1;

//  Einfügen in die Warteschlange ist nur über q1 und nicht übe q2 möglich
    boolean ok = false;
    Integer i1 = new Integer(1);
    ok = q1.enqueue(i1);
    if (ok)
      System.out.println("\nEinfuegen eines Integer-Objektes '"+i1.intValue()+"'
erfolgreich.");
//  ok = q2.enqueue(i1);                    // Compilerfehler!
    Number n1 = new Integer(2);
//  ok = q1.enqueue(n1);                    // Compilerfehler - n1 hat falschen Datentyp
//  ok = q2.enqueue(n1);                    // Compilerfehler!
    Object o1 = new Integer(3);
//  ok = q1.enqueue(o1);                    // Compilerfehler - o1 hat falschen Datentyp
//  ok = q2.enqueue(o1);                    // Compilerfehler!
    ok = q1.enqueue(null);
    if (ok)
      System.out.println("\nEinfuegen des null Pointers erfolreich.");
    ok = q2.enqueue(null);                  // Einfügen des null Pointers über q2 ist möglich!
    if (ok)
      System.out.println("\nEinfuegen des null Pointers erfolreich.");

//  Entfernen aus der Warteschlange über q1
     Integer i2 = q1.dequeue();
     System.out.println("\nEntfernen des Objektes '"+i2.intValue()+"' erfolgreich.");

//  Entfernen der beiden null Pointer und erneutes Einfügen von i1 den nachfolgenden Test
     Object o2 = q2.dequeue();
     if (o2 == null)
       System.out.println("\nEnfernen des null Pointers erfolgreich.");
     o2 = q2.dequeue();
     if (o2 == null)
       System.out.println("\nEnfernen des null Pointers erfolgreich.");
     ok = q1.enqueue(i1);
     if (ok)
       System.out.println("\nErneutes Einfuegen eines Integer-Objektes '"+i1.intValue()+"'
erfolgreich.");

//  Entfernen aus der Warteschlange über q2
     Number n2 = q2.dequeue();
     Integer i3 = (Integer)n2;
     System.out.println("\nEntfernen des Objektes '"+i3.intValue()+"' erfolgreich.");
```

```
  }
}
```

Die Quintessenz aus Beispiel jpr02165 ist, ist dass bei Einschränkung des Datentyps für den Typparameter mit dem Schlüsselwort extends bei der Instanziierung eines Objekts - also bei **Kovarianz** -

- ein Aufruf von Methoden mit Typparametern als Parametern nur mit Angabe des null Pointers als Parameter möglich ist sowie
- Rückgabewerte von Methoden mit Typparameter als Rückgabewert stets von dem bei der Instanziierung des Objektes angegebenen Datentyps sind.

2.11.2.3 Kovarianz und Kontravarianz bei der Codierung der generischen Klasse

Kovarianz und Kontravarianz ist auch bei der Codierung von generischen Klassen möglich. Wir beschränken uns hier auf die Betrachtung der Kovarianz, da sie weitaus häufiger angewendet wird. Die Wirkung der Codierung des Schlüsselwortes extends - also die Wirkung von Kovarianz - bei der Codierung einer generischen Klasse ist leicht zu erklären. Mit der Klausel `extends` werden eine oder gegebenenfalls mehrere Klassen bzw. Interfaces angegeben, von der die aktuell als Parameter anzugebenden Klassen abgeleitet sein müssen bzw. die implemtiert werden müssen. So erzwingt die Codierung von

```
public class Queue<T extends Number & Comparable>
```

dass der als aktueller Typparameter bei der Instanziierung von Objekten der generischen Klasse angegebene Datentyp sowohl von der abstrakten Klasse Number abgeleitet sein muss als auch das Interface Comparable implementieren muss. Im Beispielprogramm jpr02166 wird lediglich wie folgt gefordert, dass der als aktueller Typparameter angegebene Datentyp von der abstrakten Klasse Number abgeleitet sein muss:

```
package bsp02166;

public class Queue<T extends Number>
{
  ...
  public Queue(int laenge)
  {
    ...
  }

  public boolean enqueue(T t)
  {
    ...
  }

  public T dequeue()
  {
    ...
  }
}
```

Für die Verwendung der Klasse bsp02166.Queue bedeutet dies, dass *lediglich die abstrakte Klasse Number* sowie alle von ihr abgeleiteten Klassen als aktuelle Typparameter zulässig sind. Dies zeigt das bsp02166.WarteschlangeTest:

```
package bsp02166;

...

public class WarteschlangeTest {

  public static void main(String[] args)
  {
    ....

//  Warteschlange für vier Elemente erzeugen mit der Klasse Integer als Parameter
    Queue<Integer> q1 = new Queue<Integer>(5);
    q1.enqueue(new Integer(1));

//  Warteschlange für vier Elemente erzeugen mit der abstrakten Klasse Number als Parameter
    Queue<Number> q2 = new Queue<Number>(5);
    q2.enqueue(new Integer(2));                  // Integer ist von Number abgeleitet

//  Warteschlange für vier Elemente erzeugen mit der Klasse String als Parameter
//  Queue<String> q3 = new Queue<String>(5);     // Compilerfehler
  }
}
```

2.11.2.4 Beispiel zur Verwendung von Kovarianz bei der Codierung von generischen Klassen

Beispiel jpr02167 zeigt die Klasse bsp02167.ZahlParser<T extends Number>, die als aktuellen Typparameter jede von der Klasse Number abgeleite Klasse annimmt. Dies sind Klassen wie z. Bsp Integer, Long, Float und Double. Die Methode pruefeTyp() der Klasse bsp02167.ZahlParser<T extends Number>

- untersucht, von welchem Datentyp der bei der Instanziierung der generischen Klasse angegebene Datentyp real ist,
- wandelt den Inhalt des jeweiligen Objektes passend in seine String-Darstellung um und
- gibt den bei der Prüfung festgestellten Datentyp in der Konsole aus.

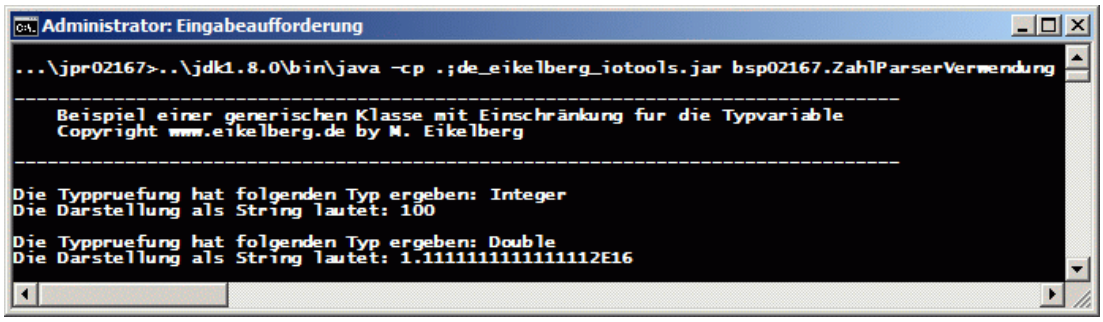
```
package bsp02167;

public class ZahlParser<T extends Number>
{
  private T zahlTyp = null;

  public ZahlParser() { }
  public ZahlParser(T zahlTyp)
  {
    this.zahlTyp = zahlTyp;
  }

  public T getZahlTyp()
  {
    return this.zahlTyp;
  }
  public void setZahlTyp(T zahlTyp)
  {
    this.zahlTyp = zahlTyp;
  }

  public void pruefeTyp()
  {
    String stringDarst = "\n*** Fehler - String-Darstellung konnte nicht ermittelt werden!
***";
    String typ = "\n*** Fehler - Typ konnte nicht ermittelt werden! ***";
    if (this.zahlTyp instanceof Integer)
    {
// Integer erbt von der Klasse Number, die als abstrakte Klasse Methoden intValue() etc.
fordert
      int x = this.zahlTyp.intValue();
      stringDarst = String.valueOf(x);
      typ = "Integer";
    }
    else if (this.zahlTyp instanceof Long)
    {
      long x = this.zahlTyp.longValue();
      stringDarst = String.valueOf(x);
      typ = "Long";
    }
    else if (this.zahlTyp instanceof Float)
    {
      float x = this.zahlTyp.floatValue();
      stringDarst = String.valueOf(x);
      typ = "Float";
    }
    else if (this.zahlTyp instanceof Double)
    {
      double x = this.zahlTyp.doubleValue();
      stringDarst = String.valueOf(x);
      typ = "Double";
    }
    System.out.println("\nDie Typpruefung hat folgenden Typ ergeben: "+typ);
    System.out.println("Die Darstellung als String lautet: "+stringDarst);
  }
}
```

Im Beispielprogramm bsp02167.ZahlParserVerwendung werden dann Objekte verschiedener von der abstrakten Klasse Number abgeleiteter Klassen als Parameter der Methode setZahlTyp(T zahlTyp) der generischen Klasse bsp02167.ZahlParser<T extends Number> eingesetzt:

```
package bsp02167;

...

public class ZahlParserVerwendung
{
```

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

// Objekt der generischen Klasse anlegen und dabei mit der abstrakten Klasse Number
// (Superklasse von Integer etc.) parametriesieren
    ZahlParser zp = new ZahlParser();

// Objekt einer von Number abgeleiteten Hüllklasse in das Objekt für den generischen Typ von zp
setzen
    Integer iC = new Integer(100);
    zp.setZahlTyp(iC);
// Methode der generischen Klasse ausführen
    zp.pruefeTyp();

// jetzt das gleiche mit einem Double-Objekt
    Double dC = new Double(1111111111111111.1d);
    zp.setZahlTyp(dC);
    zp.pruefeTyp();
  }
}
```

Der nachfolgende Bildschirmausdruck zeigt, dass der Datentyp der jeweiligen aktuellen Typparameter korrekt erkannt wird:



2.11.2.5 Beispiel zur Codierung mehrerer Typparameter

Es bestehen keine Probleme, gegebenenfalls mehrere Typparameter zu codieren. Hierbei verwendet man üblicherweise die Buchstaben in der Nachbarschaft des Buchstaben T, also z. Bsp. die Buchstaben U oder S. Beispiel jpr02168 zeigt die Codierung zweier Typparameter in der Klasse:

```
package bsp02168;

public class SpeicherMitZaehler<T, U extends Number>
{
  private U zaehler;
  private T datenObjekt;
  public SpeicherMitZaehler(U zaehler)
  {
    if (zaehler instanceof Number)
      this.zaehler = zaehler;
    else
      System.out.println("\nFehler bei der Instanziierung der Klase GenZaehler - keine von
Number abgeleitete Klasse angegeben.");
  }

  public U getZaehler()
  {
    return this.zaehler;
  }

  public T getDatenObjekt()
  {
    return this.datenObjekt;
  }

  @SuppressWarnings("unchecked")  // Annotation, die Warnungen wg. ungeprüftem
                                  // Cast (hier in den Datentyp U) unterdrückt
  public void setDatenObjekt(T datenObjekt)
  {
    if (zaehler instanceof Byte)
```

```
    {
      byte z = zaehler.byteValue();  // aus Hüllklasse entpacken
      zaehler = (U)new Byte(++z);
    }
    else if (zaehler instanceof Short)
    {
      short z = zaehler.shortValue();
      zaehler = (U)new Short(++z);
    }
    else if (zaehler instanceof Integer)
    {
      int z = zaehler.intValue();
      zaehler = (U)new Integer(++z);
    }
    else if (zaehler instanceof Long)
    {
      long z = zaehler.longValue();
      zaehler = (U)new Long(++z);
    }
    else if (zaehler instanceof Float)
    {
      float z = zaehler.floatValue();
      zaehler = (U)new Float(++z);
    }
    else if (zaehler instanceof Double)
    {
      double z = zaehler.doubleValue();
      zaehler = (U)new Double(++z);
    }
    else
      System.out.println("\nFehler bei der Feststellung des Zaehlers - keine Huellklasse eines
einfachen Datentyps angegeben.");

    this.datenObjekt = datenObjekt;
  }
}
```

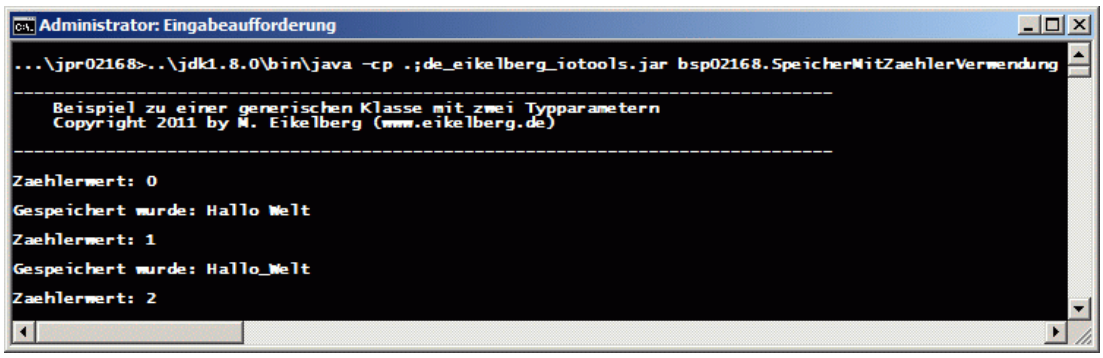
Der Konstruktor der Klasse SpeicherMitZaehler<T, U extends Number> übernimmt lediglich ein Objekt vom Typparameter U, untersucht dieses auf den Datentyp und speichert es bei geeignetem Datentyp in das Objekt der generischen Klasse ein. Dieses als Parameter des Konstruktors übergebene Objekt dient als Zähler. Die im Objekt zu speichernden Daten werden mit einer set-Methode setDatenObjekt(T datenObjekt) in das Objekt eingespeichert. Dabei wird das als Zähler verwendete Objekt auf seinen realen Datentyp hin analysiert und die in diesem Objekt enthaltene Zahl passend zum Objekttyp um eins erhöht.

Im Testprogramm bsp02168.SpeicherMitZaehlerVerwendung wird dann ein Objekt der Klasse Integer als Zähler verwendet. Eingespeichert werden Objekte der Klasse String:

```
package bsp02168;

...

public class SpeicherMitZaehlerVerwendung
{
  public static void main(String[] args)
  {
    ...

// Objekt anlegen und dabei Zähler-Objekt übergeben
    Integer zaehler = new Integer(0);
    SpeicherMitZaehler<String,Integer> smz = new SpeicherMitZaehler<String,Integer>(zaehler);

// Zählerwert feststellen
    Integer z1 = smz.getZaehler();
    System.out.println("\nZaehlerwert: "+z1.intValue());

// Objekt speichern
    String s1 = "Hallo Welt";
    smz.setDatenObjekt(s1);
    System.out.println("\nGespeichert wurde: "+smz.getDatenObjekt());

 // Zählerwert feststellen
    z1 = smz.getZaehler();
    System.out.println("\nZaehlerwert: "+z1.intValue());

 // Objekt speichern
```

```
    String s2 = "Hallo_Welt";
    smz.setDatenObjekt(s2);
    System.out.println("\nGespeichert wurde: "+smz.getDatenObjekt());

 // Zählerwert feststellen
    z1 = smz.getZaehler();
    System.out.println("\nZaehlerwert: "+z1.intValue());
  }
}
```

Der nachfolgende Bildschirmausdruck zeigt, dass diese Verarbeitung erfolgreich durchgeführt wird:



2.11.3 Reflection

Reflection ist eine Technik, die es Java-Programmen erlaubt, sich selber bzw. die von ihm verwendeten Objekte in Bezug auf die Eigenschaften - also Variablen und Methoden - zu untersuchen. Eine sehr wichtige Anwendung von Reflection ist die mit Hilfe der sogenannten JavaBeans mögliche visuelle Entwicklung grafischer Benutzeroberflächen (vgl. Kap. 13). Die hierbei verwendeten visuellen Editoren für die GUI laden die möglichen Komponenten der GUI dynamisch und untersuchen diese Komponenten mit Hilfe von Reflection auf ihre Eigenschaften. Reflection als Technik zur Untersuchung des laufenden Programms ist in vielen Programmiersprachen nicht verfügbar und damit eine der besonderen Stärken von Java.

Mit Reflection lassen sich schwierige Vorgänge effektiv und elegant codieren. Voraussetzung für die Anwendung von Reflection ist die Kenntnis generischer Klassen. Dies zeigt auch der folgende Code, der in den Beispielen jpr16600 bis jpr16602 im Abschnitt über Applet-Programmierung verwendet wird. Mit diesem Code gelingt der Zugriff auf Dateien von einem Applet aus:

```
class AccessInputStreamSwingWorker extends SwingWorker<InputStream,Object>
{
    @Override
    protected InputStream doInBackground() throws Exception
    {
      String path = "/daten/test.txt";
      String urlString = String.format(path);
      System.out.println("\nDateizugriff mit URL: "+urlString);
      Class<? extends AccessInputStreamSwingWorker> c = this.getClass();
      URL url = c.getResource(urlString);
      InputStream in = url.openStream();
      return in;
    }
}
```

Ein wesentliches Hilfsmittel bei der Untersuchung von Java-Programmen zur Laufzeit mit Reflection die die generische Klasse java.lang.Class<T>, die auch in obigem Beispiel auftritt. Objekte dieser Klasse repräsentieren Objekte des laufenden Java-Programms. Diese Objekte können mit den Methoden der generischen Klasse java.lang.Class<T> untersucht werden, z. Beispiel mit den Methoden

- getCanonicalName() - nennt den Namen der Klasse incl. Paketnamen
- getMethods() - gibt eine Liste der Methoden der Klasse zurück
- URL getResource(String) - gibt die Referenz auf das URL-Objekt zurück, das auf die als String angegebene Url zugreift, sofern ein solches URL -Objekt existiert

Die Anwendung von Reflection geschieht im obigen Beispiel wie folgt:

1. Die Klasse AccessInutStreamSwingWorker ist abgeleitet von der generischen Klasse javax.swing.SwingWorker, wobei als aktuelle Typparameter die Klassen InputStream und Object angegeben wurden.
2. Die in der Klasse javax.swing.SwingWorker wird mit der einer eigenen Methode protected InputStream doInBackground() throws Exception überschrieben.
3. In der Methode doInBackground() wird mit der Anweisung this.getClass() auf das aktuelle Objekt zugegriffen. Die Methode getClass() ist von der Klasse java.lang.Object geerbt und damit in allen Java-Klassen vorhanden. Der Typparameter der zugehörigen Objektvariablen der generischen Klasse **Class<?>** wird bei der Deklaration mit Kovarianz eingeschränkt auf Objekte, die von der Klasse AccessInputStreamWorker abgeleitet sind. Dies ist sinnvoll, denn gemäß Schritt 1 ist das aktuelle Objekt in der Tat ein Objekt der Klasse AccessInputStreamSwingWorker.
4. Mit der Methode URL getResource(String) wird jetzt *auf das URL-Objekt der eigenen Klasse* zugegriffen, das den Zugriff auf die angegebene URL verwaltet.

Im obigen Beispiel untersucht sich die Klasse AccessInputStreamSwingWorker selber in Bezug auf die durch Vererbung von der generischen Klasse javax.swing.SwingWorker geerbten Eigenschaften, um den Zugriff auf ein Objekt zu erhalten, dessen Referenz im aktuellen Kontext nicht verfügbar ist.

Beispiel jpr02169 ist ein weiteres Beispiel zur Anwendung von Reflection. In der Klasse bsp02169.ZahlParserReflexion werden die Eigenschaften der generischen Klasse bsp02169.ZahlParser untersucht. Die Klasse bsp02169.ZahlParser ist wie folgt codiert:

```
package bsp02169;

public class ZahlParser<T extends Number>
{
  private T zahlTyp = null;

  public ZahlParser() { }
  public ZahlParser(T zahlTyp)
  {
    this.zahlTyp = zahlTyp;
  }
```

```
  public T getZahlTyp()
  {
    return this.zahlTyp;
  }
  public void setZahlTyp(T zahlTyp)
  {
    this.zahlTyp = zahlTyp;
  }

  public void pruefeTyp()
  {
    String stringDarst = "\n*** Fehler - String-Darstellung konnte nicht ermittelt werden!
***";
    String typ = "\n*** Fehler - Typ konnte nicht ermittelt werden! ***";
    if (this.zahlTyp instanceof Integer)
    {
// Integer erbt von der Klasse Number, die als abstrakte Klasse Methoden intValue() etc.
fordert
      int x = this.zahlTyp.intValue();
      stringDarst = String.valueOf(x);
      typ = "Integer";
    }
    else if (this.zahlTyp instanceof Long)
    {
      long x = this.zahlTyp.longValue();
      stringDarst = String.valueOf(x);
      typ = "Long";
    }
    else if (this.zahlTyp instanceof Float)
    {
      float x = this.zahlTyp.floatValue();
      stringDarst = String.valueOf(x);
      typ = "Float";
    }
    else if (this.zahlTyp instanceof Double)
    {
      double x = this.zahlTyp.doubleValue();
      stringDarst = String.valueOf(x);
      typ = "Double";
    }
    System.out.println("\nDie Typpruefung hat folgenden Typ ergeben: "+typ);
    System.out.println("Die Darstellung als String lautet: "+stringDarst);
  }
}
```

Die Klasse bsp02169.ZahlParserReflexion untersucht jetzt ein Objekt der Klasse bsp02169.ZahlParser wie folgt in Bezug auf seine Eigenschaften:

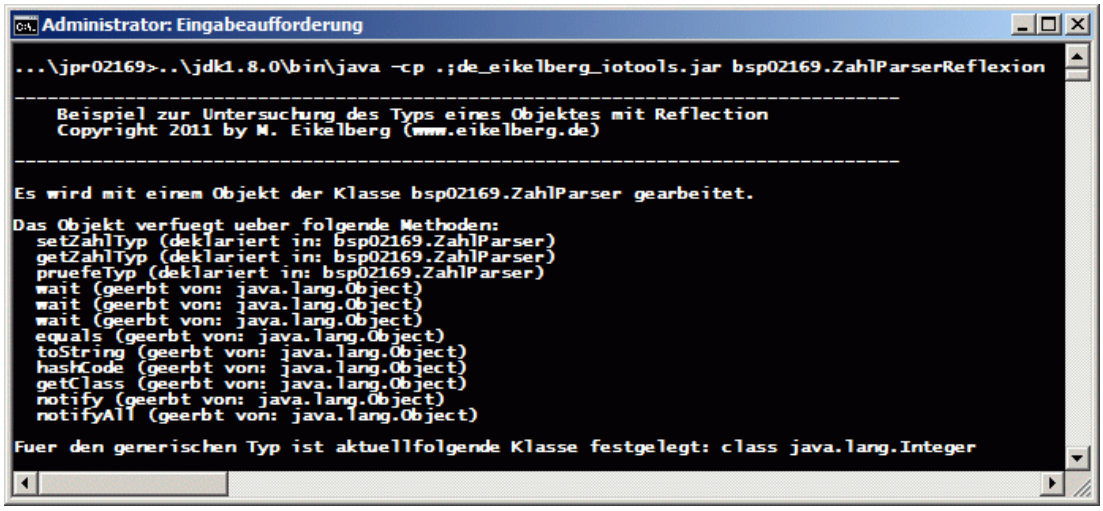
```
package bsp02169;

import java.lang.reflect.Method;
...

public class ZahlParserReflexion
{
  public static void main(String[] args)
  {
    ...

/*
 * Reflection für ein Objekt der generischen Klasse ZahlParser<T extends Number>
 */
// Objekt der generischen Klasse anlegen und dabei mit der abstrakten Klasse Number
// (Superklasse von Integer etc.) parametrisieren
    ZahlParser<Number> zp = new ZahlParser<Number>();

// Objekte der generischen Klasse Class stellen Java-Klassen dar. Im vorliegenden Fall
// ergibt zp.getClass() die generische Klasse ZahlParser.
    Class<?> c = zp.getClass();

// Reflection: Jetzt können Eigenschaften der Klasse c erkundet werden. Z. Bsp. ergibt
// die Methode getCanonicalName() den Namen der Klasse c, hier also bsp02163.ZahlParser
// und es kann eine Liste der Methoden der Klasse erstellt werden:
    System.out.println("\nEs wird mit einem Objekt der Klasse "+c.getCanonicalName()+"
```

```
gearbeitet.");
    Method[] methodenListe = c.getMethods();

    System.out.println("\nDas Objekt verfuegt ueber folgende Methoden: ");
    for (Method m : methodenListe)
    {
      System.out.print("  "+m.getName());
      String klassenName = m.getDeclaringClass().getCanonicalName();
      if (klassenName.equals(c.getCanonicalName()))
        System.out.println(" (deklariert in: "+klassenName+")");
      else
        System.out.println(" (geerbt von: "+klassenName+")");
    }

// Klasse für den generischen Typ festlegen und anschließend den Typ der Klasse feststellen
    Integer iC = new Integer(100);
    zp.setZahlTyp(iC);
    System.out.println("\nFuer den generischen Typ ist aktuellfolgende Klasse festgelegt:
"+zp.getZahlTyp().getClass());
  }
}
```

Die Verarbeitung des Programms bsp02169.ZahlParserReflexion geschieht wie folgt:

- Mit der Methode von der Klasse java.lang.Object geerbten Methode getClass() wird zunächst auf das Objekt zp der generischen Klasse ZahlParser<T extends Number> zugegriffen. Die Referenz auf das Objekt wird in einer Variablen vom generischen Typ Class<?> gespeichert, wobei der Typparameter nicht eingeschränkt wird.
- Mit der Methode getMethods der generischen Klasse Class<?> wird eine Liste der Methoden geholt. Diese Liste wird in Form eines Arrays von Objekten der Klasse java.lang.reflect.Method zurückgegben.
- Die in der Liste der Methoden aufgeführten Methoden werden mit Hilfe der Methode getName() und getDeclaringClass() in Bezug auf ihren Namen und die Klasse untersucht, in der sie codiert wurden. Diese Untersuchung wird insofern genauer ausgeführt, als das der Name der Klasse, in der sie codiert wurden mit dem Namen der untersuchten Klasse verglichen wird um festzustellen, welche Methoden geerbt wurden und welche Methoden in der untersuchten Klasse implementiert wurden.
- Zum Abschluss wird das Objekt daraufhin untersucht, welcher Datentyp aktuell für den Typparameter eingesetzt wurde.

Der nachfolgende Bildschirmausdruck zeigt die erfolgreiche Untersuchung der Klasse bsp02169.ZahlParser in der main-Methode der Klasse bsp02169.ZahlParserReflexion:



Nicht als Bezeichner verwendbare Zeichenketten

Die folgenden Zeichenketten können in einem Java-Programm *nicht* als Bezeichner (identifier) eingesetzt werden. Dabei gibt es formal zwei verschiedenen Typen von Ausnahmen: Schlüsselwörter (keywords) und Literale.

Schlüsselworter

- abstract
- boolean
- break
- byte
- case
- catch
- char
- class
- const (nur reserviert, wird in Java jedoch nicht verwendet)
- continue
- default
- do
- double
- else
- extends
- final
- finally
- float
- for
- goto (nur reserviert, wird in Java jedoch nicht verwendet)
- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- package
- private
- protected
- public
- return
- short
- static
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- try
- void
- volatile
- while

Literale

- false
- null
- true

Anleitung zur Installation und Verwendung des Programms HIEW

Anleitung für Windows XP (32 bit)

Unter Windows XP kann das Programm HIEW von der URL

ftp://pc132w.bsnw.fb4.fh-bochum.de/eikelberg/javakurs/download/Hiew/Version 5.91 fuer DOS und NT

heruntergeladen werden. Es ist empfehlenswert, die Version 5.9.1 zu verwenden. Beachten Sie ferner, dass das Programm eine Eingabeaufforderung in Originalkonfiguration (Rasterschriftart 8*12, Fensterpuffergröße und Fenstergröße nicht erhöht, also auf 80 Zeichen Breite und 25 Zeilen eingestellt) erwartet. Änderungen der Konfiguration der Eingabeaufforderung können zu einer Fehlermeldung "unsupported videomode" führen.

Gehen Sie zur Installation von HIEW 5.91 wie folgt vor - dabei beschreibt die Anleitung die Installation in ein Laufwerk D:

> Wenn Sie HIEW im Rahmen des Praktikums auf einem Windows XP Rechner installieren wollen, so müssen Sie das Programm im Verzeichnis E:\$daten\hiew (statt D:\hiew) installieren.

1. Laden sie die Datei
   ftp://pc132w.bsnw.fb4.fh-bochum.de/eikelberg/javakurs/download/Hiew/Version 5.91 fuer DOS und NT/hiew591.zip
   herunter und speichern Sie das ZIP-Archiv in einem Verzeichnis D:\hiew. Entpacken Sie das Archiv hiew591.zip so, dass die im Archiv enthaltenen Dateien wieder direkt im Verzeichnis D:\hiew gespeichert werden. Beim Entpacken entsteht ein weiteres ZIP-Archiv HIEW_591.ZIP. Entpacken Sie dieses Archiv erneut so, dass die im Archiv enthaltenen Dateien wieder direkt im Verzeichnis D:\hiew gespeichert werden. Bei diesem Vorgang ensteht eine Datei HIEW.exe. Diese Datei enthält das ausführbare Programm.

Hinweise:

- Die oben beschriebene Installation von HIEW kann in jedes beliebige Laufwerk erfolgen, für das Sie die für eine Programminstallation sowie Programmausführung erforderlichen Rechte besitzen.

Anleitung für Windows 7 (64 bit)

Für die Verwendung unter Windows 7 sollte wie für die Verwendung unter Windows XP die Version 5.9.1 verwendet werden. Sie kann gefunden werden unter der ULR

ftp://pc132w.bsnw.fb4.fh-bochum.de/eikelberg/javakurs/download/Hiew/Version 5.91 fuer DOS und NT

HIEW ist jedoch nicht direkt unter Windows 7 lauffähig. Vielmehr muss als Programm "zwischen" Windows 7 und HIEW das Programm DosBox verwendet werden. Sie können dieses Programm von der URL

ftp://pc132w.bsnw.fb4.fh-bochum.de/eikelberg/javakurs/download/DosBox0.72

herunterladen.

Gehen Sie zur Installation von HIEW 5.91 und DosBox 0.72 wie folgt vor - dabei beschreibt die Anleitung die Installation in ein Laufwerk D:

> Wenn Sie HIEW mit DosBox im Rahmen des Praktikums auf einem Windows 7 Rechner installieren wollen, so müssen Sie die Programme in Verzeichnissen E:\$daten\hiew (statt D:\hiew) bzw. E:\$daten\DosBox0.72 (statt D:\DosBox0.72) installieren.

1. Laden sie die Datei
   ftp://pc132w.bsnw.fb4.fh-bochum.de/eikelberg/javakurs/download/Hiew/Version 5.91 fuer DOS und NT/hiew591.zip
   herunter und speichern Sie das ZIP-Archiv in einem Verzeichnis D:\hiew. Entpacken Sie das Archiv hiew591.zip so, dass die im Archiv enthaltenen Dateien wieder direkt im Verzeichnis D:\hiew gespeichert werden. Beim Entpacken entsteht ein weiteres ZIP-Archiv HIEW_591.ZIP. Entpacken Sie dieses Archiv erneut so, dass die im Archiv enthaltenen Dateien wieder direkt im Verzeichnis D:\hiew gespeichert werden.
2. Laden Sie die Dateien
   ftp://pc132w.bsnw.fb4.fh-bochum.de/eikelberg/javakurs/download/DosBox0.72/DOSBox_Portable_0.72_Rev_2.paf.exe
   ftp://pc132w.bsnw.fb4.fh-bochum.de/eikelberg/javakurs/download/DosBox0.72/dosbox.conf
   herunter und speichern Sie diese Dateien in einem Verzeichnis D:\DosBox0.72\download. Starten Sie die Datei DOSBox_Portable_0.72_Rev_2.paf.exe und installieren Sie die Software in ein Verzeichnis D:\DosBox0.72\install.
3. Kopieren Sie die Datei D:\DosBox0.72\download\dosbox.conf nach D:\DosBox0.72\install\Data\settings und überschreiben Sie dabei die gegebenenfalls dort bereits vorhandene Datei. Falls Sie noch keinen Ordner settings finden, beachten Sie bitte folgenden Hinweis: Direkt nach Ausführung der Datei DOSBox_Portable_0.72_Rev_2.paf.exe existiert der Ordner settings noch nicht, denn der Ordner settings wird erst beim ersten Start des Programms DOSBoxPortable.exe erstellt. Legen Sie in diesem Fall den Ordner settings im Ordner Data manuell an und fügen Sie die heruntergeladene Datei D:\DosBox0.72\download\dosbox.conf in den neu angelegten Ordber D:\DosBox0.72\install\Data\settings ein.
4. Der Start von HIEW 5.91 mit Ausführung in DosBox 0.72 erfolgt durch Doppelklick auf folgende Datei: D:\DosBox0.72\install \DOSBoxPortable.exe

Hinweise:

- Bei der vorliegenden Konfiguration werden in der DosBox die Laufwerke D: (für die Programme) und E: (für die Daten) gemountet. Werden andere Laufwerksbuchstaben verwendet, muss eine Anpassung der Datei dosbox.conf erfolgen (Änderung / Ergänzung der mount-Befehle).
- Das Programm Hacker's View finden Sie auf sowohl auf dem FTP-Server PC132W.bsnw.fb4.fh-bochum.de unter den oben angegebenen URLs sowie auf der DVD.

Wenn Sie HIEW mit DosBox im Rahmen des Praktikums auf einem Windows 7 Rechner installieren wollen, so müssen Sie die Programme in Verzeichnissen E:\$daten\hiew (statt D:\hiew) bzw. E:\$daten\DosBox0.72 (statt D:\DosBox0.72) installieren. Die Datei dosbox.conf muss in diesem Fall so abgeändert werden, dass sie mit folgenden Anweisungen endet:

Einträge bei normaler Installation:	Einträge bei Installation in Laufwerk E:
`# mount c: c:\bat`	`mount e: e:\`
`mount d: d:\`	`echo Hiew starten...`
`mount e: e:\`	`e:`
`mount j: j:\`	`cd $daten`
`# mount f: f:\`	`cd hiew`
`echo Hiew starten...`	`hiew.exe`
`d:`	
`cd hiew`	
`hiew.exe`	

- Eine Installation von HIEW und DosBox auf einem Windows 7 Rechner in Laufwerk C: kann analog zur Beschreibung der Installation der beiden Programme im Rahmen des Praktikums in Laufwerk E: erfolgen.