Grundlagen der Datenverarbeitung II

Version 20. Juni 2003

SS 03

Prof. Dr. Sven Eric Panitz
TFH Berlin

Der Inhalt dieses Skriptes wurde im Laufe der Vorlesung im WS2002/03 entwickelt. Das Skript entstand zur Vorbereitung auf das SS03.

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird. Der XML-Quelltext verweist direkt auf ein XSLT-Skript, daß eine HTML-Darstellung erzeugt, so daß ein entsprechender Browser mit XSLT-Prozessor die XML-Datei direkt als HTML-Seite darstellen kann.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Ziel der Vorlesung

Diese Vorlesung soll über die Benutzung des Computers mit den gängisten Werkzeugen hinausgehen und sich mit den Interna hinter den Kulissen beschäftigen. Hierzu wird ein Einblick in die Programmierung gegeben. Ziel ist nicht, eine komplette Programmierausbildung zu geben (was in einer einsemestrigen zweistündigen Vorlesung auch nicht möglich wäre), sondern mit den Grundprinzipen der Programmierung vertraut zu machen. Hierzu gibt es zwei Hauptkapitel:

- als Beispiel für eine Programmiersprache werden an kleinen Javaprogrammen Konzepte wie Unterprogramme, Variablen, zusammengesetzte Befehle und Datentypen eingeführt.
- als Beispiel für ein Datenformat und die Strukturierung von Dokumenten werden die Grundprinzipien von XML vorgestellt.

In den Übungen wird der Umgang mit der Kommandozeile und die Benutzung von Kommandozeilenprogrammen erlernt.

Es werden in dieser Vorlesung keine speziellen Werkzeuge oder Anwendungsprogramme vorgestellt.

1.2 Betriebssysteme

Der Rechner, der auf unseren Schreibtisch steht, ist unsere Hardware. Er besteht in der Regel aus einem Prozessor, Hauptspeicher, einer Festplatte und eine Reihe von Peripheriegeräten zur Ein- und Ausgabe wie Bildschirm, Tastatur, Maus, Drucker, Scanner, Modem etc.

Wir wollen uns dem Computer mit der Sicht eines Programmierers nähern. Hierzu machen wir uns zunächst ein wenig mit der Funktionsweise des Rechners vertraut:

Wenn wir den Computer anschalten, so ist er so konstruiert, daß er auf verschiedenen Betriebsmitteln nach einem ladbaren Programm sucht; dieses Programm ist das Betriebssystem. In der Regel findet er auf der Festplatte das Betriebssystem und startet dieses.

Ein Betriebssystem ist also nichts weiter als ein Programm. Dem Computer als Hardware ist egal, was das Betriebssystem als Software für ein Programm ist. Für ihn ist es lediglich eine Folge von Nullen und Einsen, die Befehle für seinen Prozessor darstellen. Wenn das Betriebssystem gestartet wird, so wird diese Folge von Nullen und Einsen in den Hauptspeicher geladen und nach und nach dem Prozessor als Befehle gegeben.

Dem Prozessor ist egal, was für Befehle er ausführt. Theoretisch könnten wir auf der Festplatte ein Programm als Betriebssystem installieren, das nicht die Funktionalität eines Betriebssystems hat, sondern z.B. ein Computerspiel ist.[1]

1.2.1 Aufgaben eines Betriebssystems

Was sind die Aufgaben eines Betriebssystems. Diese Frage ist nicht pauschal zu beantworten und in gewisser Weise eine Definitionsfrage, manchmal sogar eine juristische Frage. Gehört zu einem Betriebssystem eine graphische Benutzeroberfläche? Was sind noch die Aufgaben des Betriebssystems und was ist Anwendungssoftware? Gehört zu einem Betriebssystem ein Webbrowser? Diese Ansicht vertrat die Firma Microsoft als sie ihren Webbrowser Internet Explorer zu einem integralen Bestandteil des Betriebssystems Windows machen wollte. Die Firma Netscape, deren Hauptprodukt ein Webbrowser war, klagte gegen diese Ansicht.

Die Hauptaufgabe eines Betriebssystems ist, Betriebsmittel zu verwalten und Programme zu starten. Alle weiteren Aufgaben, wie z.B. eine graphische Benutzeroberfläche, Benutzerverwaltung, nebenläufige Programmausführung usw. sind im Prinzip schon Luxus.

Von einem minimalen Betriebssystem sollte erwartet werden, daß es ein Eingabegerät (die Tastatur) und ein Ausgabegerät (den Bildschirm) steuert, ein Dateisystem auf der Festplatte verwalten kann und die Möglichkeit hat, Anwendungsrogamme zu starten.

Wenn das Betriebssystem läuft und den Benutzer erlaubt z.B. über die Kommandozeile ein Anwendungsprogramm zu starten, dann sorgt das Betriebssystem dafür, daß dieses Anwendungsprogramm, das wieder eine Folge von Nullen und Einsen ist, in den Hauptspeicher geladen wird. Anschließend werden dem Prozessor nicht mehr die Befehle des Betriebssystems zum Ausführen gegeben, sondern die Befehle des Anwendungsprogramms. Das Betriebssystem unterbricht also die Abarbeitung seiner Befehle zu Gunsten der Befehle des Anwendungsprogrammes. Wenn dieses komplett abgearbeitet ist, dann fährt das Betriebssystem wieder mit der Ausführung seiner eigenen Befehle fort.

Auf diese Weise läßt sich immer nur ein Programm zur Zeit abarbeiten und der Computer kann während dieser Zeit nichts anderes für uns tun. Ein Betriebssystem kann anbieten Prozesse zu verwalten.

[1] Mehr oder weniger ist dieses sogar bei Spielekonsolen der Fall und umgekehrt ist es Tüftlern gelungen auf Spielekonsolen auch komplette Betriebssysteme zu installieren.

Von heutigen Betriebssystemen sind wir gewohnt, daß sie scheinbar mehrere Programme gleichzeitig laufen lassen können und zur selben Zeit auch noch Betriebssystemaufaben wahrnehmen können. Hierzu verwaltet das Betriebssystem mehrere Prozesse und für die verschiedenen Anwendungsprogramme. Das Betriebssystem sorgt dafür, daß in sehr kurzen Zeitabständen zwischen den Prozessen umgeschaltet wird, so daß nacheinander jeder wieder Gelegenheit bekommt, einige seiner Befehle dem Prozessor zur Verarbeitung zukommen zu lassen. Wegen der hohen Geschwindigkeit des Prozessors erscheint es uns so, als liefen mehrere Programme gleichzeitig. Man spricht von Nebenläufigkeit.

Über dieses minimale Szenario bieten heutige Betriebssysteme eine große Anzahl weiterer Funktionalität an. Eine wichtige Funktion ist die Bereitstellung von Bibliotheken, die Anwendungsprogramme benutzen können. Heutige Anwendungsprogramme programmieren nicht alle Funktionalität selbst, sondern nutzen von dem Betriebssystem angebotene Programmteile, z.B. um ein Fenster auf dem Bildschirm darzustellen. Im Anwendungsprogramm stehen dann nicht die einzelnen Befehle, was auf dem Bildschirm darzustellen ist, sondern ein Aufruf an das Betriebssystem, ein Fenster bestimmter Größe und bestimmten Inhalts darzustellen. Der Effekt ist, daß alle Programme, die diese Funktionen des Betriebssystems wahrnehmen, optisch und funktional gleiche vom Betriebssystem verwaltete Fenster haben; aber die Programme sind damit auch an ein bestimmtes Betriebssystem gebunden und können nicht ohne Aufwand auf ein anderes Betriebssystem übertragen werden. Die Anwendungsprogramme werden damit abhängig von dem Betriebssystem, für das sie geschrieben wurden. Auch hierbei führt es immer wieder zu juristischen Fragen: eine Firma, die sowohl ein Betriebssystem als auch Anwendungsprogramme entwickelt, kann sich einen Vorteil vor anderen Firma verschaffen, die auch Anwendungssoftware für dieses Betriebssystem anbieten: sie kann bestimmte interne Funktionalität des Betriebssystems nutzen, die anderen Firma nicht offengelegt wurden.

1.2.2 Beispiele für Betriebssysteme

Obwohl es auf unseren Schreibtischen machnmal so aussieht, als gäbe es nur ein wichtiges Betriebssystem, so ist die Landschaft von Betriebssysteme doch vielfältiger als sie scheinen mag.

- **Unix** ist ein schon recht altes und sehr stabiles Betriebssystem, das von Anfang an auf mehrere Benutzer und mehrere nebenläufige Prozesse ausgelegt war. Aus diesen Gründen war lange Zeit Unix das einzige Mittel der Wahl für Server und Workstations in Rechnernetzen. Viele Firma bieten ein Unix Betriebssystem an, z.B. *Hewlett Packet* HPUX, *Sun* Solaris oder *IBM* Aix.
- **Linux** ist im Prinzip eine freie Implementierung eines Unix-Betriebssystems.
- **DOS (disk operating system)** ist ein rein Kommandozeilen basiertes Betriebssystem. Die Firma IBM hat Anfang der 80er Jahre entschieden DOS für ihre PCs zu nutzen. Es gab mehrere Anbieter von DOS als Betriebssystem. IBM entschied sich für MS DOS, das die Firma Microsoft vertrieb, nachdem sie es von einer anderen Firma aufgekauft hatte.

- **Windows** ist ein graphischer Aufsatz auf DOS. Der Kern eines klassisches Windows, wie z.B. Windows 3.11, Windows 98, Windows ME, ist ein DOS System das als zusätzliche Software eine graphische Steuerung des Betriebssystems anbietet.

- **Windows NT** ist technisch ein komplett anderes Betriebssystem als die oben genannten. Varianten sind: Windows 2000, Windows XP. Dieses System ist konsequent auf Mehrbenutzer und nebenläufige Prozesse ausgerichtet und kommt damit Unix-Systemen sehr nahe.[2]

- **Mac OS** ist das Betriebssystem der Rechner der Firma Apple. 1982 kam der ersten Macintosh-Computer mit MAC OS auf den Markt. Die Firma Apple setzte konsequent auf ein System mit graphischer über Mauseingaben zu steuernden Benutzerführung für den Endverbraucher. In Qualität und Zuverlässigkeit war die Firma Apple in diesem Marktsegment damit über ein Jahrzehnt lang bis in die zweite Hälfte der 90er Jahre ungeschlagen. Am meisten Umsatz machte die Firma Microsoft zunächst mit Anwendungssoftware für den Macintosh. Die letzten Versionen dieses Systems waren: Mac OS 7.5, Mac OS 8, Mac OS 9.

- **Mac OS X** ist technisch eine komplette Neuentwicklung zu den Vorgängerversionen des Apple Betriebssystems. Der Kern ist ein Unix-System, so daß Zuverlässigkeit gesichert ist. Die graphische Oberfläche ist aber eine gewohnte Macintoshumgebung, die eine leichte Benutzung für den Endverbraucher garantiert.

- **Nextstep** war der erste Versuch ein System in der technischen Qualität von Unix und der Benutzerfreundlichkeit eines Apple zu kreieren. Im Prinzip das, was heute mit Mac OS X erreicht ist.

- **BeOS** wurde als technisch ausgereiftere Alternative zum allgegenwärtigen Windows entwickelt und viel bewundert, hat sich aber nie im Massenmarkt durchsetzen können.

- **OS/2** war der Versuch von IBM eine technisch solidere Alternative zu Windows zu etablieren. Besonders im Finanzbereich fand OS/2[3] einige Verbreitung. Marktechnisch scheint IBM den Vertrieb von OS/2 nicht konsequent genug gegen Windows verfolgt zu haben.

- **BSD** ist ebenso wie Linux eine freie Unix-Variante. Es gibt drei große Hauptstränge der BSD-Familie: NetBSD, OpenBSD, FreeBSD

- **TOS** war Mitte der 80er Jahre das System der Rechner der Firma Atari. Rechner und Betriebssystem ahmten die von Mac OS gewohnte graphische Benutzerführung nach, waren aber weitaus billiger aber auch fehleranfälliger.

[2]Es ist wohl offiziell nie bekannt gegeben worden, wofür das NT im Namen steht. Es wird vermutet, daß der Name abgekürzt **WNT** sich aus **VMS** ableitet, indem für jeden Buchstaben von VMS der nächsthöhere Buchstabe des Alphabets gewählt wurde. Der Chefentwickler von Windows NT war vorher Entwickler des Betriebssystems VMS. Ähnlich soll der Computer *HAL* in dem Film 2001 aus den Buchstaben *IBM* abgelitten worden sein.

[3]Spötter und Liebhaber sprechen gern von OS-Halbe.

- **Amiga** mit dazugehörigen Betriebssystem ware Mitte der 80er Jahre das Angebot der durch den C64 legendär gewordenen Firma Commodore und konnte sich auf Dauer ebenso wie Atari nicht auf dem Markt behaupten, hatte aber verblüffend lange noch eine eingeschworene Fangemeinde.

1.3 Das Dateisystem

Eine der Hauptaufgaben des Betriebssystems, ist es, das Dateisystem zu verwalten. Dateien sind logische Einheiten von Daten, die auf einem Speichermedium (Festplatte, CD-Rom, Floppy) abgespeichert sind. Dateien haben Namen. Zusätzlich ist das Dateisystem hierarchisch in Ordner geglieder und hat somit eine Baumstruktur. Es gibt einen Wurzelordner, in dem das komplette Dateisystem liegt. Ein Ordner kann als Inhalt weitere Unterordner und Dateien enthalten. Diese hierarchische Baumstruktur kennen so gut wie alle Betriebssysteme. Graphische Benutzeroberflächen stellen diese Baumstruktur in naheliegender Weise dar, wie im nebenstehenden Bildschirmausschnitt aus Linux zu sehen ist.

In Unix werden alle Dateien, die auf unterschiedlichen Medien, wie Festplatten DVDs etc. liegen, in ein Dateisystem integriert (*mounting*) in Windows werden die verschiedenen Medien durch Laufwerksbuchstaben unterschieden.

Um den genauen Ort einer Datei zu beschreiben, reicht es also nicht aus, den Namen der Datei zu kennen; sondern der Ordner, in dem es liegt, und wiederum dessen Lage bis hin zur Wurzel sind anzugeben.

Man spricht vom Pfad im Dateibaum, der zu einer Datei führt. Der Pfad gibt an, über welche Ordner von der Wurzel aus gehend, man zu einer Datei gelangt. In der Notation, wie ein Pfad beschrieben wird, unterscheiden sich die wichtigsten Betriebssysteme leicht. Zum Trennen der Ordnerangaben benutzt Unix einen Schrägstrich `/`, Windows einen rückwärtigen Schrägstrich `\` und Apple einen Doppelpunkt.[4]

Im obigen angegebenen Dateissystem läßt sich eine Datei mit ihrem vollem Pfad in diesen drei Betriebssystemen wie folgt angeben:

- **Unix**: `/docs/api/java/index.html`
- **Windows**: `c:\docs\api\java\index.html`
- **Apple**: `:macintosh HD:docs:api:java:index.html`

Dateien und Ordner haben Namen. Was für Namen erlaubt sind, welche Länge sie haben dürfen und ob Groß- und Kleinschreibung relevant ist, ist abhängig vom Betriebssystem. In Unix ist Groß-/Kleinschreibung relevant, bei Apple und in Windows nicht. DOS kannte nur Namen mit 8 Zeichen.

Eine mit Punkt angehängte Erweiterung des Namens kann angeben, um was für eine Art von Datei es sich handelt, also im Prinzip, wie die Folge von Nullen und Einsen, die gespeichert ist, zu interpretieren ist.[5]

[4] Was der Endbenutzer beim Apple nie zu sehen bekommt.

[5] Apple kennt diese Namenserweiterung traditionell nicht. Dort gibt es eine interne Speicherung über die Art der Datei.

Abbildung 1.1: Baumdarstellung des Dateisystems

> **Hinweis:** Windows ist standardmäßig derzeit so eingestellt, daß in der graphischen Darstellung des Dateissystems die Namenserweiterungen nicht angezeigt werden. Dieses ist mitunter gefährlich, denn man kann so eine ausführbare Datei leicht für etwas anderes vielleicht ein Bild ansehen und unbedarft ein trojanisches Pferd oder einen Virus öffnen. Es ist sehr zu empfehlen, diese Einstellung von Windows zu ändern.

Über den Namen und die Ordnerstruktur hinaus, können noch weitere Informationen über Dateien im Betriebssystem abgespeichert werden. So können bestimmte Dateien Benutzern zugeordnet sein und verschiedene Berechtigungen für verschiedene Benutzer haben. Desweiteren speichert das Betriebssystem zu jeder Datei ab, wann sie das letzte Mal geändert wurde und wie groß sie ist.

1.4 Umgang mit der Kommandozeile

Der rudimentärste Umgang mit einem Betriebssystem ist über eine Kommandozeile. Frühere Betriebssysteme kannten nur die Kommandozeile zur Steuerung, so insbesondere die verschiedenen DOS-Systeme. Nur wenige Betriebssystem haben überhaupt keine Kommandozeileneingabe. Die Betriebssystem MacOS bis einschließlich Version 9 kannten keine Kommandozeile[6].

Eine Kommandozeile erwartet die Eingabe eines Befehls an das Betriebssystems über die Tastatur. Ein Befehl wird mit dem Drücken der `ENTER`-Taste aktiviert.

Das Bildschirmfoto in Abbildung 1.4 zeigt eine Eingabekonsole im Betriebssystem Windows. In Windows ist die Standardeinstellung so, daß der Hintergrund schwarz und die Schrift weiß ist. Diese Farbeinstellung kann geändert werden.

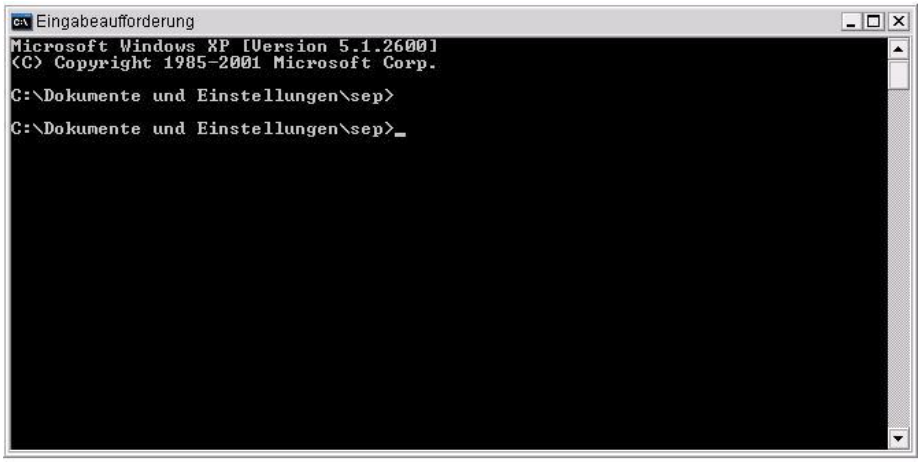

Abbildung 1.2: Kommandozeile in Windows

Die Eingabekonsole befindet sich immer in einem bestimmten Ordner, auf den sich alle Befehle beziehen. Wird ein Dateiname ohne explizite komplette Angabe eines Pfades von der Wurzel des Dateibaums gemacht, bezieht sich die Angabe auf den aktuellen Ordner, in dem sich die Konsole befindet. Das aktuelle Verzeichnis wird bei Windows standardmäßig in der Eingabeaufforderung angegeben.

1.4.1 Befehle auf der Kommandozeile

Die gängigsten Befehle, die Unix-artige und DOS-basierte Betriebssysteme kennen, beschäftigen sich mit dem Dateisystem und dem Starten von Programmen.

[6] Zumindest nicht für den Endanwender. Anwendungsentwicklern standen durchaus Werkzeuge mit einer Kommandozeile zur Verfügung.

Es folgt eine kleine tabellarische Übersicht wichtiger Kommandos.

DOS	**Unix**	**Beschreibung**
`dir`	`ls`	zeigt die Dateien im aktuellen Verzeichnis an
`cd` *verzeichnispfad*	`cd` *verzeichnispfad*	wechselt zum Verzeichzeichnis, das durch den *verzeichnispfad* gekennzeichnet ist. `..` steht dabei für den nächsthöheren Ordner.
progname	*progname*	startet das Programm *progname*, sofern dieses auf dem Pfad existiert
	`touch` *dateiname*	aktualisiert den Zeitstempel der Datei *dateiname*. Falls eine solche Datei nicht existiert, wird sie mit leerem Inhalt angelegt.
`del` *dateiname*	`rm` *dateiname*	löscht die Datei *dateiname* komplett aus dem Dateisystem. Es gibt in der Regel keine Sicherheitsnachfrage. Die Datei befindet sich hinterher auch nicht in einem Papierkorb.
`ren` *datei1 datei2*	`mv` *datei1 datei2*	Nenne die Datei *datei1* um in *datei2*. Danach gibt es keine Datei *datei1* mehr im Dateisystem.
`cp` *datei1 datei2*	`cp` *datei1 datei2*	Kopiere die Datei *datei1* in *datei2*. Danach gibt es weiterhin die Datei *datei1* im Dateisystem.
`mkdir` *ordner*	`mkdir` *ordner*	Erzeuge einen neuen Ordner.
`chmod`*mods file*	`chmod` *mods file*	Ändert die Zugriffsberechtigung der Datei oder des Ordners *file* nach den in *mods* angegebene Schema. Benutzer werden hier spezifiziert nach: `u` für den Besitzer der Datei, `g` für die Gruppe, der die Datei zugeordnet ist, `o` für alle weiteren Benutzer und `a` für alle Benutzer. Es gibt Berechtigungen wiew zum Schreiben, `r` zum Lesen und `x` zum Ausführen. Ein Plus- oder Minuszeichen vergibt Rechte oder nimmt sie wieder. z.B. `chmod a+r myFile` erlaubt allen Benutzer die Datei `myFile` zu lesen.

Wie man sieht, liegen DOS und Unix in diesem Fall nicht sehr weit auseinander.

Die Befehle kenne zusätzlich noch weitere Optionen, diese werden dem Befehl durch Leerzeichen getrennt nachgestellt. Optionen sind in der Regel in

Unix durch ein vorangestelltes Minuszeichen und in Windows durch einen Schrägstrich gekennzeichnet. So kennt der Befehl `ls` z.B. die Option `-l` bzw. `/l`, um eine detaillierte Darstellung der Dateiinformation zu geben, oder die Option `-a` um auch versteckte Dateien anzuzeigen.

Es gibt auf der Kommandozeile auch Möglichkeiten, mehr als eine Datei zu beschreiben. Hierzu bedient man sich des Zeichens `*`, das 0 bis n beliebige Zeichen stehen kann.

Die Kommandoeingabe bietet eine Hilfe an. Hierzu kann man den Befehl `help` benutzen.

Beispiel:
Ein paar kleine Beispiele für Befehle auf der Kommandozeile:

- Liste alle Dateien im aktuellen Ordner mit der Erweiterung `.java` auf.

```
sep@swe10:~/fh/internal/beispiele/jugs> ls *.java
Jugs.java             Main.java                        OptionsDialog.java
JugsClassLoader.java  MainJugsTestClass.java
JugsGui.java          MainJugsTestClassPrototype.java
sep@swe10:~/fh/internal/beispiele/jugs>
```

- Erzeuge den neuen Unterordner `classes`.

```
sep@swe10:~/fh/internal/beispiele/jugs> mkdir classes
sep@swe10:~/fh/internal/beispiele/jugs>
```

- Verschiebe alle Dateien mit Erweiterung `.class` aus den aktuellen Ordner in den Unterordner `classes`.

```
sep@swe10:~/fh/internal/beispiele/jugs> mv *.class classes/
sep@swe10:~/fh/internal/beispiele/jugs>
```

- Starte das Programm `netscape` durch Angabe seines vollständigen Pfadnamens.

```
sep@swe10:~/fh/internal/beispiele/jugs> /usr/local/netscape/netscape
```

1.4.2 Skripte

Wie man sehen kann, indem man das Kommando `help` in der Kommandozeile eingibt, kennt die Kommandozeile schon eine Vielzahl von Befehlen. Damit stellt die Kommandozeile schon eine Art Programmiersprache dar. Ein Kommandozeilenprogramm besteht aus einer Folge von Befehlen, die die Kommandozeile versteht. Solche Programme können geschrieben werden. Hierzu schreibt man mit einem einfachen Texteditor eine Datei mit der Endung `.sh` in Unix oder `.bat` in Windows. In dieser Datei können nacheinander die Befehle an die Eingabekonsole geschrieben werden. Die Datei wird abgespeichert. Anschließend gibt man ihr mit dem Befehl `chmod a+x` die Berechtigung zum Ausführen. Nun läßt sich die Textdatei als Programm starten.

Beispiel:
Als Beispiel schreiben wir ein kleines zweizeiliges Skript, das erst einen neuen Ordner erzeugt, und dann die Dateien mit der Endung `.class` in diesen Ordner verschiebt.

```
1 mkdir classes
2 mv *.class classes
```

Diese Skript speichern wir unter den Namen `mvClasses.sh` ab. Schließlich setzen wir die Berechtigung zum Ausführen und führen das Programm aus.

```
sep@swe10:~/fh/internal/beispiele/jugs> chmod a+x mvClasses.sh
sep@swe10:~/fh/internal/beispiele/jugs> mvClasses.sh
sep@swe10:~/fh/internal/beispiele/jugs>
```

1.4.3 Umgebungsvariablen

Ein Betriebssystem hat Variablen, in denen es globale Werte abspeichert, wie z.B. den Rechnernamen oder den Benutzernamen. Die Gesamtheit dieser Werte nennt man Umgebung und die einzelnen Variablen Umgebungsvariablen. Über die Kommandozeile können Umgebungsvariablen abgefragt werden. Wenn man z.B. in Windows den Befehl `set` eingibt, so folgt eine Auflistung aller bekannten Umgebungsvariablen und ihrer Werte.

1.5 Programm und Maschine

Bevor wir im nächsten Kapitel anfangen eigene Programme zu schreiben, sollten wir uns zunächst eimnal darüber klar werden, was ein Programm überhaupt ist, und was eine Programmiersprache ausmacht. Ein Programm, das wir im Betriebssystem starten, sei es durch Aufruf in der Kommandozeile, sei es durch einen Doppelklick auf der graphischen Oberfläche, ist eine Folge von Nullen und Einsen. Wir sprechen von einer Binärdatei im Gegensatz zu Textdateien. In Windows hat ein Programm zumeist die Erweiterung `.exe`. Die Nullen und Einsen eines Programms kodieren direkt Befehle des Prozessors. Wenn das Programm gestartet wird, dann werden diese Befehle direkt an den Prozessor gegeben, der diese ausführt.

1.5.1 Programmiersprachen

Ein Programmierer, der ein Programm schreibt, erzeugt aber nicht diese ausführbare Binärdatei. So gut wie nie kennt der Programmierer die Befehle des Prozessors oder könnte die Binärdatei selbst erzeugen. Der Programmierer schreibt ein Programm in einer Programmiersprache.

Programme einer Programmiersprache sind Texte, die mit einem einfachen Texteditor geschrieben werden können. Die Definition der Programmiersprache legt fest, was es für Befehle und Strukturierungsmöglichkeiten gibt. Hierzu hat eine Programmiersprache eine Grammatik ganz analog zu natürlichen Sprachen. In dieser Hinsicht ist eine Programmiersprache zunächst vollkommen unabhängig von einem konkreten Computer eine formal definierte Sprache. Damit ein Programm, das in einer Programmiersprache geschrieben wurde, von einem Computer ausgeführt werden kann, muß es entweder in eine Folge von Prozessorbefehlen übersetzt werden, oder aber in irgendeiner Weise interpretiert werden.

1.5.2 Kompilierung

Zum Übersetzen einer Textdatei, die das Programm darstellt, das ein Programmierer geschrieben hat, in eine durch den Prozessor ausführbare Binärdatei benötigt man ein Werkzeug, den sogenannten Compiler. Ein Compiler bekommt als Eingabe eine Textdatei, die ein in einer Programmiersprache geschriebenes Programm enthält. Er prüft zunächst, ob das Programm nach den Regeln der Programmiersprache korrekt geschrieben wurde. Wenn die Prüfung keinen Fehler gefunden hat, erzeugt der Compiler für dieses Programm eine ausführbare Binärdatei. Ein Compiler ist ein Übersetzer von einer Programmiersprache in die Maschinensprache eines Computers. Der Compiler ist auch ein Programm, das von der Kommandozeile aus gestartet werden kann.

Beispiel:
C ist eine Programmiersprache mit einem Compiler. Folgendes kleine C-Programm gibt den Text `hallo welt` auf dem Bildschirm aus:

```
1 int main(){
2     printf("hallo welt\n");
3     return 0;
4 }
```

Dieses Programm läßt sich mit einen normalen Texteditor schreiben. Dann kann es mit dem C-compiler `gcc` übersetzt werden, um dann ausgeführt zu werden:

```
sep@swe10:~/fh/dv2> ls hallo*
hallo.c
sep@swe10:~/fh/dv2> gcc -o hallo.exe hallo.c
sep@swe10:~/fh/dv2> ls hallo*
hallo.c  hallo.exe
sep@swe10:~/fh/dv2> ./hallo.exe
hallo welt
sep@swe10:~/fh/dv2>
```

1.5.3 Interpreter

Statt ein Programm in eine Binärdatei zu übersetzen, gibt es noch eine zweite Möglichkeit, wie ein Programm zu Ausführung gelangt. Der Programmtext wird nicht in eine ausführbare Datei übersetzt sondern durch einen Interpreter Stück für Stück anhand des Quelltextes ausgeführt. Hierzu muß stets der Interpreter zur Verfügung stehen, um das Programmm auszuführen. Interpretierte Programme sind langsamer in der Ausführung als übersetzte Programme. Eine populäre interpretierte Sprache ist *Lisp*.

1.5.4 Java

Java hat eine dritte Form der Ausführung, indem eine abstrakte Maschine mit *byte code* benutzt wird. Dieses ist quasi eine Mischform aus den obigen zwei Ausführungsmodellen. Der Quelltext wird in Befehle übersetzt nicht für einen konkreten Computer, sondern für eine abstrakte Maschine. Für diese abstrakte

Maschine steht dann ein Interpreter zur Verfügung. Der Vorteil ist, daß durch die zusätzliche Abstraktionsebene der Übersetzer unabhängig von einer konkreten Maschine Code erzeugen kann und das Programm auf auf allen Systemen laufen kann, für die es einen Interpreter der abstrakten Maschine gibt, aber trotzdem schneller ist als ein rein interpretiertes Programm.

Das bedeutet insbesondere, daß man zwei Programme braucht um Javaprogramm zu übersetzen und laufen zu lassen. Es wird ein Compiler benötigt, der den Code für die abstrakte Maschine erzeugt, und es wird ein Intepreter benötigt, der die abstrakte Maschine auf den konkreten Rechner realisiert. Diese beiden Programme heißen `javac` und `java`.

1.6 Disziplinen der Programmierung

Mit dem Begriff Programmierung wird zunächst die eigentliche Codierung eines Programms assoziiert. Eine genauerer Blick offenbart jedoch, daß dieses nur ein kleiner Teil von vielen recht unterschiedlichen Schritten ist, der zur Erstellung von Software notwendig ist:

- **Spezifikation:** Bevor eine Programmieraufgabe bewerkstelligt werden kann, muß das zu lösende Problem spezifiziert werden. Dieses kann informell durch eine natürlichsprachliche Beschreibung bis hin zu mathematisch beschriebenen Funktionen geschehen. Gegen die Spezifikation wird programmiert. Sie beschreibt das gewünschte Verhalten des zu erstellenden Programms.

- **Modellieren:** Bevor es an die eigentliche Codierung geht, wird in der Regel die Struktur des Programms modelliert. Auch dieses kann in unterschiedlichen Detaillierungsgraden geschehen. Manchmal reichen Karteikarten als hilfreiches Mittel aus, andernfalls empfiehlt sich eine umfangreiche Modellierung mit Hilfe rechnergestützter Werkzeuge. Für die objektorientierte Programmierung hat sich UML als eine geeignete Modellierungssprache durchgesetzt. In ihr lassen sich Klassendiagramme, Klassenhierarchien und Abhängigkeiten graphisch darstellen. Mit bestimmten Werkzeugen wie *Together* oder *Rational Rose* läßt sich direkt für eine UML Modellierung Programmtext generieren.

- **Codieren:** Die eigentliche Codierung ist in der Regel der einzige Schritt, der direkt Code in der gewünschten Programmiersprache von Hand erzeugt. Alle anderen Schritte der Programmierung sind mehr oder weniger unabhängig von der zugrundeliegenden Programmiersprache.

  Für die Codierung empfiehlt es sich Konventionen zu verabreden, wie der Code geschrieben wird, was für Bezeichner benutzt werden, in welcher Weise der Programmtext eingerückt wird. Entwicklungsabteilungen haben zumeist schriftlich verbindlich festgeschriebene Richtlinien für den Programmierstil. Dieses erleichtert den Code der Kollegen im Projekt schnell zu verstehen.

- **Testen:** Beim Testen sind generell zu unterscheiden:

  - *Entwicklertests:* Diese werden von den Entwicklern während der Programmierung selbst geschrieben, um einzelne Programmteile (Methoden, Funktionen) separat zu testen. Es gibt eine Schule, die propagiert, Entwicklertests vor dem Code zu schreiben (*test first*). Die Tests dienen in diesem Fall als kleine Spezifikationen.
  - *Qualitätssicherung:* In der Qualitätssicherung werden die fertigen Programme gegen ihre Spezifikation getestet (*black box tests*). Hierzu werden in der Regel automatisierte Testläufe geschrieben. Die Qualitätssicherung ist personell von der Entwicklung getrennt. Es kann in der Praxis durchaus vorkommen, daß die Qualitätsabteilung mehr Mitarbeiter hat als die Entwicklungsabteilung.

- **Optimieren:** Sollten sich bei Tests oder in der Praxis Performanzprobleme zeigen, sei es durch zu hohen Speicherverbrauch als auch durch zu lange Ausführungszeiten, so wird versucht ein Programm zu optimieren. Hierzu bedient man sich spezieller Werkzeuge (*profiler*) die für einen Programmdurchlauf ein Raum- und Zeitprofil erstellen. In diesem Profil können Programmteile, die besonders häufig durchlaufen werden, oder Objekte die im großen Maße Speicher belegen, identifiziert werden. Mit diesen Informationen lassen sich gezielt inperformante Programmteile optimieren.

- **Verifizieren:** Eine formale Verifikation eines Programms ist ein mathematischer Beweis der Korrektheit bezüglich der Spezifikation. Das setzt natürlich voraus, daß die Spezifikation auch formal vorliegt. Man unterscheidet:

  - *partielle Korrektheit:* wenn das Programm für eine bestimte Eingabe ein Ergebnis liefert, dann ist dieses bezüglich der Spezifikation korrekt.
  - *totale Korrektheit:* Das Programm ist partiell korrekt und terminiert für jede Eingabe, d.h. liefert immer nach endlich langer Zeit ein Ergebnis.

  Eine formale Verifikation ist notorisch schwierig und allgemein nicht automatisch durchführbar. In der Praxis werden nur in ganz speziellen kritischen Anwendungen formale Verifikationen durchgeführt, z.B. bei Steuerungen gefahrenträchtiger Maschinen, so daß Menschenleben von der Korrektheit eines Programms abhängen können.

- **Wartung/Pflege:** den größten Teil seiner Zeit verbringt ein Programmierer nicht mit der Entwicklung neuer Software sondern der Wartung bestehender Software. Hierzu gehört die Anpassung des Programms an neue Versionen benutzter Bibliotheken, oder neuer Versionen des Betriebssystems, auf dem das Programm läuft, sowie die Korrektur von Fehlern.

- **Debuggen:** Bei einem Programm ist immer damit zu rechnen, daß es Fehler enthält. Diese Fehler werden im besten Fall von der Qualitätssicherung entdeckt, im schlechteren Fall treten sie beim Kunden auf. Um Fehler im Programmtext zu finden, gibt es Werkzeuge, die ein schrittweises Ausführen des Programms ermöglichen *(debugger)*. Dabei lassen sich die

Werte, die in bestimmten Speicherzellen stehen, auslesen und auf diese Weise den Fehler finden.

- **Internationalisieren (I18N)[7]:** Softwarefirmen wollen möglichst viel Geld mit ihrer Software verdienen und streben deshalb an, ihre Programme möglichst weltweit zu vertreiben. Hierzu muß gewährleistet sein, daß das Programm auf weltweit allen Plattformen läuft und mit verschiedenen Schriften und Textcodierungen umgehen kann. Das Programm sollte ebenso wie mit lateinischer Schrift auch mit Dokumenten in anderen Schriften umgehen können. Fremdländische Akzente und deutsche Umlaute sollten bearbeitbar sein. Aber auch unterschiedliche Tastaturbelegungen bis hin zu unterschiedliche Schreibrichtungen sollten unterstützt werden.

  Die Internationalisierung ist ein weites Feld, und wenn nicht am Anfang der Programmerstellung hierauf Rücksicht genommen wird, so ist es schwer, nachträglich das Programm zu internationalisieren.

- **Lokalisieren (L12N):** Ebenso wie die Internationalisierung beschäftigt sich die Lokalisierung damit, daß ein Programm in anderen Ländern eingesetzt werden kann. Beschäftigt sich die Internationalisierung damit, daß fremde Dokumente bearbeitet werden können, versucht die Lokalisierung das Programm komplett für die fremde Sprache zu übersetzen. Hierzu gehören Menueinträge in der fremden Sprache, Beschriftungen der Schaltflächen, oder auch Fehlermeldungen in fremder Sprache und Schrift. Insbesondere haben verschiedene Schriften unterschiedlichen Platzbedarf; auch das ist beim Erstellen der Programmoberfläche zu berücksichtigen.

- **Portieren:** Oft wird es nötig ein Programm auf eine andere Plattform zu portieren. Ein unter Windows erstelltes Programm soll z.B. auch auf Unix-Systemen zur Verfügung stehen.

- **Dokumentieren:** Der Programmtext allein reicht in der Regel nicht aus, daß das Programm von fremden oder dem Programmierer selbst nach geraumer Zeit gut verstanden werden kann. Um ein Programm näher zu erklären, wird im Programmtext Kommentar eingefügt. Kommentare erklären die benutzten Algorithmen, die Bedeutung bestimmter Datenfelder oder die Schnittstellen und Benutzng bestimmter Methoden.

  Es ist zu empfehlen, sich anzugewöhnen, Quelltextdokumentation immer auf Englisch zu schreiben. Es ist oft nicht abzusehen, wer einmal einen Programmtext zu sehen bekommt. Vielleicht ein japanischer Kollege, der das Programm für Japan lokalisiert, oder der irische Kollege, der, nachdem die Firma mit einer anderen Firma fusionierte, das Programm auf ein anderes Betriebssystem portiert, oder vielleicht die englische Werksstudentin, die für ein Jahr in der Firma arbeitet.

[7] I18N ist eine Abkürzung für das Wort *internationalization*, das mit einem i beginnt, mit einem n endet und dazwischen 18 Buchstaben hat.

Kapitel 2

Grundkonzepte der Programmierung

Mit der uns umgebenen Technik eines heutigen Computer sind wir jetzt soweit vertraut, daß wir wissen, was wir tun, wenn wir ein Programm schreiben. Wir wollen am Beispiel von Java nun einige Grundkonzpte der Programmierung kennenlernen und ausprobieren.

In der Programmierung kann man generell unterscheiden zwischen den passiven Daten und den Programmen, die diese Daten manipulieren. Daten können Zahlen oder Texte, aber auch Listen, Tabellen, ganze Dokumente, Bilder etc. sein. Programme manipulieren diese Daten, verändern, löschen oder legen sie neu an.

2.1 Module und Unterprogamme

In kaum einer Programmiersprache stehen alle Befehle des Programms in einer Folge untereinander. Es gibt zwei Hauptstrukturierungsmöglichkeiten:

- Unterprogramme: Diese sind als eigene funtionale Einheiten zu verstehen. Sie bekommen bestimmte Daten als Eingabe, die sogenannten Parameter oder auch Argumente. Sie können diese Daten verändern, und sie können ein Ergebnis berechnen. In Java heißen die Unterprogramme *Methoden*. Andere Programmiersprachen sprechen von Funktionen oder Prozeduren.
- Module: In einem Modul sammelt man in der Regel mehrere Unterprogramme, die logisch sinnvoll zusammengehören, weil sie z.B. die selbe Art von Daten manipulieren. In Java heißen die Programmmodule *Klassen*.

  Ein Javaprogramm besteht aus einer Menge von Klassen. Jede Klasse wird in einer eigenen Datei geschrieben. Der Dateiname besteht aus dem Namen der Klasse und hat die Endung `.java`. Hierbei ist Groß- und Kleinschreibung relevant.

Beispiel:
Die einfachste Klasse die man sich denken kann enthält keine Unterprogramme. Sie ist leer. Sie stellt ein leeres Modul dar. Sie besteht nur aus dem Wort `class` gefolgt vom frei wählbaren Namen der Klasse und einem Paar geschweifter Klammern:

Minimal.java

```
1 class Minimal { }
```

2.2 Die Hauptmethode

2.2.1 Das erste Java Programm

Ein Programm besteht aus einer Menge von Unterprogrammen, die in unteschiedlichen Modulen stehen können. Eines dieser Unterprogramme muß als besonderes Unterprogramm ausgezeichnet sein, nämlich als das Programm, mit dem bei der Ausführung zu beginnen ist. In Java heißt dieses Unterprogramm `main`[1]. Wenn man den Javainterpreter sagt, er solle ein Programm ausführen, dann gibt man ihn einen Klassennamen an. Der Javainterpreter erwartet dann, daß es in dieser Klasse eine Methode mit dem Namen `main` gibt. Wenn dieses der Fall ist, so führt der Javainterpreter diese Methode aus, andernfalls gibt es eine Fehlermeldung.

Beispiel:
Unser erstes Programm mit einer `main`-Methode. Dieses Programm gibt auf der Kommandozeile den Text `"hallo welt"` aus:

Hallo.java

```
1 class Hallo {
2   public static void main (String [] args){
3     System.out.println("hallo welt");
4   }
5 }
```

Aufgabe 1 Schreiben Sie das obige Programm mit einen Texteditor ihrer Wahl. Speichern Sie es als `Hallo.java` ab. Übersetzen Sie es mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `hallo.class`. Führen Sie das Programm mit dem Javainterpreter `java` aus.

Im obigen Beispiel sehen wir schon fast alles, was zur Syntax eines Javaprogramms gehört:

- es gibt eine Menge von reservierten Wörtern, sogenannten Schlüsselwörter, die fest zur Sprache Java gehören und das Programm inhaltlich strukturieren. Im Beispiel sind dieses die Wörter: `class, public, static, void.`
- es gibt eine Menge von Sonderzeichen, die zur Strukturierung eines Javaprogramms zu benutzen sind. Hierzu gehören runde `()` und geschweifte Klammern, das Semikolon und der Punkt.

[1] Dieses gilt für die meisten Programmiersprachen.

- Es gibt Namen, die Klassen bezeichnen. In unserem Beispiel einmal der Name `Hallo`, der die Klasse bezeichnet, die wir gerade selbst definieren. Zusätzlich nehmen wir auf zwei bereits existierende Klassen bezug: der Klasse `String` und der Klasse `System`. Per Konvention beginnen Klassennamen immer mit einem Großbuchstaben[2]. Ein Klassenname darf kein Leerzeichen enthalten.
- es gibt Namen, die Methoden bezeichnen. In unserem Beispiel definieren wir die Methode mit den Namen `main` und benutzen die Methode `println`. Einer Methode folgt in runden Klammern eine Liste von Parametern. Solange wir noch nicht genau wissen, was Parameter sind, ist diese Liste als leer zu denken. Methodennamen beginnen per Konvention immer mit einem Kleinbuchstaben.
- Es gibt freie Texte. Dieses sind beliebige in Anführungszeichen eingeschlossene Zeichenketten. Ein solcher Text darf nicht über eine Zeile hinausgehen. In unserem Beispiel haben wir den Text: `hallo welt`. Dieser stellt die Daten dar, mit denen unser Programm arbeitet.

Die Bedeutung der Schlüsselwörter, die vor unserer Methode `main` stehen (`public static void`) werden wir erst an späterer Stelle erklären. Man merke sich, daß die Startmethode stets mit diesen Schlüsselwörtern zu beginnen hat. Die Startmethode hat einen Parameter (`String [] args`). Auch diesen hinterfragen wir noch nicht, merken uns aber, daß er für eine Startmethode existieren muß.

Wir haben in unserem Programm bestimmte Einrückungen vorgenommen. Diese sind für Java technisch nicht notwendig, sondern sollen uns die Lesbarkeit des Programms erleichtern.

Aufgabe 2 Übersetzen Sie das Programm `Minimal` aus dem letzem Abschnitt und versuchen Sie dieses Programm laufen zu lassen. Was für eine Fehlermeldung bekommen Sie?

2.2.2 Das zweite Java Programm

Das Programmodul `Hallo` enthielt nur eine Methode und diese enthielt nur einen Befehl, nämlich `System.out.println("hallo welt")`.

Ein Unterprogramm kann mehrere Befehle enthalten. Diese Befehle werden dann nacheinander ausgeführt. Jeder Befehl endet mit einem Semikolon.

Beispiel:
Das folgende Programm enthält mehrere Befehle in der Hauptmethode. Zweimal wird ein Text ausgegeben und anschließend das Ergebnis einer arithmetischen Berechnung.

Zwei.java

```
1 class Zwei {
2
```

[2]Sofern ein Alphabet benutzt wird, daß die Unterscheidung von Groß- und Kleinschreibung kennt.

```
3   public static void main (String [] args){
4     System.out.println("hallo welt");
5     System.out.println("3*3 ist");
6     System.out.println(3*3);
7   }
8 }
```

Wie wir sehen können wir auch arithmetische Ausdrücke schreiben, deren Wert dann bei der Ausführung von Java berechnet wird.

2.3 Methoden

Bisher haben wir nur Klassen mit genau einer Methode, der Hauptmethode, geschrieben. Betrachten Sie das folgende Programm:

Drei.java

```
1 class Drei{
2   public static void main (String [] args){
3     System.out.println("*******");
4     System.out.println("3*3 ist");
5     System.out.println(3*3);
6     System.out.println("*******");
7   }
8 }
```

es enthält drei verschiedene Befehle, von denen einer zweimal verwendet wird, nämlich der Befehl:
`System.out.println("*******");`

Wann immer eine bestimmte Befehlsfolge mehrfach in einem Programm auftaucht, ist diese ein guter Kanditat in ein eigenes Unterprogramm ausgelagert zu werden. Dann kann jeweils die fragliche Befehlsfolge durch einen Aufruf an das neue Unterprogramm ersetzt werden.

2.3.1 Methodendeklaration

Wir können für bestimmten Code eine eigene Methode definieren. Die Methodendefinition unterscheidet sich dann in ihrer Art nicht von der Definition der Startmethode: wir können einen Namen für die Methode frei wählen. Vor dem Methodennamen stehen zusätzliche Informationen zur Methode, nach dem Methodennamen zunächst eine in runden Klammern eingeschlossene Liste von Parametern und schließlich in geschweiften Klammern eingeschlossen die Befehle der Methode.

Beispiel:
In der folgenden Klasse schreiben wir zur Startmethode noch eine zusätzliche Methode mit Namen `zierleiste`.

DreiPlus.java

```
1  class DreiPlus{
2    public static void main (String [] args){
3      System.out.println("*******");
4      System.out.println("3*3 ist");
5      System.out.println(3*3);
6      System.out.println("*******");
7    }
8
9    public static void zierleiste(){
10     System.out.println("*******");
11   }
12 }
```

2.3.2 Methodenaufruf

Methoden, die wir definiert haben, können jetzt an beliebiger Stelle aufgerufen werden. Ein Methodenaufruf besteht aus dem Methodennamen gefolgt von in runden Klammern eingeschlossenen konkreten Parametern.

Beispiel:
Da wir zum Drucken der Zierleiste eine eigene Methode geschrieben haben, können wir diese in der Startmethode aufrufen, anstatt ihren Code direkt zu schreiben.

DreiPlusPlus.java

```
1  class DreiPlusPlus{
2    public static void main (String [] args){
3      zierleiste();
4      System.out.println("3*3 ist");
5      System.out.println(3*3);
6      zierleiste();
7    }
8
9    public static void zierleiste(){
10     System.out.println("*******");
11   }
12 }
```

Es ist hierbei egal, in welcher Reihenfolge die Methoden einer Klasse auftreten. Eine Methode kann aufgerufen werden, auch wenn sie erst an späterer Stelle im Programm definiert wird.

2.4 Unsere ersten Typen

Bisher haben wir schon zwei verschiedene Arten von Daten kennengelernt. Man spricht dabei auch von Typen. Der eine Typ stellte beliebige Zeichenketten dar. Dieser Typ heißt `String`. In einem der Beispiel haben wir zusätzlich gesehen,

daß wir mit Zahlen rechnen können. Der Typ der ganzen Zahlen wird in Java[3] als `int` bezeichnet.

2.5 Methoden mit Parametern

Bisher haben wir nur Methoden denen ein leeres Klammernpaar folgte, geschrieben. Solche Methoden nennt man parameterlose Methoden. Wir können aber auch einer Methode für ihre Ausführung Daten übergeben, mit denen die Methode arbeiten soll. Hierzu muß in der Definition einer Methode deklariert sein, was für Typen von Daten die Methode erwartet. Hierzu schreibt man in der Methodendefinition in die runden Klammern für die Parameter den Namen des Typs der erwarteten Daten. Diesem Typnamen folgt ein frei wählbarer Name für den Parameter, damit wir uns mit Hilfe dieses Parameternamens in den Befehlen der Methode auf den Parameter beziehen können. Beim Aufruf der Methode sind für die Parameter entsprechende Daten zu übergeben.

Beispiel:
Wir schreiben die erste eigene Methode mit einem Parameter. Die Methode `druckeDoppelt` erwartet in ihrem Aufruf eine Zeichenkette. Die Methode gibt die übergebene Zeichenkette dann zweimal auf dem Bildschirm aus.

Doppeldruck.java

```
1  class Doppeldruck {
2
3    public static void main(){
4      druckeDoppelt("hallo");
5      druckeDoppelt("Ilja");
6    }
7
8    public static void druckeDoppelt(String text){
9      System.out.println(text);
10     System.out.println(text);
11   }
12
13 }
```

Aufgabe 3 Testen Sie, was passiert, wenn Sie im obigen Programm die Methode `druckeDoppelt` ohne konkreten Wert für den Parameter aufrufen.

2.5.1 rekursive Methoden

Wir haben gelernt, daß Methopden überall aufgerufen werden kann. Eine Methode kann sogar in ihrem eigenen Rumpf aufgerufen werden.

Beispiel:
In der folgenden Klasse ruft die Methode `ganzOft` sich selbst wieder auf. Der Effekt ist, daß dadurch endlos `hallo` auf dem Bildschirm ausgegeben wird.

[3] und fast allen anderen Programmiersprachen

GanzOft.java

```
1  class GanzOft{
2
3    public static void main(String [] args){
4      ganzOft("hallo");
5    }
6
7    public static void ganzOft(String x){
8      System.out.println(x);
9      ganzOft(x);
10   }
11 }
```

Methoden, die sich selbst wieder aufrufen, nennt man rekursiv.

2.6 Methoden mit Rückgabewert

Unsere Methoden haben bisher nur einen Befehl benutzt, nämlich etwas auf dem Bildschrim auszugeben. In einem Programm werden in der Regel Ergebnisse berechnet. Hierzu kann man Unterprogramme schreiben, deren Ausführung ein Ergebnis errechnen. Damit bekommt eine Methode nicht nur über die Parameter Daten als Eingabe, sondern berechnet ein Ergebnis und gibt dieses Ergebnis an die aufrufende Stelle zurück. Hierzu ist statt des Schlüsselwortes `void` in der Methodendefinition der Typname für die Art der Daten, die als Ergebnis berechnet werden, anzugeben. In einer Methode, die eine ganze Zahl als Ergebnis hat, ist statt `void` also der Typname `int` zu schreiben.

Zusätzlich ist in einer Methode, die ein Ergebnis berechnet auch noch ein `return`-Befehl zu schreiben. Der `return`-Befehl sorggt dafür, daß die Berechnung beendet wird und der Wert, der nach dem Wort `return` folgt als Ergebnis zurückzugeben ist.

Eine Methode mit Parameter und einem Rückgabewert entsprich einer mathematischen Funktion. Die Mathematikunterricht oft benutzte Funktion: $f(x) = x^2$ ist Javatechnisch eine Methode mit einem Zahlenparameter und einer Zahl als Ergebnis.

Beispiel:
In der folgenden Klasse schreiben wir eine Methode, die für ihren Parameter die Quadratzahl als Ergebnis liefert.

Quadrat.java

```
1  class Quadrat {
2
3    public static void main(){
4      System.out.println(quadrat(2));
5      System.out.println(quadrat(3));
6      System.out.println(quadrat(4));
7      System.out.println(quadrat(5));
8    }
```

```
9
10    public static int quadrat(int x){
11      return x*x;
12    }
13
14  }
```

Aufgabe 4 Schreiben Sie eine zusätzliche Methode in obiger Klasse. Die Methode soll zwei Zahlen `x` und `y` als Parameter haben und als Ergebnis $2*x+4*y$ berechnen.

2.7 Wahrheitswerte

Ein in der Informatik häufig gebrauchte Unterscheidung ist, ob etwas wahr oder falsch ist, also eine Unterscheidung zwischen genau zwei Werten.[4] Hierzu bedient man sich der Wahrheitswerte aus der formalen Logik. Java bietet einen primitiven Typ an, der genau zwei Werte annehmen kann, den Typ: `boolean`. Die Bezeichnung ist nach dem englischen Mathematiker und Logiker George Bool (1815–1869) gewählt worden.

Entsprechend der zwei möglichen Werte für diesen Typ stellt Java auch zwei Literale zur Verfügung: `true` und `false`.

Bool'sche Daten lassen sich ebenso wie numerische Daten benutzen. Methoden können diesen Typ verwenden.

Testbool.java

```
1   class Testbool{
2     public static void main(String [] args){
3       System.out.println(true);
4     }
5
6     public static boolean traceBoolean(boolean b){
7       System.out.println(b);
8       return b;
9     }
10  }
```

2.8 Operatoren

Wir haben jetzt gesehen, was Java uns für Typen zur Darstellung von Zahlen zur Verfügung stellt. Jetzt wollen wir mit diesen Zahlen nach Möglichkeit auch noch rechnen können. Wir haben in den vorhergehenden Abschnitten stillschweigend schon ein paar der von Java zur Verfügung gestellten wie `*, -, /` etc. benutzt. Prinzipell gibt es in der Informatik für Operatoren drei mögliche Schreibweisen:

[4] Wahr oder falsch, eine dritte Möglikchkeit gibt es nicht. Logiker postulieren dieses als *tertium non datur*.

- **Präfix:** Der Operator wird vor den Operanden geschrieben, also z.B. `(* 2 21)`. Im ursprünglichen Lisp gab es die Prefixnotation für Operatoren.
- **Postfix:** Der Operator folgt den Operanden, also z.B. `(21 2 *)`. Forth und Postscript sind Beispiele von Sprachen mit Postfixnotation.
- **Infix:** Der Operator steht zwischen den Operanden. Dieses ist die gängige Schreibweise in der Mathematik und für das Auge die gewohnteste. Aus diesem Grunde bedient sich Java der Infixnotation: `42 * 2`.

2.8.1 Die Grundrechenarten

Java stellt für Zahlen die vier Grundrechenarten zur Verfügung.

Bei der Infixnotation gelten für die vier Grundrechenarten die üblichen Regeln der Bindung, nämlich Punktrechnung vor Strichrechnung. Möchte man diese Regel durchbrechen, so sind Unterausdrücke in Klammern zu setzen. Folgende kleine Klasse demonstriert den Unterschied:

PunktVorStrich.java

```
1 class PunktVorStrich{
2   public static void main(String [] args){
3     System.out.println(2 + 20 * 2);
4     System.out.println((2 + 20) * 2);
5   }
6 }
```

Wir können nun also Methoden schreiben, die Rechnungen vornehmen. In der folgenden Klasse definieren wir z.B. eine Methode, zum Berechnen der Quadratzahl der Eingabe:

Square.java

```
1 class Square{
2   static int square(int i){
3     return i*i;
4   }
5 }
```

2.8.2 Vergleichsoperatoren

Obige Operatoren rechnen jeweils auf zwei Zahlen und ergeben wieder eine Zahl als Ergebnis. Vergleichsoperatoren vergleichen zwei Zahlen und geben einen bool'schen Wert, der angibt, ob der Vergleich wahr oder falsch ist. Java stellt die folgenden Vergleichsoperatoren zur Verfügung: `<`, `<=`, `>`, `>=`, `!=`, `==`. Für die Gleichheit ist in Java das doppelte Gleichheitszeichen `==` zu schreiben, denn das einfache Gleichheitszeichen ist bereits für den Zuweisungsbefehl vergeben. Die Ungleichheit wird mit`!=` bezeichnet.

Folgende Tests demonstrieren die Benutzung der Vergleichsoperatoren.

Vergleich.java

```
class Vergleich{

  public static void main(String[] args){
    System.out.println(1+1 < 42);
    System.out.println(1+1 <= 42);
    System.out.println(1+1 >  42);
    System.out.println(1+1 >= 42);
    System.out.println(1+1 == 42);
    System.out.println(1+1 != 42);
  }
}
```

2.8.3 Bool'sche Operatoren

In der bool'schen Logik gibt es eine ganze Reihe von binären Operatoren für logische Ausdrücke. Für zwei davon stellt Java auch Operatoren bereit: `&&` für das logische und ($\wedge$) und`||` für das logische Oder ($\vee$).

Zusätzlich kennt Java noch den unären Operator der logischen Negation $\neg$. Er wird in Java mit `!` bezeichnet.

Wie man im folgenden Test sehen kann, gibt es auch unter den bool'schen Operatoren eine Bindungspräzedenz, ähnlich wie bei der Regel Punktrechnung vor Strichrechnung. Der Operator `&&` bindet stärker als der Operator `||`:

TestboolOperator.java

```
class TestboolOperator{
  public static void main(String [] args){
    System.out.println(true && false);
    System.out.println(true || false);
    System.out.println(!true || false);
    System.out.println(true || true && false);
  }
}
```

In der formalen Logik kennt man noch weitere Operatoren, z.B. die Implikation $\rightarrow$. Diese Operatoren lassen sich aber durch die in Java zur Verfügung stehenden Operatoren ausdrücken. $A \rightarrow B$ entspricht $\neg A \vee B$. Wir können somit eine Methode schreiben, die die logische Implikation testet.

TestboolOperator2.java

```
class TestboolOperator2{
  static boolean implication(boolean a, boolean b){
    return !a || b;
  }

  public static void main(String [] args){
    System.out.println(implication(true, false));
  }
}
```

2.8.4 Der Operator + auf String

Wir haben den Operator `+` kennengelernt als Additionsoperator für Zahlen. Er kann aber auch benutzt werden, um zwei `Strings` aneinanderzuhängen. Dann sind beide Operanden vom Typ `String`. Man kann das Pluszeichen aber auch so verwenden, daß ein Operand ein `String` und der andere eine Zahl. Dann ist Java so konzipiert, daß es die Zahl erst in einen entsprechenden `String` umwandelt.

Beispiel:
Die folgende Klasse benutzt den Operator `+` auf unterschiedlichen Operanden.

TestboolOperator2.java

```
1  class OperatorPlus {
2
3    public static void main(String [] args){
4      System.out.println("hallo"+" welt");
5      System.out.println("1"+"1");
6      System.out.println(1+1);
7      System.out.println("1"+1);
8      System.out.println(1+"1");
9    }
10 }
```

Puristen kritisieren gerne diese mehrfache Bedeutung des Pluszeichens in Java, insbesondere weil `"1"+"1"` das Ergebis `"11"` hat und `1+1` als Ergebnis `2`.

2.9 Felder

Wir kennen jetzt Methoden als die eigentlichen Programmteile. Methoden können Daten in Form von Parametern als Eingabe bekommen und Daten als Ergebnis berechnen. In Programmiersprachen gibt es ein zusätzliches Konzept, um Daten zwischenzuspeichern. In Java heißt dieses Konzept *Feld*, man spricht in anderen Sprachen und auch in Java oft von Variablen. Ein Feld ist ein Platz, in dem Daten abgelegt werden können, um sie bei einem späteren Befehl wieder benutzen zu können. Die Daten, die in einem Feld stehen können verändert werden. Hierzu gibt es den Zuweisungsbefehl. Ein Feld muß bevor es benutzt werden kann, deklariert werden. Damit wird der Javamaschine gesagt, daß ein Speicherplatz benötigt wird, in dem bestimmte Daten abgelegt werden sollen. Eine Felddeklaration besteht aus einem Typnamen gefolgt von einem frei wählbaren Feldnamen. Per Konvention beginnt der Feldname mit einem Kleinbuchstaben. Felder können an beliebiger Stelle deklariert werden, z.B. zwischen zwei Befehlen. In einem Feld werden Daten durch den Zuweisungsbefehl geschrieben, der Zuweisungsbefehl besteht aus einem Gleichheitszeiche. Links von dem Gleichheitszeichen steht ein Feldame, rechts davon stehen die Daten, die in das Feld geschrieben werden sollen.

Beispiel:
In der Hauptmethode der folgenden Klassen werden nacheinander 4 Zahlen in ein Feld geschrieben, die dann jeweils benutzt werden, um den Wert des Feldes auf dem Bildschirm auszugeben.

FirstField.java

```
1  class FirstField {
2    public static void main(){
3      int i;
4      i=1;
5      System.out.println(i);
6      i=2;
7      i=3;
8      System.out.println(i);
9      i=4;
10     System.out.println(i);
11   }
12 }
```

In einem Feld können insbesondere auch Ergebnisse von Methoden abgespeichert werden:

2.10 Zusammengesetzte Befehle

Streng genommen kennen wir bisher nur einen Befehl, den Zuweisungsbefehl, der einem Feld einen neuen Wert zuweist.[5] In diesem Abschnitt lernen wir weitere Befehle kennen. Diese Befehle sind in dem Sinne zusammengesetzt, daß sie andere Befehle als Unterbefehle haben.

2.10.1 Bedingungsabfrage mit: if

Ein häufig benötigtes Konstrukt ist, daß ein Programm abhängig von einer bool'schen Bedingung, sich verschieden verhält. Hierzu stellt Java die **if**-Bedingung zur Verfügung. Dem Schlüsselwort `if` folgt in Klammern eine bool'sche Bedingung, anschließend kommen in geschweiften Klammern die Befehle, die auszuführen sind, wenn die Bedingung wahr ist. Anschließend kann optional das Schlüsselwort `else` folgen mit den Befehlen, die andernfalls auszuführen sind.

FirstIf.java

```
1  class FirstIf {
2
3    static void firstIf(boolean bedingung){
4      if (bedingung) {
5        System.out.println("Bedingung ist wahr");
6      } else {
7        System.out.println("Bedingung ist falsch");
8      }
9    }
10
11   public static void main(String [] args){
```

[5] Konstrukte mit Operatoren nennt man dagegen Ausdrücke.

```
12      firstIf(true || false);
13    }
14
15  }
```

Das `if`-Konstrukt erlaubt es uns also Fallunterscheidungen zu treffen. Wenn in den Alternativen nur ein Befehl steht, so können die geschweiften Klammern auch fortgelassen werden. Unser Beispiel läßt sich also auch schreiben als:

FirstIf2.java

```
1   class FirstIf2 {
2
3     static void firstIf(boolean bedingung){
4       if (bedingung) System.out.println("Bedingung ist wahr");
5       else System.out.println("Bedingung ist falsch");
6     }
7
8     public static void main(String [] args){
9       firstIf(true || false);
10    }
11
12  }
```

Eine Folge von mehreren `if`-Konstrukten lassen sich auch direkt hintereinanderschreiben, so daß eine Kette von `if` und `else`-Klauseln entsteht.

ElseIf.java

```
1   class ElseIf {
2
3     static String lessOrEq(int i,int j){
4       if (i<10)  return "i kleiner zehn";
5       else if (i>10)  return "i größer zehn";
6       else if (j>10)  return "j größer zehn";
7       else if (j<10)  return "j kleiner zehn";
8       else return "j=i=10";
9     }
10
11    public static void main(String [] args){
12      System.out.println(lessOrEq(10,9));
13    }
14
15  }
```

Wenn zuviele `if`-Bedingungen in einem Programm einander folgen und ineinander verschachtelt sind, dann wird das Programm schnell unübersichtlich. Man spricht auch von *Spaghetti-code*. In der Regel empfiehlt es sich, in solchen Fällen noch einmal über das Design nachzudenken, ob die abgefragten Bedingungen sich nicht durch verschiedene Klassen mit eigenen Methoden darstellen lassen.

Mit der Möglichkeit in dem Programm abhängig von einer Bedingung unterschiedlich weiterzurechnen, haben wir theoretisch die Möglichkeit, alle durch ein Computerprogramm berechenbaren mathematischen Funktionen zu programmieren. So können wir z.B. eine Methode schreiben, die für eine Zahl `i` die Summe von `1` bis `n` berechnet.

Summe.java

```
1 class Summe{
2   static int summe(int i){
3     if (i==1)  {
4       return 1;
5     }else {
6       return summe(i-1) + i;
7     }
8   }
9 }
```

Wir können diese Programm von Hand ausführen, indem wir den Methodenaufruf für `summe` für einen konkreten Parameter `i` durch die für diesen Wert zutreffende Alternative der Bedingungsabfrage ersetzen. Wir kennzeichnen einen solchen Ersetzungsschritt durch einen Pfeil →.

```
summe(4)
→summe(4-1)+4
→summe(3)+4
→summe(3-1)+3+4
→summe(2)+3+4
→summe(2-1)+2+3+4
→summe(1)+2+3+4
→1+2+3+4
→3+3+4
→6+4
→10
```

Wie man sieht wird für `i=4` die Methode `summe` genau viermal wieder aufgerufen, bis schließlich die Alternative mit dem konstanten Rückgabewert `1` zutrifft. Unser Trick war, im Methodenrumpf die Methode, die wir gerade definieren, bereits zu benutzen. Diesen Trick nennt man in der Informatik *Rekursion*. Mit diesem Trick ist es uns möglich, ein Programm zu schreiben, bei dessen Ausführung ein bestimmter Teil des Programms mehrfach durchlaufen wird.

Das wiederholte Durchlaufen von einem Programmteil ist das A und O der Programmierung. Daher stellen Programmiersprachen in der Regel Konstrukte zur Verfügung, mit denen man dieses direkt ausdrücken kann. Die Rekursion ist lediglich ein feiner Trick, dieses zu bewerkstelligen. In den folgenden Abschnitten lernen wir die zusammengesetzten Befehle von Java kennen, die es erlauben auszudrücken, daß ein Programmteil mehrfach zu durchlaufen ist.

Aufgabe 5 Schreiben Sie eine Methode, die für eine ganze Zahl die Fakultät dieser Zahl berechnet. Testen Sie die Methode zunächst mit kleinen Zahlen, anschließend mit großen Zahlen. Was stellen Sie fest.

2.10.2 Iteration

Die im letzten Abschnitt kennengelernte Programmierung der Programmwiederholung der Rekursion kommt ohne zusätzliche zusammengesetzte Befehle von Java aus. Da Rekursionen von der virtuellen Maschine Javas nur bis zu einen gewissen Maße unterstützt werden, bietet Java spezielle Befehle an, die es erlauben, einen Programmteil kontrolliert mehrfach zu durchlaufen. Die entsprechenden zusammengesetzten Befehle heißen Iterationsbefehle. Java kennt drei unterschiedliche Iterationsbefehle.

Schleifen mit: while

Ziel der Iterationsbefehle ist es, einen bestimmten Programmteil mehrfach zu durchlaufen. Hierzu ist es notwendig, eine Bedingung anzugeben, für wie lange eine Schleife zu durchlaufen ist. `while`-Schleifen in Java haben somit genau zwei Teile:

- die Bedingung
- und den Schleifenrumpf.

Java unterscheidet zwei Arten von `while`-Schleifen: Schleifen für die vor dem Durchlaufen der Befehle des Rumpfes die Bedingung geprüft wird, und Schleifen, für die nach Durchlaufen es Rumpfes die Bedingung geprüft wird.

Vorgeprüfte Schleifen Die vorgeprüfte Schleifen haben folgendes Schema in Java:

`while (`*pred*`){`*body*`}`

pred ist hierbei ein Ausdruck, der zu einem bool'schen Wert auswertet. *body* ist eine Folge von Befehlen. Java arbeitet die vorgeprüfte Schleife ab, indem erst die Bedingung *pred* ausgewertet wird. Ist das ergebnis `true` dann wird der Rumpf *(body)* der Schleife durchlaufen. Anschließend wird wieder die Bedingung geprüft. Dieses wiederholt sich so lange, bis die Bedingung zu `false` auswertet.

Ein simples Beispiel einer vorgeprüften Schleife ist folgendes Programm, das die Zahlen von 0 bis 9 auf dem Bildschirm ausgibt:

WhileTest.java

```
1 class WhileTest {
2   public static void main(String [] args){
3     int i = 0;
4     while (i < 10){
5       i = i+1;
6       System.out.println(i);
7     }
8   }
9 }
```

Mit diesen Mitteln können wir jetzt versuchen, die im letzten Abschnitt rekursiv geschriebene Methode `summe` iterativ zu schreiben:

Summe2.java

```
class Summe2 {
  public static int summe(int n){

    int erg = 0 ;              // Feld für Ergebnis
    int j   = n ;              // Feld zur Schleifenkontrolle

    while (j>0){               // j läuft von n bis 1
      erg = erg + j;           // akkumuliere das Ergebnis
      j = j-1;                 // verringere Laufzähler
    }

    return erg;
  }
}
```

Wie man an beiden Beispielen oben sieht, gibt es oft ein Feld, das zur Steuerung der Schleife benutzt wird. Dieses Feld verändert innerhalb des Schleifenrumpfes seinen Wert. Abhängig von diesem Wert wird die Schleifenbedingung beim nächsten Bedingungstest wieder wahr oder falsch.

Schleifen haben die unangenehme Eigenschaft, daß sie eventuell nie verlassen werden. Eine solche Schleife läßt sich minimal wie folgt schreiben:

Bottom.java

```
class Bottom {
   static public void bottom(){
     while (true){}
   }
}
```

Ein Aufruf der Methode `bottom` startet eine nicht endende Berechnung.

Häufige Programmierfehler sind inkorrekte Schleifenbedingungen, oder falsch kontrollierte Schleifenvariablen. Das Programm terminiert dann mitunter nicht. Solche Fehler sind in komplexen Programmen oft schwer zu finden.

Nachgeprüfte Schleifen In der zweiten Variante der `while`-Schleife steht die Schleifenbedingung syntaktisch nach dem Schleifenrumpf:

`do` {*body*} `while` (*pred*)

Bei der Abarbeitung einer solchen Schleife wird entsprechend der Notation, die Bedingung erst nach der Ausführung des Schleifenrumpfes geprüft. Am Ende wird also geprüft, ob die Schleife ein weiteres Mal zu durchlaufen ist. Das impliziert insbesondere, daß der Rumpf mindestens einmal durchlaufen wird.

Die erste Schleife, die wir für die vorgeprüfte Schleife geschrieben haben, hat folgende die nachgeprüfte Variante:

DoTest.java

```
1  class DoTest {
2    public static void main(String [] args){
3      int i = 0;
4      do {
5        System.out.println(i);
6        i = i+1;
7      } while (i < 10);
8    }
9  }
```

Man kann sich leicht davon vergewissern, daß die nachgeprüfte Schleife mindestens einmal durchlaufen[6] wird:

VorUndNach.java

```
1  class VorUndNach {
2
3    public static void main(String [] args){
4
5        while (falsch())
6           {System.out.println("vorgeprüfte Schleife");};
7
8        do {System.out.println("nachgeprüfte Schleife");}
9        while (false);
10   }
11
12   public static boolean falsch(){return false;}
13 }
```

Schleifen mit: for

Das syntaktisch aufwendigste Schleifenkonstrukt in Java ist die *for*-Schleife.

Wer sich die obigen Schleifen anschaut, sieht, daß sie an drei verschiedenen Stellen im Programmtext Code haben, der kontrolliert, wie oft die Schleife zu durchlaufen ist. Oft legen wir ein spezielles Feld an, dessen Wert die Schleife kontrollieren soll. Dann gibt es im Schleifenrumpf einen Zuweisungsbefehl, der den Wert dieses Feldes verändert. Schließlich wird der Wert dieses Feldes in der Schleifenbedingung abgefragt.

Die Idee der *for*-Schleife ist, diesen Code, der kontrolliert, wie oft die Schleife durchlaufen werden soll, im Kopf der Schleife zu bündeln. Solche Daten sind oft Zähler vom Typ `int`, die bis zu einem bestimmten Wert herunter oder hoch gezählt werden. Später werden wir noch die Standardklasse `Iterator` kennenlernen, die benutzt wird, um durch Listenelemente durchzuiterieren.

Eine *for*-Schleife hat im Kopf

- eine Initialisierung der relevanten Schleifensteuerungsvariablen *(init)*,

[6] Der Javaübersetzer macht kleine Prüfungen auf konstanten Werten, ob Schleifen jeweils durchlaufen werden oder nicht terminieren. Deshalb brauchen wir die Hilfsmethode `falsch()`.

- ein Prädikat als Schleifenbedingung *(pred)*
- und einen Befehl, der die Schleifensteuerungsvariable weiterschaltet *(step)*.

`for (`*init, pred, step*`){`*body*`}`

Entsprechend sieht unsere jeweilige erste Schleife, der Ausgabe der Zahlen von 0 bis 9 in der `for`-Schleifenversion wie folgt aus.

ForTest.java

```
1  class ForTest {
2    public static void main(String [] args){
3
4      for (int i=0; i<10; i=i+1){
5        System.out.println(i);
6      }
7
8    }
9  }
```

Die Reihenfolge, in der die verschiedenen Teile der `for`-Schleife durchlaufen wird, wirkt erst etwas verwirrend, ergibt sich aber natürlich aus der Herleitung der `for`-Schleife aus der vorgeprüften `while`-Schleife:

Als erstes wird genau einmal die Initialisierung der Schleifenvariablen ausgeführt. Anschließend wird die Bedingung geprüft. Abhängig davon wird der Schleifenrumpf ausgeführt. Als letztes wird die Weiterschaltung ausgeführt, bevor wieder die Bedingung geprüft wird.

Die nun schon hinlänglich bekannte Methode `summe` stellt sich in der Version mit der `for`-Schleife wie folgt dar.

Summe3.java

```
1  class Summe3 {
2    public static int summe(int n){
3
4      int erg = 0 ;                    // Feld für Ergebnis
5
6      for (int j = n;j>0;j=j-1){       // j läuft von n bis 1
7        erg = erg + j;                 // akkumuliere das Ergebnis
8      }
9
10     return erg;
11   }
12 }
```

Beim Vergleich mit der `while`-Version erkennt man, wie sich die Schleifensteuerung im Kopf der `for`-Schleife nun gebündelt an einer syntaktischen Stelle befindet.

Die drei Teile des Kopfes einer `for`-Schleife können auch leer sein. Dann wird in der Regel an einer anderen Stelle der Schleife entsprechender Code zu finden

sein. So können wir die Summe auch mit Hilfe der `for`-Schleife so schreiben, daß die Schleifeninitialisierung und Weiterschaltung vor der Schleife, bzw. im Rumpf durchgeführt wird:

Summe4.java

```
1  class Summe4 {
2    public static int summe(int n){
3
4      int erg = 0 ;              // Feld für Ergebnis
5      int j   = n ;              // Feld zur Schleifenkontrolle
6
7      for (;j>0;){               // j läuft von n bis 1
8        erg = erg + j;           // akkumuliere das Ergebnis
9        j = j-1;                 // verringere Laufzähler
10     }
11
12     return erg;
13   }
14 }
```

Wie man jetzt sieht, ist die `while`-Schleife nur ein besonderer Fall der `for`-Schleife. Obiges Programm ist ein schlechter Programmierstil. Hier wird ohne Not die Schleifensteuerung mit der eigentlichen Anwendungslogik vermischt.

Kapitel 3

XML

XML ist eine Sprache, die es erlaubt Dokumente mit einer logischen Struktur zu beschreiben. Die Grundidee dahinter ist, die logische Struktur eines Dokuments von seiner Visualisierung zu trennen. Ein Dokument mit einer bestimmten logischen Struktur kann für verschiedene Medien unterschiedlich visualisiert werden, z.B. als HTML-Dokument für die Darstellung in einem Webbrowser, als pdf- oder postscript-Datei für den Druck des Dokuments und das für unterschiedliche Druckformate. Eventuell sollen nicht alle Teile eines Dokuments visualisiert werden. XML ist zunächst eine Sprache, die logisch strukturierte Dokumente zu schreiben, erlaubt.

Dokumente bestehen hierbei aus den eigentlichen Dokumenttext und zusätzlich aus Markierungen dieses Textes. Die Markierungen sind in spitzen Klammern eingeschlossen.

Beispiel:
Der eigentliche Text des Dokuments sei:

```
1  The Beatles White Album
```

Die einzelnen Bestandteile dieses Textes können markiert werden:

```
1  <cd>
2    <artist>The Beatles</artist>
3    <title>White Album</title>
4  </cd>
```

Die XML-Sprache wird durch ein Industriekonsortium definiert, dem W3C (`http://www.tfh-berlin.de/~panitz/http://www.w3c.org`) . Dieses ist ein Zusammenschluß vieler Firmen, die ein gemeinsames Interesse eines allgemeinen Standards für eine Markierungssprache haben. Die eigentlichen Standards des W3C heißen nicht Standard, sondern Empfehlung *(recommendation)*, weil es sich bei dem W3C nicht um eine staatliche oder überstaatliche Standartisiertungsbehörde handelt.

XML enstand Ende der 90er Jahre und ist abgeleitet von einer umfangreicheren Dokumentenbeschreibungssprache: SGML. Der SGML-Standard ist wesentlich komplizierter und krankt daran, daß es extrem schwer ist, Software für die Verarbeitung von SGML-Dokumenten zu entwickeln. Daher fasste SGML nur Fuß in Bereichen, wo gut strukturierte, leicht wartbare Dokumente von fundamentaler Bedeutung waren, so daß die Investition in teure Werkzeuge zur Erzeugung und Pflege von SGML-Dokumenten sich rentierte. Dies waren z.B. Dokumentationen im Luftfahrtbereich.[1]

Die Idee bei der Entwicklung von XML war: eine Sprache mit den Vorteilen von SGML zu Entwickeln, die klein, übersichtlich und leicht zu handhaben ist.

3.1 XML-Format

Die grundlegendste Empfehlung des W3C legt fest, wann ein Dokument ein gültiges XML-Dokument ist, die Syntax eines XML-Dokuments. Die nächsten Abschnitte stellen die wichtigsten Bestandteile eines XML-Dokuments vor.

Jedes Dokument beginnt mit einer Anfangszeile, in dem das Dokument angibt, daß es ein XML-Dokument nach einer bestimmten Version der XML Empfehlung ist:

```
1 <?xml version="1.0"?>
```

Dieses ist die erste Zeile eines XML-Dokuments. Vor dieser Zeile darf kein Leerzeichen stehen. Die derzeitig aktuellste und einzige Version der XML-Empfehlung ist die Version 1.0. Ein Entwurf für die Version 1.1 liegt vor. Nach Aussage eines Mitglieds des W3C ist es sehr unwahrscheinlich, daß es jemals eine Version 2.0 von XML geben wird. Zuviele weitere Techniken und Empfehlungen basieren auf XML, so daß die Definition von dem, was ein XML-Dokument ist kaum mehr in größeren Rahmen zu ändern ist.

3.1.1 Elemente

Der Hauptbestandteil eines XML-Dokuments sind die Elemente. Dieses sind mit der Spitzenklammernotation um Teile des Dokuments gemachte Markierungen. Ein Element hat einen *Tagnamen*, der ein beliebiges Wort ohne Leerzeichen sein kann. Für einen Tagnamen *name* beginnt ein Element mit `<name>` und endet mit `</name>`. Zwischen dieser Start- und Endemarkierung eines Elements kann Text oder auch weitere Elemente stehen.

Es wird für XML-Dokument verlangt, daß es genau ein einziges oberstes Element hat.

> **Beispiel:**
> Somit ist ein einfaches XML-Dokument ein solches Dokument, in dem der gesammte Text mit einem einzigen Element markiert ist:

[1] Man sagt, ein Pilot brauche den Copiloten, damit dieser die Handbücher für das Flugzeug trägt.

```
<?xml version="1.0"?>
<myText>Dieses ist der Text des Dokuments. Er ist
mit genau einem Element markiert.
</myText>
```

Im einführenden Beispiel haben wir schon ein XML-Dokument gesehen, das mehrere Elemente hat. Dort umschließt das Element `<cd>` zwei weitere Elemente, die Elemente `<artist>` und `<title>`. Die Teile, die ein Element umschließt, werden der Inhalt des Elements genannt.

Ein Element kann auch keinen, sprich den leeren Inhalt haben. Dann folgt der öffnenden Markierung direkt die schließende Markierung.

Beispiel:
Folgendes Dokument enthält ein Element ohne Inhalt:

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page></page>
  <page>dritte Seite</page>
</skript>
```

Leere Elemente

Für ein Element mit Tagnamen *name*, das keinen Inhalt hat, gibt es die abkürzenden Schreibweise: `<name/>`

Beispiel:
Das vorherige Dokument läßt sich somit auch wie folgt schreiben:

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page/>
  <page>dritte Seite</page>
</skript>
```

Gemischter Inhalt

Die bisherigen Beispiele haben nur Elemente gehabt, deren Inhalt entweder Elemente oder Text waren, aber nicht beides. Es ist aber auch möglich Elemente mit Text und Elementen als Inhalt zu schreiben. Man spricht dann vom gemischten Inhalt *(mixed content)*.

Beispiel:
Ein Dokument, in dem das oberste Element einen gemischten Inhalt hat:

```
<?xml version="1.0"?>
<myText>Der <landsmann>Italiener</landsmann>
<eigename>Ferdinand Carulli</eigename> war  als Gitarrist
ebenso wie der <landsmann>Spanier</landsmann>
<eigename>Fernando Sor</eigename> in <ort>Paris</ort>
ansäßig.</myText>
```

XML-Dokumente als Bäume

Die wohl wichtigste Beschränkung für XML-Dokumente ist, daß sie eine hierarchische Struktur darstellen müssen. Zwei Elemente dürfen sich nicht überlappen. Ein Element darf erst wieder geschlossen werden, wenn alle nach ihm geöffneten Elemente wieder geschlossen wurden.

Beispiel:
Das folgende ist kein gültiges XML-Dokument. Das Element `<bf>` wird geschlossen bevor das später geöffnete Element `<em>` geschlossen worde.

```
<?xml version="1.0"?>
<illegalDocument>
  <bf>fette Schrift <em>kursiv und fett</bf>
  nur noch kursiv</em>.
</illegalDocument>
```

Das Dokument wäre wie folgt als gültiges XML zu schreiben:

```
<?xml version="1.0"?>
<validDocument>
  <bf>fette Schrift</bf><bf> <em>kursiv und fett</em></bf>
  <em>nur noch kursiv</em>.
</validDocument>
```

Dieses Dokument hat eine hierarchische Struktur.

Die hierarchische Struktur von XML-Dokumenten läßt sich sehr schön veranschaulichen, wenn man die Darstellung von XML-Dokumenten in Microsofts Internet Explorer betrachtet.

3.1.2 Attribute

Die Elemente eines XML-Dokuments können als zusätzliche Information auch noch Attribute haben. Attribute haben einen Namen und einen Wert. Syntaktisch ist ein Attribut dargestellt durch den Attributnamen gefolgt von einem Gleichheitszeichen gefolgt von dem in Anführungszeichen eingeschlossenen Attributwert. Attribute stehen im Starttag eines Elements.

Attribute werden nicht als Bestandteils des eigentlichen Textes eines Dokuments betrachtet.

Beispiel:
Dokument mit einem Attribut für ein Element.

```
<?xml version="1.0"?>
<text>Mehr Information zu XML findet man auf den Seiten
des <link address="www.w3c.org">W3C</link>.</text>
```

3.1.3 Kommentare

XML stellt auch eine Möglichkeit zur Verfügung, bestimmte Texte als Kommentar einem Dokument zuzufügen. Diese Kommentare werden mit `<!--` begonnen und mit `-->` beendet. Kommentartexte sind nicht Bestandteil des eigentlichen Dokumenttextes.

Beispiel:
Im folgenden Dokument ist ein Kommentar eingefügt:

```
<?xml version="1.0"?>
<drehbuch filmtitel="Ben Hur">
<akt>
  <szene>Ben Hur am Vorabend des Wagenrennens.
    <!--Diese Szene muß noch ausgearbeitet werden.-->
  </szene>
</akt>
</drehbuch>
```

3.1.4 Character Entities

Sobald in einem XML-Dokument eine der spitze Klammern `<` oder `>` auftaucht, wird dieses als Teil eines Elementtags interpretiert. Sollen diese Zeichen hingegen als Text und nicht als Teil der Markierung benutzt werden, sind also Bestandteil des Dokumenttextes, so muß man einen Fluchtmechanismus für diese Zeichen benutzen. Diese Fluchtmechanismen nennt man *character entities*. Eine Character Entity beginnt in XML mit dem Zeichen `&` und endet mit einem Semikolon`;`. Dazwischen steht der Name des Buchstabens. XML kennt die folgenden Character Entities:

Entity	Zeichen	Beschreibung
`&lt;`	`<`	(less than)
`&gt;`	`>`	(greater than)
`&amp;`	`&`	(ampersant)
`&quot;`	`"`	(quotation mark)
`&apos;`	`'`	(apostroph)

Somit lassen sich in XML auch Dokumente schreiben, die diese Zeichen als Text beinhalten.

Beispiel:
Folgendes Dokument benutzt Character Entities um mathematische Formeln zu schreiben:

```
<?xml version="1.0"?>
<gleichungen>
  <gleichung>x+1&gt;x</gleichung>
  <gleichung>x*x&lt;x*x*x für x&gt;1</gleichung>
</gleichungen>

```

3.1.5 CDATA-Sections

Manchmal gibt es große Textabschnitte in denen Zeichen vorkommen, die eigentlich durch character entities zu umschreiben wären, weil sie in XML eine reservierte Bedeutung haben. XML bietet die Möglichkeit solche kompletten Abschnitte als eine sogenannte *CData Section* zu schreiben. Eine *CData section* beginnt mit der Zeichenfolge `<![CDATA[` und endet mit der Zeichenfolge: `]]>`. Dazwischen können beliebige Zeichenstehen, die eins zu eins als Text des Dokumentes interpretiert werden.

Beispiel:
Die im vorherigen Beispiel mit Character Entities beschriebenen Formeln lassen sich innerhalb einer CDATA-Section wie folgt schreiben.

```
<?xml version="1.0"?>
<formeln><![CDATA[
  x+1>x
  x*x<x*x*x für x > 1
]]></formeln>

```

3.1.6 Processing Instructions

In einem XML-Dokument können Anweisung stehen, die angeben, was mit einem Dokument von einem externen Programm zu tun ist. Solche Anweisungen können z.B. angeben, mit welchen Mitteln das Dokument visualisiert werden soll. Wir werden hierzu im nächsten Kapitel ein Beispiel sehen. Syntaktisch beginnt eine *processing instruction* mit `<?` und endet mit `?>`. Dazwischen stehen wie in der Attributschreibweise Werte für den Typ der Anweisung und eine Referenz auf eine externe Quelle.

Beispiel:
Ausschnitt aus dem XML-Dokument diesen Skripts, in dem auf ein Stylesheet verwiesen wird, daß das Skript in eine HTML-Darstellung umwandelt:

```
<?xml version="1.0"?>
<?xml-stylesheet
   type="text/xsl"
   href="../transformskript.xsl"?>

<skript>
<titelseite>
<titel>Grundlagen der Datenverarbeitung<white/>II</titel>
<semester>WS 02/03</semester>
</titelseite>
</skript>
```

3.1.7 Namensräume

Die *Tagnamen* sind zunächst einmal Schall und Rauch. Erst eine externes Programm wird diesen Namen eine gewisse Bedeutung zukommen lassen, indem es auf die *Tagnamen* in einer bestimmten Weise reagiert.

Da jeder Autor eines XML-Dokuments zunächst vollkommen frei in der Wahl seiner *Tagnamen ist*, wird es vorkommen, daß zwei Autoren denselben *Tagnamen* für die Markierung gewählt haben, aber semantisch mit diesem Element etwas anderes ausdrücken wollen. Spätestens dann, wenn verschiedene Dokumente verknüpft werden, wäre es wichtig, daß *Tagnamen* einmalig mit einer Eindeutigen Bedeutung benutzt wurden. Hierzu gibt es in XML das Konzept der Namensräume.

Tagnamen können aus zwei Teilen bestehen, die durch einen Doppelpunkt getrennt werden:

- dem Präfix, der vor dem Doppelpunkt steht.
- dem lokalen Namen, der nach dem Doppelpunkt folgt.

Hiermit allein ist das eigentliche Problem gleicher *Tagnamen* noch nicht gelöst, weil ja zwei Autoren den gleichen Präfix und gleichen lokalen Namen für ihre Elemente gewählt haben können. Der Präfix wird aber an einem weiteren Text gebunden, der eindeutig ist. Dieses ist der eigentliche Namensraum. Damit garantiert ist, daß dieser Namensraum tatsächlich eindeutig ist, wählt man als Autor seine Webadresse, denn diese ist weltweit eindeutig.

Um mit Namensräume zu arbeiten ist also zunächst ein Präfix an eine Webadresse zu binden; dies geschieht durch ein Attribut der Art:
`xmlns:myPrefix="http://www.myAdress.org/myNamespace"`.

Beispiel:
Ein Beispiel für ein XML-Dokument, daß den Präfix `sep` an einem bestimmten Namensraum gebunden hat:

```
<?xml version="1.0"?>
<sep:skript
```

```
3     xmlns:sep="http://www.tfh-berlin.de/~panitz/dv2">
4    <sep:titel>Grundlagen der DV 2</sep:titel>
5    <sep:autor>Sven Eric Panitz</sep:autor>
6  </sep:skript>
7
```

Die Webadresse eines Namensraumes hat keine eigentliche Bedeutung im Sinne des Internets. Das Dokument geht nicht zu dieser Adresse und holt sich etwa Informationen von dort. Es ist lediglich dazu da, einen eindeutigen Namen zu haben. Streng genommen brauch es diese Adresse noch nicht einmal wirklich zu geben.

3.2 Codierungen

XML ist ein Dokumentenformat, das nicht auf eine Kultur mit einer bestimmten Schrift beschränkt ist, sondern in der Lage ist, alle im Unicode erfassten Zeichen darzustellen, seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines XML-Dokuments kann potentiell eines dieser mehrerern zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein XML-Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen die 8 Bit im Speichern belegen, ein sogenannter Byte (auch Oktett). Ein Byte ist in der Lage 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments darzustellen). Für ein Chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Codierung* eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem ß zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit 2 Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet.

Man sieht, daß es Codierungen geben muß, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüberhinaus auch eine Reihe Mischformen, manche Zeichen werden durch ein Byte andere durch 2 oder sogar durch 3 Byte dargestellt.

Im Kopf eines XML-Dokuments kann angegeben werden, in welcher Codierung das Dokument abgespeichert ist.

Beispiel:
Dieses Skript ist in einer Codierung gespeichert, die für westeuropäische Dokumente gut geeignet ist, da es für die verschiedenen Sonderzeichen der westeuropäischen Schriften einen Zahlenwert im 8-Bit-Bereich zugeordnet hat. Die Codierung mit dem Namen: `iso-8859-1`. Diese wird im Kopf des Dokuments angegeben:

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <skript><kapitel>blablabla</kapitel></skript>
```

Wird keine Codierung im Kopf eines Dokuments angegeben, so wird als Standardcodierung die sogenannte `utf-8` Codierung benutzt. In ihr belegen lateinische Zeichen einen Byte und Zeichen anderer Schriften (oder auch das Euro Symbol) zwei bis drei Bytes.

Eine Codierung, in der alle Zeichen mindestens mit zwei Bytes dargestellt werden ist: `utf-16`, die Standardabbildung von Zeichen, wie sie im *Unicode* definiert ist.

3.3 HTML

Im ersten Semester dieser Vorlesung wurde bereits HTML vorgestellt. HTML sieht äußerlich zunächst XML sehr ähnlich. Es gibt ebenso die Elemente in Spitzen Klammern und Elemente können auch Attribute haben. In HTML gibt es eine feste vordefinierte Anzahl von Elementen, die vom Webbrowser als Anweisung, wie das Dokument darzustellen ist, interpretiert wird. HTML ist älter als XML und ebenso wie XML aus SGML abgeleitet. Während ein XML-Dokument die logische Struktur beschreibt, beschreibt ein HTML-Dokument die Visualisierung eines Dokuments.

Leider sind in HTML einige Dinge erlaubt, die HTML-Dokumente zu ungültigen XML-Dokumenten machen. Dieses sind vor allen:

- Ein öffnendes Tag braucht kein schließendes Tag.
- Html-Dokumente müssen nicht streng hierarchisch sein.
- Attribute können einen Namen ohne Wert haben.
- Attributwerte brauchen nicht unbedingt in Anführungszeichen eingeschlossen sein.

Ein neuerer HTML-Dialekt behebt die Unstimmigkeiten, die verhindern, daß HTML-Dokumente gültige XML-Dokumente sind: XHTML. Es gibt Werkzeuge, die HTML-Dokumente in XHTML-Dokumente und damit auch in XML-Dokumente konvertieren.

In der folgenden Tabelle sind die gebräuchlichsten Tagnamen für HTML-Elemente aufgeführt.

Elementname	**Eigenschaft**

`html`	oberstes Element
`title`	Elemente für Titelzeile
`body`	Element für Dokumentinhalt
`h1`	Hauptüberschrift
`h2`	Unterüberschrift
`h3`	zweite Unterüberschrift
`h4`	weitere Unterüberschrift
`p`	Absatz
`br`	Zeilenumbruch
`center`	zentrierter Text
`b`	**Fettdruck**
`it`	*Kursivdruck*
`em`	*Hervorhebung*
`ul`	Punktliste
`li`	Listenelement
`table`	Tabelle
`tr`	Tabellenzeile
`td`	Tebellenzelle
`<a href=" http://myAdresse/" >`	Link zu einer Adresse
`<img src=" myPicture.gif" >`	Einbindung eines Bildes

Am einfachsten ist es HTML mit einem WYSIWYG-Editor[2] zur schreiben. Hierzu gibt es eine Vielzahl von Programmen; auch der Webbrowser *Netscape* ist in der Lage HTML-Dokumente zu edieren.

3.3.1 Frames

Eine häufig auf Webseiten angewendete Technik ist, mehrere ganze HTML-Seiten nebeneinander in einem Browserfenster anzuzeigen. Hierzu gibt es ein besonderes Konstrukt in HTML, das über zwei Elemente ausgedrückt wird: `<FRAMESET>` und `<Frame>`. In einem Frameset wird definiert wieviele HTML-Seiten auf welche Weise nebeneinander gleichzeitig darzustellenen sind. Die Frameelemente verweisen dann auf die entsprechenden Webseiten.

Beispiel:
Folgender HTML-Code definiert eine Frameseite mit zwei Frames, einen oberen, der 30 Prozent der Gesamtseite ausmacht und einen unteren, der die restlichen 70 Prozent ausmacht. Zwischen diesen beiden Seiten gibt es keinen Trennrand. Dieses wird durch das Attribut `border="0"` ausgedrückt.

[2]WYSIWYG steht für *what you see is what you get* und drückt aus, daß bereits im Editor beim Edieren das fertige Dokument in seiner eigentlichen Darstellungsform zu sehen ist.

```
<html>
<head>
  <title>Testseite für Frames</title>
</head>

<frameset rows="30%, 70%" border="0">
<frame src="obereSeite.html" name="oben"></frame>
<frame src="untereSeite.html" name="unten"></frame>
</frameset>
</html>
```

Wenn es die Dateien `obereSeite.html` und `untereSeite.html` im Dateisystem gibt, so zeigt die Frameseite diese beiden beim Öffnen im Browser an.

Zusätzlich ist den beiden Frames noch das Attribut `name` vergeben worden. Hier ist diesen ein eindeutiger frei gewählter Name zugeordnet worden. Dieses ermöglicht Hyperlinks so zu definieren, das die Seite des Links in einem bestimmten Frame geöffnet wird und nicht als komplett neue Seite, wie es standardmäßig der Fall ist. Hierzu ist mit einem Attribut `target` im Element `a` des Hyperlinks der Name des Frames anzugeben, in dem die Seite des Links geöffnet werden soll.

```
<html><head><title>obere Seite</title>
<body>hier kommt
ein <a href="http://www.tfh-berlin.de" target="unten">Link</a>,
der im unteren Frame geöffnet wird.</body>
</head></html>

```

Als typische Anwendung von Frames wird ein Frame als Navigationsseite benutzt, auf der die Links zu allen Seiten einer Website gesammelt sind und der andere Frame wird benutzt, um die Links deer Navigationsseite anzuzeigen, sofern sie gedrückt werden. Die Navigationsseite wird dabei in der Regel entweder am linken Rand oder über der eigentlichen Inhaltsseite angezeigt.

3.3.2 Sonderzeichen

Auch in HTML-Elemente ist es möglich internationale Zeichen zu verwenden. Leider benutzt HTML ursprünglich nicht denselben Mechanismus wie XML, indem auf gleiche Weise eine Codierung der Zeichen angegeben wird.

Character Entities

HTML kennt im Gegensatz zu den wenigen Character Entities in XML eine große Anzahl solcher Fluchtmechanismen für die unterschiedlichsten Zeichen. So können in einem deutschen Text Umlaute wie mit den folgenden Entities umschrieben werden:

`&uuml;`	u Umlaut	ü
`&Uuml;`	U Umlaut	Ü
`&auml;`	a Umlaut	ä
`&Auml;`	A Umlaut	Ä
`&ouml;`	o Umlaut	ö
`&Ouml;`	O Umlaut	Ö
`&szlig;`	sz-Ligatur	ß

Diese Umschreibung ist bei Tippen eines Textes von Hand sehr umständlich.

Codierung festlegen

In HTML läßt sich die Codierung eines Dokuments durch ein Element `meta` angeben:

```
<html>
<head>
  <meta http-equiv="content-type"
       content="text/html; charset=ISO-8859-1">
</head>
<body>
Deutscher Text mit Umlauten ü Ü ö Ö ä Ä ß
</body>
</html>
```

Damit können Umlaute direkt im Text eingetippt werden und brauchen nicht durch character entities umschrieben zu werden.

3.3.3 CSS

In HTML kann das Layout der Seite relativ genau beschrieben werden. Es kann die Hintergrundfarbe definiert werden, es kann der Schrifttyp festgelegt werden oder auch die Schriftgröße kann festgelegt werden. Wenn man eine komplette Website schreibt, die eventuell viele hundert HTML-Seiten umfasst, z.B. der Webauftritt einer großen Versandhausfirma, so ist es ratsam, bestimmte Stilvorgaben, über Farben und Schriften nicht in jeder Seite einzeln zu definieren, sondern getrennt in einer Stildatei, auf die alle HTML-Seiten verweisen. Soll der gesammte Webauftritt optisch verändert werden, z.B. ein neues Farbschema oder andere Schriften benutzt werden, so brauch dann nur die eine Datei, in der der Stil definiert ist, abgeändert werden. Die eigentlichen HTML-Seiten bleiben davon dann unberührt. Für HTML gibt es eine entsprechendes Stilformat: CSS (*cascading style sheet*).

Eine CSS-Datei definiert den Stil der HTML-Seiten. in den eigentlichen HTML-Seiten wird dieser Stil schließlich nur noch importiert.

Wir geben ein kleines Beispiel für den Einsatz eines CSS. Der interessierte Student sei auf eines der zahlreichen Tutorials im Netz oder die recommendation des W3C verwiesen.

Beispiel:
Folgende kleines CSS-Datei definiert für die Dokumente einen roten Hintergrund und schwarze Schrift. Text in Elementen `div` wird im Blocksatz gesetzt, Fraben für Links gesetzt. Schließlich wird noch ein neues Stilelement Kasten definiert. Diese Datei wird unter den Namen `myStyle.css` abgespeichert.

```
body { background:#ff0000
     ; color:#000000}


.kasten { background:#999999
        ; color: black
        ; border-width: thick
        ; border-color:#000000
        ; border-style:double }

a:link    { color: blue }
a:visited { color: #ffffff }
a:hover   { color: green }
a:active  { color: lime }

div { text-align:justify }
```

Diese Stilbeschreibung kann nun in HTML-Dokumenten benutzt werden, indem auf die CSS-Datei verwiesen wird.

```
<html>
<head>
  <title>CSS Test</title>
   <link href="myStyle.css" rel="stylesheet" type="text/css" />
    <meta http-equiv="content-type"
          content="text/html; charset=ISO-8859-1" />
 </head>
<body>
<div>Diese Seite testet den Einsatz von CCS. Sie verweist in
einem Element <em class="kasten">link</em> auf
die CSS-Datei, die den Stil  bestimmen soll. Wer mehr über
CSS erfahren will sei auf das
<a href="http://www.w3.org">W3C</a> verwiesen.
</div>
</body>
</html>
```

Die Anzeige dieser Seite im Netscape ist in Abbildung 3.3.3 zu bewundern.

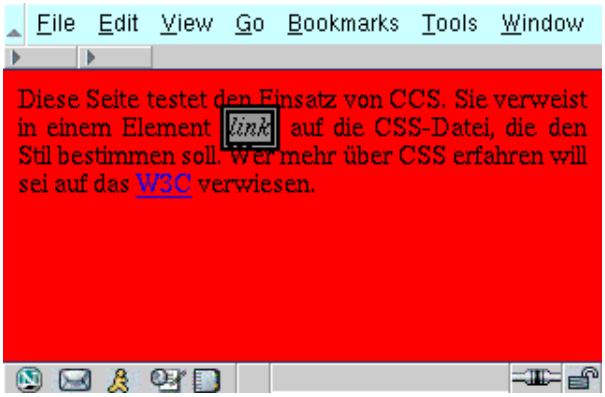

Abbildung 3.1: Anzeige der HTML, deren Stil per CSS-Skript formuliert wurde.

3.4 SVG

Wir haben bisher XML-Dokumente geschrieben, um einen Dokumenttext zu strukturieren. Die Strukturierungsmöglichkeiten von XML machen es zu einem geeigneten Format, um beliebige Strukturen von Daten zu beschreiben. Eine gängige Form von Daten sind Graphiken. SVG *(scalable vector graphics)* sind XML-Dokumente, die graphische Elemente beschreiben. Zusätzlich kann in SVG ausgedrückt werden, ob und in welcher Weise Graphiken animiert sind.

SVG-Dokumente sind XML-Dokumente mit bestimmten festgelegten Tagnamen. Auch SVG ist dabei ein Standard, der vom W3C definiert wird.

Beispiel:
Folgendes kleine SVG-Dokument definiert eine Graphik, die einen Kreis, ein Viereck und einen Text enthält.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <svg xmlns="http://www.w3.org/2000/svg" width="300" height="200">
3  <ellipse cx="100" cy="100" rx="48" ry="90" fill="limegreen" />
4  <text x="20" y="115">SVG Textobjekt</text>
5  <rect x="50" y="50" width="50" height="60" fill="red"/>
6 </svg>
```

Für ausführliche Informationen über die in SVG zur Verfügung stehenden Elemente sei der interessierte Student auf eines der vielen Tutorials im Netz oder direkt auf die Seiten des W3C verwiesen.

3.5 XSLT

XML-Dokumente enthalten keinerlei Information darüber, wie sie visualisiert werden sollen. Hierzu kann man getrennt von seinem XML-Dokument ein sogenanntes *Stylesheet* schreiben. XSL ist eine Sprache zum Schreiben von Stylesheets für XML-Dokumente. XSL ist in gewisser Weise eine Programmiersprache, deren Programme eine ganz bestimmte Aufgabe haben: XML Dokumente

in andere XML-Dokumente zu transformieren. Die häufigste Anwendung von XSL dürfte sein, XML-Dokumente in HTML-Dokumente umzuwandeln.

Beispiel:
Wir werden die wichtigsten XSLT-Konstrukte mit folgendem kleinem XML Dokument ausprobieren:

```
1  <?xml version="1.0" encoding="iso-8859-1" ?>
2  <?xml-stylesheet type="text/xsl" href="cdTable.xsl"?>
3  <cds>
4   <cd>
5    <artist>The Beatles</artist>
6    <title>White Album</title>
7    <label>Apple</label>
8   </cd>
9   <cd>
10   <artist>The Beatles</artist>
11   <title>Rubber Soul</title>
12   <label>Parlophone</label>
13  </cd>
14  <cd>
15   <artist>Duran Duran</artist>
16   <title>Rio</title>
17   <label>Tritec</label>
18  </cd>
19  <cd>
20   <artist>Depeche Mode</artist>
21   <title>Construction Time Again</title>
22   <label>Mute</label>
23  </cd>
24  <cd>
25   <artist>Yazoo</artist>
26   <title>Upstairs at Eric's</title>
27   <label>Mute</label>
28  </cd>
29  <cd>
30   <artist>Marc Almond</artist>
31   <title>Absinthe</title>
32   <label>Some Bizarre</label>
33  </cd>
34  <cd>
35   <artist>ABC</artist>
36   <title>Beauty Stab</title>
37   <label>Mercury</label>
38  </cd>
39 </cds>
40
```

3.5.1 Gesamtstruktur

XSLT-Skripte sind syntaktisch auch wieder XML-Dokumente. Ein XSLT-Skript hat feste Tagnamen, die eine Bedeutung für den XSLT-Prozessor haben. Diese Tagnamen haben einen festen definierten Namensraum. Das äußerste Element eines XSLT-Skripts hat den Tagnamen `stylesheet`. Damit hat ein XSLT-Skript einen Rahmen der folgenden Form:

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <xsl:stylesheet
3   version="1.0"
4   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
5 </xsl:stylesheet>
6
```

Einbinden in ein XML-Dokument

Wir können mit einer *Processing-Instruction* am Anfang eines XML-Dokumentes definieren, mit welchem XSLT Stylesheet es zu bearbeiten ist. Hierzu wird als Referenz im Attribut `href` die XSLT-Datei angegeben.

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <?xml-stylesheet type="text/xsl" href="cdTable.xsl"?>
3 <cds>
4  <cd>........
```

3.5.2 Templates (Formulare)

Das wichtigste Element in einem XSLT-Skript ist das Element `xsl:template`. In ihm wird definiert, wie ein bestimmtes Element transformiert werden soll. Es hat schematisch folgende Form:
`<xsl:template match="`*Elementname*`">`
zu erzeugender Code
`</xsl:template >`

Beispiel:
Folgendes XSLT-Skript transformiert die XML-Datei mit den CDs in eine HTML-Tabelle, in der die CDs tabellarisch aufgelistet sind.

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <xsl:stylesheet
3   version="1.0"
4   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
5
6 <!-- Startregel für das ganze Dokument. -->
7 <xsl:template match="/">
8   <html><head><title>CD Tabelle</title></head>
9     <body>
```

```
10        <xsl:apply-templates/>
11      </body>
12    </html>
13  </xsl:template>
14
15
16  <!-- Regel für die CD-Liste. -->
17  <xsl:template match="cds">
18    <table border="1">
19      <tr><td><b>Interpret</b></td><td><b>Titel</b></td></tr>
20      <xsl:apply-templates/>
21    </table>
22  </xsl:template>
23
24  <xsl:template match="cd">
25    <tr><xsl:apply-templates/></tr>
26  </xsl:template>
27
28  <xsl:template match="artist">
29    <td><xsl:apply-templates/></td>
30  </xsl:template>
31
32  <xsl:template match="title">
33    <td><xsl:apply-templates/></td>
34  </xsl:template>
35
36  <!-- Regel für alle übrigen Elemente.
37       Mit diesen soll nichts gemacht werden -->
38  <xsl:template match="*">
39  </xsl:template>
40
41  </xsl:stylesheet>
```

Öffnen wir nun die XML Datei, die unsere CD-Liste enthält im Webbrowser, so wendet er die Regeln des referenzierten XSLT-Skriptes an und zeigt die so generierte Webseite wie in Abbildung 3.5.2 zu sehen an.

Läßt man sich vom Browser hingegen den Quelltest der Seite anzeigen, so wird kein HTML-Code angezeigt sondern der XML-Code.

3.5.3 Auswahl von Teildokumenten

Das Element `xsl:apply-templates` veranlasst den XSLT-Prozessor die Elemente des XML Ausgangsdokuments weiter mit dem XSL Stylesheet zu bearbeiten. Wenn dieses Element kein Attribut hat, wie in unseren bisherigen Beispielen, dann werden alle Kinder berücksichtigt. Mit dem Attribut `select` lassen sich bestimmte Kinder selektieren, die in der Folge nur noch betrachtet werden sollen.

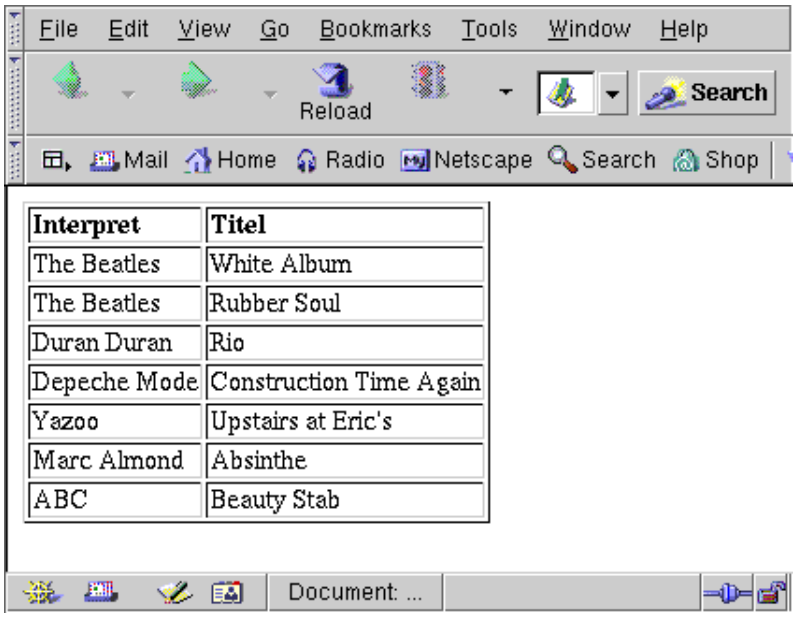

Abbildung 3.2: Anzeige der per XSLT-Skript generierten HTML-Seite.

Beispiel:
Im folgenden XSL Element selektieren wir für `cd`-Elemente nur die `title` und `artist` Kinder und ignorieren die `label` Kinder.

```
1
2  <!-- für das Element "cd" -->
3  <xsl:template match="cd">
4    <!--erzeuge ein Element "tr" -->
5    <tr>
6       <!-- wende das stylesheet weiter an auf die Linderelemente
7       "title" -->
8       <xsl:apply-templates select="title"/>
9       <!-- wende das stylesheet weiter an auf die Linderelemente
10      "artist" -->
11
12      <xsl:apply-templates select="artist"/>
13   </tr>
14 </xsl:template>
```

3.5.4 Sortieren von Dokumentteilen

XSLT kennt ein Konstrukt, um zu beschreiben, daß bestimmte Dokumente sortiert werden sollen nach bestimmten Kriterien. Hierzu gibt es das XSLT-Element `xsl:sort`. In einem Attribut `select` wird

angegeben, nach welchen Elementteil sortiert werden soll.

Beispiel:
Zum Sortieren der CD-Liste kann mit `xsl:sort` das Unterelement `artist` als Sortierschlüssel bestimmt werden.

```
<xsl:template match="cds">
  <table border="1">
    <tr><td><b>Interpret</b></td><td><b>Titel</b></td></tr>
    <xsl:apply-templates select="cd">
      <xsl:sort select="artist"/>
    </xsl:apply-templates>
  </table>
</xsl:template>
```

3.5.5 Weitere Konstrukte

XSLT kennt noch viele weitere Konstrukte, so z.B. für bedingte Ausdrücke wie in herkömmlichen Programmiersprachen durch einen **if**-Befehl und explizite Schleifenkonstrukte, Möglichkeiten das Ausgangsdokument, mehrfach verschiedentlich zu durchlaufen, und bei einem Element nicht nur auf Grund seines Elementnamens zu reagieren, sondern auch seine Kontext im Dokument zu berücksichtigen. Der interessierte Leser sei auf eines der vielen Tutorials im Netz oder auf die Empfehlung des W3C verwiesen.

3.6 DTD und Schema

Wir haben bereits verschiedene Typen von XML-Dokumenten kennengelernt: XHTML-, XSLT- und SVG-Dokumente. Solche Typen von XML-Dokumenten sind dadurch gekennzeichnet, daß sie gültige XML-Dokumente sind, in denen nur bestimmte vordefinierte Tagnamen vorkommen und das auch nur in einer bestimmten Reihenfolge. Solche Typen von XML-Dokumenten können mit einer Typbeschreibungssprache definiert werden. Für XML gibt es zwei solcher Typbeschreibungssprachen: DTD und Schema.

3.6.1 DTD

DTD *(document type description)* ermöglicht es zu formulieren, welche Tags in einem Dokument vorkommen sollen. DTD ist keine eigens für XML erfundene Sprache, sondern aus SGML geerbt.

DTD-Dokumente sind keine XML-Dokumente, sondern haben eine eigene Syntax. Wir stellen im einzelnen diese Syntax vor:

- `<!DOCTYPE` *`root-element`* `[` *`doctype-declaration...`* `]>`
  Legt den Namen des top-level Elements fest und enthält die gesammte Definition des erlaubten Inhalts.

- `<!ELEMENT` *`element-name content-model`*`>`
  assoziiert einen *content model* mit allen Elementen, die diesen Namen haben.

  content models können wie folgt gebildet werden.:

  - `EMPTY`: Das Element hat keinen Inhalt.
  - `ANY`: Das Element hat einen beliebigen Inhalt
  - `#PCDATA`: Zeichenkette, also der eigentliche Text.
  - durch einen regulären Ausdruck, der aus den folgenden Komponenten gebildet werden kann.
    - Auswahl von Alternativen (oder): `(...|...|...)`
    - Sequenz von Ausdrücken: `(...,...,...)`
    - Option: `...?`
    - Ein bis mehrfach Wiederholung: `...*`
    - Null bis mehrfache Wiederholung: `...+`

- `<!ATTLIST`*`element-name attr-name attr-type attr-default`* `...>`
  Liste, die definiert, was für Attribute ein Element hat:

  Attribute werden definiert durch:

  - `CDATA`: beliebiger Text ist erlaubt
  - (*`value`*`|...`): Aufzählung erlaubter Werte

  Attribut *default* sind:

  - `#REQUIRED`: Das Attribut muß immer vorhanden sein.
  - `#IMPLIED`: Das Attribut ist optional
  - `"value"`: Standardwert, wenn das Attribut fehlt.
  - `#FIXED "value"`: Attribut kennt nur diesen Wert.

  Beispiel:
  Ein Beispiel für eine DTD, die ein Format für eine Rezeptsammlung definiert.

```
1  <!ELEMENT collection (description,recipe*)>
2
3  <!ELEMENT description ANY>
4
5  <!ELEMENT recipe (title,ingredient*,preparation
6                   ,comment?,nutrition)>
7
8  <!ELEMENT title (#PCDATA)>
9
10 <!ELEMENT ingredient (ingredient*,preparation)?>
11 <!ATTLIST ingredient name CDATA #REQUIRED
12                      amount CDATA #IMPLIED
13                      unit CDATA #IMPLIED>
```

```
14
15  <!ELEMENT preparation (step*)>
16
17  <!ELEMENT step (#PCDATA)>
18
19  <!ELEMENT comment (#PCDATA)>
20
21  <!ELEMENT nutrition EMPTY>
22  <!ATTLIST nutrition fat CDATA #REQUIRED
23                      calories CDATA #REQUIRED
24                      alcohol CDATA #IMPLIED>
```

Aufgabe 6 Schreiben Sie ein XML Dokument, daß nach den Regeln der obigen DTD gebildet wird.

3.6.2 Schema

Daß DTDs keine XML-Dokumente sind, hat die Erfinder von XML im Nachhinein recht geärgert. Außerdem war für die vorgesehen Zwecke im Bereich Datenverwaltung mit XML, die Ausdrucksstärke von DTD zum Beschreiben der Dokumenttypen nicht mehr ausdruckstark genug. Es läßt sich z.B. nicht formulieren, daß ein Attribut nur Zahlen als Wert haben darf. Aus diesen Gründen wurde eine Arbeitsgruppe ins Leben gerufen, die einen neuen Standard definieren sollte, der DTDs langfristig ersetzen kann. Die neue Sprache sollte in XML Syntax notiert werden und die bei DTDs vermissten Ausdrucksmittel beinhalten. Diese neue Typbneschreibungssprache heißt Schema und ist mitlerweile im W3C verabredet worden. Leider ist das endgültige Dokument über Schema recht komplex und oft schwer verständlich geworden. Wir wollen in dieser Vorlesung nicht näher auf Schema eingehen.

Kapitel 4

Objektorientierte Programmierung

4.1 Objektorientierung

Wir haben Java bisher als eine rein traditionelle prozedurale Programmiersprache betrachtet. Eines der Hauptmerkmale von Java, ist sein objektorientiertes Paradigma.

4.1.1 Modellierung von Objekten

Die Idee der Objektorientierung ist, Daten und Unterprogramme, die inhaltlich eine Einheit bilden in einem Objekt zu bündeln. Hierzu schreibt man Klassen, die Felder für alle Daten enthält, die zu einem Objekt gehören sollen:

Beispiel:
Folgende Klasse modelliert Objekte des Typs Buch. Ein Buch ist dadurch gekennzeichnet, daß es einen Titel, einen Autor und einen Preis hat. Autorname und Titel sind als Feldern des Typs `String` modelliert, der Preis als ein Feld des Typs `int`.

Buch1.java

```
1 class Buch1{
2   String titel;
3   String autor;
4   int    preis;
5 }
```

Obige Klasse beschreibt wie Objekte des Typs `Buch1` aussehen. Sie fassen jeweils drei Felder zusammen.

Aufgabe 7 Schreiben Sie eine Klasse `Person1`, die eine Person modelliert. eine Person soll einen Namen, einen Vornamen, eine Straße, eine Hausnummer, eine Postleitzahl und einen Ort als Daten enthalten.

4.1.2 Konstruktoren

Bisher haben wir beschrieben, wie die Objekte einer Klasse aussehen. Jetzt müssen wir noch beschreiben, wie ein Objekt konstruiert werden kann. Hierzu wird in der Klasse noch eine spezielle Methode geschrieben, der *Konstruktor*. Der Konstruktor gibt an, wie ein Objekt einer Klasse aus einzelenen Daten konstruiert werden kann. Ein Konstruktor hat in Java immer genau den Namen der Klasse.

Ein Konstruktor sieht einer Methode sehr ähnlich. Er hat einen Kopf, in dem steht, was für Argumente übergeben werden und einen Rumpf, der in geschweiften Klammern eingeschlossen ist. In diesem Rumpf befinden sich Javabefehle. Typischer Weise werden im Rumpf einer Methode die dem Konstruktor übergebenen Argumente in die Felder des Objektes abgespeichert.

Beispiel:
Folgende Klasse fügt der Klasse `Buch1` aus dem letzten Abschnitt noch einen Konstruktor hinzu:

Buch2.java

```
1  class Buch2 {
2    //die Felder der Klasse
3    String titel;
4    String autor;
5    int    preis;
6
7    //der Konstruktor. er erhält Daten für die Felder und weist
8    //diese Daten den Feldern zu.
9    Buch2(String derTitel,int derPreis,String derAutor){
10     titel=derTitel;
11     autor=derAutor;
12     preis=derPreis;
13   }
14 }
```

Aufgabe 8 Schreiben Sie eine Klasse `Person2`, die zusätzlich zu den Feldern aus ihrer Klasse `Person1` noch einen Konstruktor enthält, der die Felder der Klasse mit als Argumente übergebenen Daten füllt.

4.1.3 Benutzen von Objekten

Wir sind jetzt soweit, daß wir in einer Klasse beschrieben haben, was für Daten die Objekte dieses Typs enthalten und wie Objekte konstruiert werden können. Klassen die einen Objekttyp beschreiben definieren einen neuen Javatyp. Wir können den Typ `Buch2` jetzt genauso benutzen, wie den Typ `String`. Wir können Felder des Typs `Buch2` definieren, oder Parameter vom Typ `Buch2` für Methoden vorsehen.

Jetzt müssen wir nur noch Objekte erzeugen und und mit ihnen arbeiten.

Erzeugen von Objekten

Wenn wir eine Klasse mit einem Konstruktor geschrieben haben, so kann mit dem Wort `new` der Konstruktor benutzt werden. Dem Wort `new` folgt dabei der Name der Klasse von dem ein neues Objekt erzeugt werden soll, gefolgt schließlich von in runden Klammern eingeschlossenen Argumenten, die dem Konstruktor übergeben werden sollen.

Beispiel:
In folgender Klasse schreiben wir eine Hauptmethode, die in zwei Feldern vom Typ `Buch2` jeweils ein neues Objekt speichert.

ErzeugeBuch2.java

```
1  class ErzeugeBuch2 {
2    public static void main(String [] _){
3      Buch2 b1 = new Buch2("Zettels Traum",150,"Arno Schmidt");
4      Buch2 b2 = new Buch2("Making History",15,"Stephen Fry");
5    }
6  }
```

Die verschiedenen Objekte einer Klasse sind unabhängig voneinander. Im obigen Beispel werden zwei Bücher erzeugt. Diese beiden Bücher haben eigene Titel, Preise und Autornamen. Sie sind wie zwei Objekte des Typs `String` oder zwei Zahlen voneinander unabhängig.

Aufgabe 9 Erzeugen Sie in einer Testklasse zwei Objekte des Typs `Person2`.

Zugriff auf Daten eines Objektes

Wir haben jetzt gelernt, wie Objekte erzeugt werden. Jetzt wollen wir für Objekte wieder auf ihre einzelnen Daten zugreifen können, oder diese eventuell verändern. Hierzu hat Java eine Punktnotation. Wenn man auf einzelne Eigenschaften eines Objektes zugreifen will, so schreibt man in Java einen Punkt gefolgt von der Eigenschaft des Feldes.

Beispiel:
In folgender Klasse erzeugen wir zwei Buchobjekte. Dann werden einzelne Eigenschaften dieser Objekte ausgegeben und schließlich wird für ein Objekt der Preis um 8 Prozent erhöht:

BenutzeBuch2.java

```
1  class BenutzeBuch2 {
2   public static void main(String [] _){
3      Buch2 b1 = new Buch2("Zettels Traum",150,"Arno Schmidt");
4      Buch2 b2 = new Buch2("Making History",15,"Stephen Fry");
5
6      System.out.println("Das Buch "+b1.titel+" von "+b1.autor
7                         + " kostet "+b1.preis+" Euro.");
8      System.out.println("Das Buch "+b2.titel+" von "+b2.autor
9                         + " kostet "+b2.preis+" Euro.");
10
```

```
11      b1.preis = b1.preis+(b1.preis*8/100);
12      System.out.println("Das Buch "+b1.titel+" von "+b1.autor
13                         + " kostet jetzt "+b1.preis+" Euro.");
14      System.out.println("Das Buch "+b2.titel+" von "+b2.autor
15                         + " kostet weiterhin "+b2.preis+" Euro.");
16    }
17  }
```

Wie man sieht, funktionieren Felder eines Objektes genauso, wie wir bisher Felder benutzt haben. Man kann ihre Werte auslesen und verändern. Man muß lediglich dazu notieren zu welchen Objekt das Feld gehört.

Aufgabe 10 Schreiben Sie ein kleines Testprogramm, daß Ihre Klasse `Person2` benutzt. Erzeugen Sie zwei Objekte des Typs `Person2`. Geben Sie einige der Daten dieser Objekte auf dem Bildschirm aus. Ändern Sie dann die Adresse einer der Personen und geben Sie die neuen Daten dieser Person auf dem Bildschirm aus.

4.1.4 Methoden für Objekte

Schreiben von Methoden für Objekte

Der eigentliche Clou der Objektorientierung ist, daß zu Objekten nicht nur Daten gehören, die in Feldern gespeichert sind, sondern auch Methoden, die auf diesen Daten operieren. Hierzu kann man in einer Klasse Methoden schreiben. Dabei läßt man das Attribut `static`, daß wir bisher vor Methoden geschrieben haben fort. Damit bezieht sich eine Methode nur auf ein bestimmtes Objekt. Sie rechnet dann nur noch auf den Daten eines einzelnen Objektes.

Beispiel:
Wir ergänzen unsere Klasse zur Modellierung von Büchern um drei Methoden. Eine Methode berechnet den Preis des Objektes in Dollar. Die andere Methode erhöht den Preis um einen bestimmten Prozentsatz, die letzte Methode gibt alle Daten eines Buches als `String` zurück.

Buch.java

```
1   class Buch {
2     //die Felder der Klasse
3     String titel;
4     String autor;
5     int    preis;
6
7     //der Konstruktor. er erhält Daten für die Felder und weist
8     //diese Daten den Feldern zu.
9     Buch(String derTitel,int derPreis,String derAutor){
10      titel=derTitel;
11      autor=derAutor;
12      preis=derPreis;
13    }
```

```
14
15    //Methoden für die Objekte
16    public int preisInDollar(){
17      return preis+preis*17/100;
18    }
19
20    public void erhöhePreisUmProzent(int prozent){
21      preis=preis+preis*prozent/100;
22    }
23
24    public String toString(){
25      return
26       "Das Buch "+titel+" von "+autor+ " kostet "+preis+" Euro.";
27    }
28  }
```

Wie man sieht können die Methoden, die nicht statisch sind, genauso alle Merkmale der statischen Methoden haben. Sie können Parameter haben (wie die Methode `erhöhePreisUmProzent`), sie können eine Rückgabewert berechnen (wie die Methoden `preisInDollar` und `toString`). Aber sie können auch die Werte eines Objektes verändern, soe wie es die Methode `erhöhePreisUmProzent` macht.

Aufgabe 11 Schreiben Sie eine Klasse `Person`, indem Sie Ihre Klasse `Person2` um zwei Methoden erweitern.

- `public String toString()` soll für eine Person einen `String` erzeugen, der den kompletten Namen und die Adresse einer Person beschreibt.
- `public void ändereAdresse(String neueStr, int neueHausnr)` soll die Adresse so ändern, daß die Person innerhalb einer Stadt in eine andere Straße umgezogen ist.

Benutzung von Methoden auf Objekten

Nachdem wir für Objekte Methoden geschrieben haben, wollen wir diese natürlich auch benutzen. Hierzu wird die gleiche Syntax benutzt, wie für den Zugriff auf die Felder eines Objektes, nämlich die Punktnotation. Der Punkt trennt das Objekt von dem Methodenaufruf für das Objekt.

Beispiel:

BenutzeBuch.java

```
1  class BenutzeBuch{
2    public static void main(String [] _){
3      Buch b1 = new Buch("Zettels Traum",150,"Arno Schmidt");
4      Buch b2 = new Buch("Making History",15,"Stephen Fry");
5
6      System.out.println(b1.toString());
```

```
7       System.out.println(b2.toString());
8
9       b1.erhöhePreisUmProzent(8);
10      System.out.println(b1.toString());
11
12      System.out.println
13       ("Das Buch "+b1.titel+" kostet in Dollar: "
14        +b1.preisInDollar()+"USD");
15    }
16  }
```

Wie man sieht, bezieht sich eine Methode tatsächlich auf genau ein Objekt und operiert nur auf dessen Daten.

Aufgabe 12 Schreiben Sie eine Testklasse für Ihre Klasse `Person`, so daß Sie die Methoden der Klasse `Person` alle einmal aufgerufen haben.

4.1.5 Objekte, die Objekte enthalten

Klassen, die wir für bestimmte Objekttypen schreiben, definieren genauso Typen, wie die Typen, die in Java schon vordefiniert sind. Die Klassen `Buch` und `Person` können jetzt also genauso als Typ benutzt werden, wie z.B. die Klasse `String`.

Beispiel:
Wir definieren eine neue Klasse, die ausdrückt, daß ein Buch von einer bestimmten Person ausgeliehen ist:

Buchausleihe.java

```
1   class Buchausleihe {
2     Buch buch;
3     Person ausleiher;
4     int rückgabeTag;
5     int rückgabeMonat;
6     int rückgabeJahr;
7
8     Buchausleihe(Buch b,Person a,int tag,int monat,int jahr){
9       buch=b;
10      ausleiher=a;
11      rückgabeTag=tag;
12      rückgabeMonat=monat;
13      rückgabeJahr=jahr;
14    }
15
16    public String toString(){
17      return ausleiher+" hat das Buch "+buch
18          +" ausgeliehen. Rückgabe ist der "
19          +rückgabeTag+"."+rückgabeMonat+"."+rückgabeJahr;
20    }
```

```
21
22    void verlängerteAusleihe(){
23      if (rückgabeMonat==12){
24        rückgabeMonat=1;
25        rückgabeJahr=rückgabeJahr+1;
26      }else{
27        rückgabeMonat=rückgabeMonat+1;
28      }
29    }
30  }
```

Aufgabe 13 Schreiben Sie ein kleines Testprogramm, das Objekte des Typs `Buchausleihe` erzeugt und Methoden darauf anwendet.

4.1.6 Die Java Standardklassen String

Wir haben bisher den Typ `String` benutzt, ohne uns darüber Gedanken zu machen, daß es sich dabei auch um eine Klasse handelt, und die einzelnen Daten der Klasse `String` auch Objekte sind, auf die Methoden aufgerufen werden können. `String` ist nichts weiteres als eine Klasse, die nicht wir geschrieben haben, sondern die Entwickler der Sprache Java für uns geschrieben haben. Der einzige Unterschied zu Klassen, die wir schreiben ist, daß Stringobjekte nicht durch das `new`-Konstrukt erzeugt werden, sondern durch eine eigene Notation, in der der Text des `String`-Objekts einfach in Anführungszeichen eingeschlossen wird. Wenn wir in die Javadokumentation schauen, stellen wir fest, daß es eine ganze Reihe von unterschiedlichen Methoden für die Klasse `String` gibt, z.B. zur Berechnung der Länge eines Textes, oder um Teiltexte zu selektieren.

Beispiel:
In der folgenden Klasse testen wir einige der Methoden, die für `String`-Klasse definiert sind:

StringTest.java

```
1   class StringTest {
2
3     public static void main(String [] _){
4       String str1 = "hallo da draußen";
5
6       System.out.println(str1.length());
7       String str2 = str1.substring(2,6);
8       System.out.println(str2);
9       System.out.println(str2.length());
10
11      System.out.println(str1.toUpperCase());
12      System.out.println(str1.toUpperCase().length());
13    }
14  }
```

Aufgabe 14 Suchen Sie sich aus der Javadokumentation zur Klasse `String` weitere Methoden heraus und schreiben Sie ein Testprogramm, daß diese Methoden ausprobiert.

4.1.7 Erweitern von Klassen

Ein weiteres Schlüsselkonzept der objektorientierten Programmierung erlaubt es, Klassen, die Objekte beschreiben, um zusätzliche Information zu erweitern. Diese zusätzlichen Informationen können weitere Datenfelder, oder zusätzliche Methoden sein.

Definition von Unterklassen

Hierzu schreibt man eine neue Klasse, die eine bestehende Klasse erweitert. Man spricht von einer Unterklasse. Die Unterklasse hat alle Eigenschaften der Oberklasse und zusätzlich vielleicht ein paar weitere Eigenschaften. In Java gibt man durch eine `extends`-Klausel nach dem Klassennamen an, daß die neue Klasse eine bestehende Klasse erweitert.

Der erste Befehl im Konstruktor einer Unterklasse muß den Konstruktor der Oberklasse aufrufen. Damit ist gewährleistet, daß bevor man ein Objekt um zusätzliche Eigenschaften erweitert, ersteinmal das zu erweiternde Objekt erzeugt werden muß. In Java wird der Konstruktor der Oberklasse durch den Befehl `super` aufgerufen. Diesem gibt man die Argumente zum Konstruieren der Superklasse mit.

Beispiel:
Wir schreiben eine Klasse `UebersetztesBuch`, die die Klasse `Buch` erweitert, und zwei zusätzliche Eigenschaft hat, nämlich den Namen des Übersetzers und die Sprache des Originals:

UebersetztesBuch.java

```
1  class UebersetztesBuch extends Buch {
2    String übersetzer;
3    String spracheDesOriginals;
4
5    UebersetztesBuch
6        (String derTitel,int derPreis,String derAutor
7        ,String derÜbersetzer,String sprache){
8      super(derTitel,derPreis,derAutor);
9      übersetzer = derÜbersetzer;
10     spracheDesOriginals = sprache;
11   }
12 }
```

Hier ist `UebersetztesBuch` die Unterklasse, die die Oberklasse `Buch` erweitert.

Aufgabe 15 Schreiben Sie eine Klasse `Student`, die die Klasse `Person` um ein weiteres Feld `matrikelNr` erweitert.

Benutzung von Unterklassen

Objekte einer Unterklasse können vollständig wie Objekte der Oberklasse benutzt werden. Überall dort, wo ein Objekt der Oberklasse erwartet wird, kann auch ein Objekt der Unterklasse benutzt werden.

Beispiel:
In der folgenden Klassen wird ein Objekt des Typs `Buch` und ein Objekt des Typs `UebersetzesBuch` erzeugt. Auf beiden werden einige Eigenschaften getestet.

BenutzeUebersetztesBuch.java

```
1  class BenutzeUebersetztesBuch{
2    public static void main(String [] _){
3      Buch b1 = new Buch("Zettels Traum",150,"Arno Schmidt");
4      UebersetztesBuch b2
5       = new UebersetztesBuch
6                ("Eines Tages ich sprechen hübsch"
7                ,19,"Sedaris","Harry Rowolth","Englisch");
8
9      System.out.println(b1.toString());
10     System.out.println(b2.toString());
11
12     b2.erhöhePreisUmProzent(8);
13     System.out.println(b2.toString());
14
15     System.out.println
16     ("Der Übersetzer von: "+b2.titel+" ist: "+b2.übersetzer);
17   }
18 }
```

Aufgabe 16 Schreiben Sie eine Testklasse, in der Objekte des Typs `Student` getestet werden.

Anhang A

Beispielaufgaben

Aufgabe 1 Die folgenden Dokumente sind kein wohlgeformetes XML. Begründen Sie, wo der Fehler liegt, und wie dieser Fehler behoben werden kann.

a)
```
<a>to be <b> or not to be </a>
```

b)
```
<a> x + 1 > x </a>
```

c)
```
<a><dick>The world<kursiv> is</dick>
 my </kursiv> oyster.</a>
```

Aufgabe 2 Was geben die folgenden Javaprogramme auf dem Bildschirm aus, wenn sie ausgeführt werden:

a)
```
class Aufgabe2a{
  static int q(int x){
    int result = x*x;
    result = result*result;
    return result;
  }

  static public void main(String[] args){
    System.out.println(q(2));
    System.out.println(q(3));
  }
}
```

b)
```
class Aufgabe2b {
  static void m1(String s){
    for (int i =1; i<4;i=i+1){
      System.out.println(s);
    }
```

```
6     }
7
8     public static void main(String [] args){
9       m1("hallo");
10      m1("Illja");
11    }
12  }
```

Aufgabe 3 Schreiben Sie das folgende Programm so um, daß es statt einer `for`-Schleife eine `while`-Schleife benutzt.

```
1   class Aufgabe3 {
2
3     static public int hochN(int i, int exp){
4       int result = 1;
5
6       for (int j=0; j < exp;j=j+1){
7         result = result*i;
8       }
9
10      return result;
11    }
12
13  }
```

Aufgabe 4 Ergänzen Sie das untere Programm, indem Sie den fehlenden Programmtext der Methode `m` gemäß der im Kommentar gegebenen Spezifikation einfügen.

```
1   class Aufgabe4 {
2     static int m(int x){
3        //m berechnet das vierfache des Eigabeparameters x.
4     }
5
6   }
```

Aufgabe 5 Gegeben sind das folgende XML-Dokument:

```
1   <?xml version="1.0" encoding="iso-8859-1" ?>
2   <?xml-stylesheet type="text/xsl" href="person.xsl"?>
3   <person>
4     <name>Shakespeare</name>
5     <vorname>William</vorname>
6   </person>
```

und das darin referenzierte XSL Stylesheet `person.xsl`:

```
1  <xsl:stylesheet version="1.0"
2                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3
4    <xsl:template match="/">
5       <html><head><title>Person</title></head>
6         <body><xsl:apply-templates/></body>
7       </html>
8    </xsl:template>
9
10   <xsl:template match="name">
11      <b><xsl:apply-templates/></b><br/>
12   </xsl:template>
13
14   <xsl:template match="vorname">
15      <em><xsl:apply-templates/></em>
16   </xsl:template>
17 </xsl:stylesheet>
```

Wie sieht der durch den XSLT-Prozessor aus diesen zwei Dateien generierte HTML-Code aus.

Aufgabe 6 Sie sollen ein Javaapplet schreiben, das auf einer Webseite eingebunden ist. Immer, wenn jemand die Seite von der Website auf seinem lokalen Rechner betrachtet, soll das Applet auf dem lokalen Rechner eine Datei anlegen, in der Datum und Uhrzeit für die Aktivierung des Applets geschrieben wird.

Warum können Sie ein solches Applet nicht realisieren?

Anhang B

Gesammelte Aufgaben

Aufgabe 1 Schreiben Sie das obige Programm mit einen Texteditor ihrer Wahl. Speichern Sie es als `Hallo.java` ab. Übersetzen Sie es mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `hallo.class`. Führen Sie das Programm mit dem Javainterpreter `java` aus.

Aufgabe 2 Übersetzen Sie das Programm `Minimal` aus dem letzem Abschnitt und versuchen Sie dieses Programm laufen zu lassen. Was für eine Fehlermeldung bekommen Sie?

Aufgabe 3 Testen Sie, was passiert, wenn Sie im obigen Programm die Methode `druckeDoppelt` ohne konkreten Wert für den Parameter aufrufen.

Aufgabe 4 Schreiben Sie eine zusätzliche Methode in obiger Klasse. Die Methode soll zwei Zahlen `x` und `y` als Parameter haben und als Ergebnis $2*x+4*y$ berechnen.

Aufgabe 5 Schreiben Sie eine Methode, die für eine ganze Zahl die Fakultät dieser Zahl berechnet. Testen Sie die Methode zunächst mit kleinen Zahlen, anschließend mit großen Zahlen. Was stellen Sie fest.

Aufgabe 6 Schreiben Sie ein XML Dokument, daß nach den Regeln der obigen DTD gebildet wird.

Aufgabe 7 Schreiben Sie eine Klasse `Person1`, die eine Person modelliert. eine Person soll einen Namen, einen Vornamen, eine Straße, eine Hausnummer, eine Postleitzahl und einen Ort als Daten enthalten.

Aufgabe 8 Schreiben Sie eine Klasse `Person2`, die zusätzlich zu den Feldern aus ihrer Klasse `Person1` noch einen Konstruktor enthält, der die Felder der Klasse mit als Argumente übergebenen Daten füllt.

Aufgabe 9 Erzeugen Sie in einer Testklasse zwei Objekte des Typs `Person2`.

Aufgabe 10 Schreiben Sie ein kleines Testprogramm, daß Ihre Klasse `Person2` benutzt. Erzeugen Sie zwei Objekte des Typs `Person2`. Geben Sie einige der Daten dieser Objekte auf dem Bildschirm aus. Ändern Sie dann die Adresse einer der Personen und geben Sie die neuen Daten dieser Person auf dem Bildschirm aus.

Aufgabe 11 Schreiben Sie eine Klasse `Person`, indem Sie Ihre Klasse `Person2` um zwei Methoden erweitern.

- `public String toString()` soll für eine Person einen `String` erzeugen, der den kompletten Namen und die Adresse einer Person beschreibt.
- `public void ändereAdresse(String neueStr, int neueHausnr)` soll die Adresse so ändern, daß die Person innerhalb einer Stadt in eine andere Straße umgezogen ist.

Aufgabe 12 Schreiben Sie eine Testklasse für Ihre Klasse `Person`, so daß Sie die Methoden der Klasse `Person` alle einmal aufgerufen haben.

Aufgabe 13 Schreiben Sie ein kleines Testprogramm, das Objekte des Typs `Buchausleihe` erzeugt und Methoden darauf anwendet.

Aufgabe 14 Suchen Sie sich aus der Javadokumentation zur Klasse `String` weitere Methoden heraus und schreiben Sie ein Testprogramm, daß diese Methoden ausprobiert.

Aufgabe 15 Schreiben Sie eine Klasse `Student`, die die Klasse `Person` um ein weiteres Feld `matrikelNr` erweitert.

Aufgabe 16 Schreiben Sie eine Testklasse, in der Objekte des Typs `Student getestet werden.`

Verzeichnis der Klassen

Abbildungsverzeichnis