

Objektorientiertes Programmieren mit Java

Eine Vorlesung an der Universität Bayreuth im WS 2001/2002

WWW-Seite der Vorlesung

http://www.uni-bayreuth.de/departments/math/~rbaier/lectures/java_multimedia_ws2001

Nur die mit führendem “*” gekennzeichneten Abschnitte beinhalten Informationen, alle anderen Abschnitte verweisen auf die Vorlesung.

Inhaltsverzeichnis

0	* Einführung	4
0.1	* Erste Beispiele	4
0.2	* Literatur	4
0.3	* Was ist Java	6
0.4	* Geschichte	8
0.5	* Ziele	11
0.6	* Java-Applikationen und -Applets	12
0.7	* Koordinatensystem eines Applets	23
0.8	* Der Lebenszyklus eines Applets	24
1	* Grundsätzliches in Java	27
1.1	* Kommentare	27
1.2	* Variablen	28
1.2.1	* Namenskonventionen	28
1.2.2	* Schlüsselwörter	29
1.2.3	* Geltungsbereich von Variablen/Datenelementen	29
1.3	* Elementare Datentypen	30
1.3.1	* explizite Datentypumwandlung (Cast)	31
1.4	* Portable Zeichen mit Unicode	33
1.5	* Operationen mit Datentypen	34
1.5.1	* arithmetische Operatoren	34
1.5.2	* Vergleichsoperatoren	35
1.5.3	* logische Operatoren	35
1.5.4	* Bitoperatoren	35
1.5.5	* Zuweisungsoperatoren	36
1.5.6	* sonstige Operatoren	36
1.5.7	* Vorrang und Assoziativität von Operatoren	37
1.6	* Anweisungen	37
1.6.1	Blockanweisung	37
1.6.2	* bedingte Anweisungen (if, switch)	37
1.6.3	* Wiederholungsanweisungen	44
1.6.4	* Sprunganweisungen	50
1.7	* Arrays	51
1.8	* Grundbegriffe in der Objektorientierung	56
1.8.1	Paket	56
1.8.2	Klasse	57
1.8.3	Objekt	61
1.8.4	Methode	63
1.9	* Umgang mit in Java vordefinierten Klassen	69
1.9.1	* Datenelemente	69
1.9.2	* Konstruktoren	71
1.9.3	* Methoden	74
1.10	* Umgang mit Strings in Java	75
1.10.1	* Anlegen von Strings	77

1.10.2	* Konvertierung von/in Strings	84
1.10.3	* Hauptanwendung von Strings	93
1.11	* Unterschied: Objekte anlegen mit Konstruktoren und Abfrage/Neubesetzung der Verweise	97
1.12	* Wahl des Ortes für Objektdefinitionen	100
2	Grafik mit Applets	105
2.1	* Ändern der Zeichenfarbe	105
2.2	* Zeichenobjekte	106
2.3	* Flacker-/redraw-Problematik	110
2.4	* typische Vorgehensweise	118
2.5	* Beispielprogramme	124
3	* GUI-Programmierung	126
3.1	* Grundlagen	126
3.2	* Elementare GUI-Komponenten	134
3.2.1	* Label	134
3.2.2	* Button	136
3.2.3	* TextField	137
3.2.4	* TextArea	140
3.2.5	* List	141
3.2.6	* Choice	144
3.2.7	* Checkbox und CheckboxGroup	147
3.2.8	Scrollbar	149
3.2.9	Canvas	149
3.3	* Elementare GUI-Container	150
3.3.1	Panel	150
3.3.2	* Window	150
3.3.3	* Frame	154
3.3.4	* Dialog	155
3.3.5	* FileDialog	157
3.3.6	* Scrollpane	159
4	Strings	161
4.1	* Anlegen von Strings	161
4.2	* Konvertierung von/in Strings	167
4.3	* Hauptanwendung von Strings	176
4.4	* Beispielprogramme	180

0 * Einführung

0.1 * Erste Beispiele

Java-Applikation:

vgl. Vorlesung

Java-Applet:

vgl. Vorlesung

0.2 * Literatur

Einstiegsbücher:

1. David **Flanagan**: *Java Examples in a Nutshell*. O'Reilly, dt. Übersetzung, 1998,
2. Kris A. **Jamsa**: *Java Now!* Jamsa Press, 1996, 224 Seiten.
3. Rogers **Cadenhead**: *Teach Yourself Java 1.1 Programming in 24 Hours*. Sams Net, 1997, 384 Seiten.
4. Rogers **Cadenhead**, Mark **Taber**: *Sams' Teach Yourself Java 2 in 24 Hours*. Sams Net, 1999, 450 Seiten (*bereits veraltet*).
dt. Auflage: *Java 1.2 in 21 Tagen*. Markt u. Technik, Haar, 1998, 707 Seiten (*bereits veraltet*).
5. Laura **Lemay**, Rogers **Cadenhead**: *Sams' Teach Yourself Java 2 Platform in 21 Days*. Sams Net, 1999, 800 Seiten.
dt. Auflage: *Java 2 in 21 Tagen*. Markt u. Technik, Haar, 1998, 707 Seiten.

Tutorials, Dokumentationen:

1. <http://java.sun.com/> und das Dokumentationsverzeichnis
<http://java.sun.com/docs/>
2. David **Flanagan**: *Java in a Nutshell*. Deutsche Ausgabe für Java 1.2 und 1.3. O'Reilly, 3. Auflage, 2000, 748 S., ca. 69 DM.
3. David **Flanagan**: *Java Foundation Classes in a Nutshell*. Deutsche Ausgabe für Java 2. O'Reilly, 1. Auflage, 2000, 828 S., ca. 69 DM.
4. Mary **Campione**, Kathy **Walrath** and Alison **Huml**: *The Java Tutorial: A Short Course on the Basics*, 3/e. Addison-Wesley-Longman, 3rd edition, 2000, 448 S.
5. Elliotte Rusty **Harold**: *The Java Developer's Reference: A Tutorial and On-Line Supplement*, 1/e. New York: Prentice-Hall, 1997, 624 Seiten.
6. Stefan **Middendorf**, Reiner **Singer**, Stefan **Strobel**: *Java: Programmierhandbuch und Referenz für die Java-2-Plattform - Einführung und Kernpakete*. Heidelberg: dpunkt.verlag, 1. Auflage, 1997, 827 Seiten.

weitergehende Bücher:

1. Norman **Hendrich**: *Java für Fortgeschrittene* berücksichtigt Java 1.1.
Berlin-Heidelberg-New York-Barcelona-Budapest-Hongkong-London-Mailand-Paris-Santa Clara-Singapur-Tokyo: Springer, 1997, 604 Seiten (inkl. CD).
2. Cay S. **Horstmann**, Gary **Cornell**: Core Java 2, Volume 1: Fundamentals.
(auch genannt: Core Java 1.2, Volume 1: Fundamentals). Prentice-Hall, 4th edition, 1998, 742 Seiten.
dt. Auflage: Core Java 2, Band 1: Grundlagen. Markt u. Technik, Haar, 1999, 880 Seiten, ca. 90 DM.
3. Guido **Krüger**: Go To Java 2. Addison-Wesley, 1. Auflage, 2000, 1224 S., ca. 100 DM.
4. Stefan **Middendorf**, Reiner **Singer**, Stefan **Strobel**: *Java: Programmierhandbuch und Referenz für die Java-2-Plattform - Einführung und Kernpakete*. Heidelberg: dpunkt.verlag, 1. Auflage, 1997, 827 Seiten.
5. Peter **van der Linden**: Just Java 1.2. Java Series. SunSoft Press, 1998, 777 Seiten.

Bücher über Spezialthemen (fortgeschritten):

1. David M. **Geary**: Graphic Java 1.2: Mastering the JFC : AWT. Java Series. Prentice Hall, 1998, 878 Seiten.
2. David M. **Geary**: Graphic Java 1.2: Mastering the JFC : Swing. Volume 2. Java Series. Prentice Hall, 1999, 1622 Seiten.
3. Vincent **Hardy**: Java 2D[tm] API Graphics, Java Series. Sun Microsoft Press, 554 S.
4. Cay S. **Horstmann**, Gary **Cornell**: Core Java 1.2, Volume 2: Advanced Features. Prentice-Hall, 4th edition, 1998, 750 Seiten.
dt. Auflage: Core Java 2, Band 2: Expertenwissen. Markt u. Technik, 2000, 1080 Seiten, ca. 120 DM.
5. Ralf **Kühnel**: *Die Java-1.1-Fibel*. Programmierung von Threads und Applets. Mit den neuen Bibliotheken: Security, Beans, RMI, IDL und SQL.
Bonn-Reading, Massachusetts-Menlo Park, California-New York-Harlow, England-Don Mills, Ontario-Sydney-Mexiko City-Madrid-Amsterdam: Addison & Wesley, 2. aktualisierte und erweiterte Auflage, 1997, 335 Seiten.
6. Elliott Rusty **Harold**: *Java Network Programming*. The Java Series. Cambridge-Köln-Paris-Sebastopol-Tokyo: O'Reilly & Associates Inc., 1997, 441 Seiten.

Online-Informationen:

1. JavaSeries
2. Computerbücher von Addison-Wesley (international)
Anwahl von:
 - i Introduction to Programming -i Java, or Java Textbooks
 - i Programming Languages -i Java

3. Computerbücher von Addison-Wesley (deutsch)
4. Computerbücher von Sun Microsoft Press Anwahl von:
–i Sun Microsoft Press: By Subject–i Java
5. JavaSeries von Sun Microsoft Press Anwahl von:
–i Sun Microsoft Press: By Subject–i Java

0.3 * Was ist Java

Java ist eine vollwertige Programmiersprache. Die Programme starten

- entweder in einem WWW-Browser (Netscape, Internet Explorer, Appletviewer des JDKs, ...) als *Java-Applet*
Einbettung des Java-Applets in eine HTML-Seite, dadurch Start des Java-Applets durch den Browser, der Java-fähig sein muss
- oder von einer Shell/MSDOS-Box aus als *Java-Applikation*
eigenständiges Programm, *keine HTML-Seite und kein WWW-Browser nötig*, aber ein Java-Interpreter

Wichtig: Java ist nicht JavaScript, obwohl die Notation manchmal sehr ähnlich ist. JavaScript gehört eher zu den Skriptsprachen und wird vom Web-Browser interpretiert. Java ist dagegen eine vollständige, sehr umfangreiche, leicht erweiterbare Programmiersprache und nicht auf den Einsatz in Web-Browsern limitiert.

Vorgehen:

Die Java-Programme werden als ganz normale ASCII-Dateien mit irgendeinem Editor (PFE, Notepad, NEdit, XEmacs, ...) unter der Endung ".java" abgespeichert. Diese Java-Programme werden dann kompiliert und der resultierende Bytecode (mit der Endung ".class" interpretiert. Gestartet (und in HTML-Files eingebunden werden also keine Klartext-Dateien, sondern ein sogenanntes binäres File.

wichtige Begriffe:

Java

Programmiersprache

(amerikan. Slangwort für Kaffee)

Die Programmiersprache wurde zuerst vom Green Project und dann von Sun entwickelt. Die Programmiersprache soll objektorientierte, einfache, portable Programme mit Benutzeroberfläche, Multimedia-Fähigkeiten, ... ermöglichen. Erste Anwendungszwecke: Haushaltsgeräte, Settop-Boxen für Fernsehen, Internet, ...

Applet

Java-Programm, das zum Start unbedingt einen *javafähigen Web-Browser* benötigt. Der Bytecode des kompilierten Java-Programmes wird mit dem APPLET-Befehl in einer HTML-Seite gestartet, wenn diese HTML-Seite vom Web-Browser angezeigt wird.

Applikation

Java-Programm, das ein *eigenständiges Programm* ist und daher von einer Shell/MSDOS-Box gestartet wird. Normalerweise wird der Bytecode des kompilierten Java-Programmes vom Java-Interpreter interpretiert und gestartet.

Applikationen haben normalerweise mehr Rechte (weniger Sicherheitseinschränkungen) als Applets, z.B. beim Lesen, Schreiben von lokalen Files, ...

JDK

Java Development Kit

benötigt man, wenn man Java-Programme *selbst übersetzen* will, zum Start von bereits übersetzten Programmen reicht aus das JRE

(von JavaSoft/Sun im Prinzip frei erhältliche *Programmierungsumgebung* inkl. Programmierwerkzeuge wie Appletviewer, Compiler, Interpreter, Debugger, Klassenarchive der Form *.jar und systemabhängige Bibliotheken, ...)

JRE

Java Runtime Environment

benötigt man, wenn man Java-Programme (Java-Applets und -Applikationen) selbst *starten* will

abgemagertes JDK, das auch von kommerziellen Software-Anbietern zusammen mit deren eigenen Java-Programmen an Kunden ausgeliefert werden darf

JVM

Java Virtual Machine

muss im Web-Browser enthalten sein, damit Java-Applets starten können

wird durch den Java-Interpreter bereitgestellt, damit Java-Applikationen starten können

genauer: setzt den Bytecode des kompilierten Java-Programmes

GUI

Graphical User Interface

Benutzeroberfläche, gemeint sind die Eingabe- und Ausgabemöglichkeiten von grafischen Programmen (z.B. Web-Browser selbst, Word, StarOffice, ...), die mit Buttons, Anzeige-/Auswahllisten, Menüs, ...)

SDK

Software Development Kit

Java SDK ist die neue Bezeichnung für JDK

Versionen:

JDK 1.0

völlig veraltet, schlechtes Handling von Windows-Ereignissen wie z.B. Mouseclicks, ...

JDK 1.1

veraltet, hat noch keine neuen modernen GUI-Elemente aber führt Internationalisierung, Sicherheitsaspekte, JavaBeans, Mathematik-Pakete, Aufruf von Programmen auf fremden Rechnern, innere Klassen, ... ein
verbessertes Handling von Windows-Ereignissen
wird *weitgehend* von den allermeisten Browsern (Netscape ab Version 4.0.5, Internet Explorer ab Version unterstützt)

Java 2 SDK Platform Version 1.2

relativ neu, enthält jetzt *moderne GUI-Elemente* im Swing-Paket (Buttons mit Bildern, Anzeigelisten, Trees, ...), neue Klassen für 2D-Grafik, Drag & Drop, Unterstützung für behindertengerechte Anzeige (Bildschirmvergrößerer, Spracherkennung, Tastaturunterstützung, ...) und Verbesserungen im Bereich Audio, JavaBeans, JNI, ...
(kaum Unterstützung der neuen Features in Netscape, Internet Explorer, man benötigt ein Plugin)

Java 2 SDK Platform Version 1.3

neueste ausgereifte Version, enthält HotSpot Server zur Beschleunigung von Java-Programmen, Unterstützung von Java Sound, Java Naming and Directory Interface, Verbesserungen für Swing, Applets, Sicherheit, Drag & Drop, RMI, 2D-Grafik (kaum Unterstützung der neuen Features in Netscape, Internet Explorer, man benötigt ein Plugin)

Java 2 SDK Platform Version 1.4

momentan im Beta-Stadium, soll z.B. XML-Support mit integrieren

verschiedene Distributionen:

Java Runtime Environment (JRE)

Damit kann man nur bereits compilierte Java-Programme starten lassen, enthält z.B. den Java-Compiler "javac" und einige Libraries nicht, die beim Neucompilieren von Java-Programmen benötigt werden. Beinhaltet z.B. den Java-Interpreter und den Appletviewer.

Java (Software) Development Kit (Java SDK/JDK)

Beinhaltet mehr als das JRE, insbesondere alles was zum Neucompilieren von Java-Programmen benötigt wird, also auch den Java-Compiler "javac".

0.4 * Geschichte

Entwicklung der Programmiersprache/Programmierungsumgebung:

1990

Oak (Entwicklung von James Gosling für SGML-Editor)

1990

Green (Entwurf von Patrick Naughton, James Gosling, ...)

1994

Java (1. Hauptprogramm: HotJava-Browser)

Ende 1995

Beta-Version des Java Development Kits (= JDK)

Ende 1995

Netscape und Sun kündigen *Javascript* an.

1996

JDK 1.0

Sun kündigt Just in Time(=JIT)-Compiler an

1997

JDK 1.1

JRE 1.1 (JRE = Java Runtime Environment)

Sun lizenziert Symantecs JIT-Compiler

Ende 1998

JDK 1.2 (drei Tage nach Erscheinen wird es in Java 2 Platform SDK 1.2 umbenannt)

April 1999

HotSpot-Performance Engine

Mai 2000

Java 2 SDK, Enterprise Edition, Version 1.2.1 für Linux

PersonalJava™ Runtime Environment Version 1.0 für Windows CE

Java 3D API Version 1.2 (SGI, Solaris und Windows, basierend auf OpenGL, Linux, basierend auf OpenGL oder Mesa, Portierung auf Direct3D für Windows 98/2000)

Juni 2000

Java 2 SDK, Enterprise Edition, Version 1.2.1 für Solaris und Windows

September/Oktober 2000

Java 2 SDK, Standard Edition, Version 1.3 für Linux und Solaris

Entwicklung der Organisation:

1990

James Gosling (arbeitet an SGML-Editor)

1990

Bill Joy (entwirft neue C++-ähnliche objektorientierte Sprache für Sun)

Ende 1990

Green Project (Patrick Naughton, James Gosling, Mike Sheridan, beeinflusst von Bill Joy)

1992

First Person (Tochter von Sun) entsteht aus Green Project

1993

Zusammenarbeit mit Time Warner auf dem Gebiet interaktiver Settop-Boxen scheitert.

1994

First Person wird aufgelöst, das Team in Sun eingegliedert.

1994

Mosaic Communications Corp. (später Netscape Communications Corp.) wird von Marc Andreessen gegründet, nachdem er NCSA verläßt.

1995

andere Firmen wie Adobe, AT & T, IBM, Intel, SGI, ... lizensieren Java

1996

andere Firmen wie Apple, Corel, DEC, Fujitsu, HP, Microsoft, Sun, Xerox, ... lizensieren Java

1997

andere Firmen wie Informix, Philips, Psion, Siemens, Software AG, WebTV Networks, ... lizensieren Java

Entwicklung des WWW/der Browser:

1990

Tim Berners-Lee (CERN, Genf) veröffentlicht Projektvorschlag für Hypertext-Oberfläche.

1991-1993

Das Protokoll HTTP für Webserver entsteht, 1993 gibt es weltweit 200 WWW-Server. Der Mosaic-Browser von NCSA entsteht.

1994

Patrick Naughton entwickelt in einer Woche WebRunner, den Vorläufer des HotJava-Browsers.

1. Beta-Version von Netscape entsteht (Marc Andreessen verläßt NCSA und gründet Mosaic Communications Corp.)

1995

HotJava ist der 1. WWW-Browser, der eine Java Virtual Machine (= JVM) enthält und Java-Applets starten kann.

Etwas später enthält Netscape 2.0 ebenfalls eine JVM.

1996

Zusammenarbeit mit Time Warner auf dem Gebiet interaktiver Settop-Boxen scheitert.

1994

First Person wird aufgelöst, das Team in Sun eingegliedert.

1994

Mosaic Communications Corp. (später Netscape Communications Corp.) wird von Marc Andreessen gegründet, nachdem er NCSA verläßt.

1996-1997

Microsoft Internet Explorer 3.0 versteht Java 1.0-Applets.

Die Unterstützung von JDK 1.1 ist bei Netscape und beim Internet Explorer nur mangelhaft.

Sun veröffentlicht den Java Activator, ein Plugin (Mechanismus zum Start von browserfremden Programmen im/durch den Browser), das der Java 1.1-Spezifikation genügt. Beginn der gerichtlichen Auseinandersetzung von Sun und Microsoft über die ungenügende Java 1.1-Kompatibilität des Internet Explorers (einige Technologien wie RMI, JNI, ... fehlen!)

1998-1999

Netscape 4.0.1–4.0.4 kann mit einem JDK 1.1-Patch weitestgehend Java 1.1 verstehen (es gibt jedoch Ausnahmen!). Ab Netscape 4.0.5 (auch Netscape 4.5.x) ist dieser Patch bereits in der Netscape-Version enthalten. Microsoft Internet Explorer 3.0.2 und höhere Versionen werden durch einen Upgrade angeblich Java 1.1-kompatibel.

Der HotJava-Browser 3.0 versteht Java 1.1.

1999 wird das Java Plugin (früher Java Activator) in die Java Runtime Environment (= JRE) und das JDK integriert. Es versteht Java 2 Platform (früher JDK 1.2).

0.5 * Ziele

Das White Paper von J. Gosling und H. McGilton (1995) definiert als Ziele:

„Java: Eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architektur-neutrale, portable, dynamische und parallele Sprache mit hoher Performance.“

einfach Vergleich von C-File und Java-Applikation: Einlesen von Vornamen und Nachnamen als Strings und Zusammenfügen der beiden zu einem Namensstring:

C-File `TestString.c`

Java-File `TestString.java`

objektorientiert vgl. die einfachste Hello-World-Applikation bzw. das Applet:

Java-Applikation `HelloWorld_1.java`

Java-Applet `HelloWorldApplet_1.java`

Java-Applet `HelloWorldApplet_2.java`

HTML-File zum Start des Applets: `HelloWorld.html`

verteilt vgl. die geringen Unterschiede im Java-Programm, wenn das Bild als GIF-File lokal vorliegt oder über das Netz geladen wird:

Java-Applikation für lokale Bilder: `Bild.java`

Java-Applikation für Bilder im Netz: `BildURL.java`

interpretiert vgl. den Start von Applikationen bzw. Applets im Java-Interpreter oder in der Java Virtual Machine des WWW-Browsers oder des Appletviewers

robust viele Sicherheitschecks (Arrayindex-Check, Ausnahmebehandlung, ...)

sicher Applets haben stärkere Einschränkungen (z.B. dürfen sie ohne Weiteres keine lokalen Dateien lesen oder erzeugen) als Applikationen. Der Appletviewer oder Netscape lokal gestartet kann dabei dem Applet mehr Rechte einräumen als der Start eines Applets über das Netz.

Java-Applikation zum Lesen eines lokalen Files: `ReadGeheim.java`

Java-Applet zum Lesen eines lokalen Files: `ReadGeheimApplet.java`

HTML-File zum Start des Applets: `ReadGeheimApplet.html`

architektur-neutral, portabel Java liefert auf allen Plattformen (nahezu) dieselben Ergebnisse, z.B. ist der Wertebereich der Standardtypen überall gleich.

Java-Applikation: `Portabel.java`

C-File: `Portabel.c`

dynamisch Strings (Zeichenketten) passen sich automatisch der benötigten Länge an, Arrays, Referenzen auf Objekte werden dynamisch erzeugt

parallel Java kann mehrere Prozesse/Threads erzeugen und managen

Java-Applet: `Parallel.java`

HTML-File zum Start des Applets: `Parallel.html`

schnell vielleicht mit JIT und HotSpot, für Realisierung lang andauernder Algorithmen nicht gut geeignet, besser für GUI-Anwendungen, ...

0.6 * Java-Applikationen und -Applets

Bei Java-Programmen muß zwischen

Java-Applikationen

(eigenständige Programme, die auch ohne WWW-Browser oder Appletviewer lauffähig sind und in einem Java-Interpreter, z.B. "java" vom JDK lokal gestartet werden)

Sie müssen eine Klasse mit der Methode

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

enthalten.

und

Java-Applets

(Programme, die durch ein HTML-Tag in eine HTML-Seite eingebunden werden und die beim Laden dieser Seite durch einen WWW-Browser oder Appletviewer automatisch gestartet werden)

Sie können *nicht* im Java-Interpreter gestartet werden. Zum Start des Applets muss man die HTML-Seite mit dem APPLET-Befehl mit einem java-fähigen Browser (z.B. Internet Explorer, Netscape, Appletviewer vom JDK) anschauen.

unterschieden werden.

Wie schreibe und starte ich eine Java-Applikation?

Arbeitsschritte:

- 0) Falls nötig, installiere JDK Version 1.1 oder 1.2/1.3 (Java 2 Platform).
Das Verzeichnis mit `appletviewer.exe`, `javac.exe`, ... (bzw. unter Linux `appletviewer`, `javac`, ...)

`<Installationsverzeichnis vom JDK>\bin`

muß im Pfad liegen (Windows 95/98 in `autoexec.bat`), siehe Dokumentation zur Installation!

Test, ob Java installiert ist: `appletviewer` oder `javac` in einer MSDOS-Box (Windows) oder einem Shellfenster (Linux) aufrufen.

- 1) Erstelle mit einem Texteditor ein Textfile namens "Bsp_01.java" mit der Dateiendung ".java", z.B. im Verzeichnis "C:\work" (Linux-User: "\${HOME}/work")

```
public class Bsp_01                                // (1)
{
    public static void main(String [] argv)        // (2)
    {
        System.out.println("Java ist einfach schoen!"); // (3)
    }
}
```

- 2) Windows 95/NT: Start einer DOS-Box (Start → Programme → MS-DOS-Eingabeaufforderung)
Linux: Start eines Terminalfensters (abhängig vom Windowmanager bzw. starte "xterm &")

- 3) Windows 95/NT: Wechsle in der DOS-Box in das richtige Verzeichnis:

```
cd C:\work
```

Linux: Wechsle in dem Terminalfenster in das richtige Verzeichnis:

```
cd $HOME/work
```

- 4) Starte in der DOS-Box/dem Terminalfenster den Java-Compiler "javac":

```
javac Bsp_01.java
```

Es entsteht das File "Bsp_01.class", das den portablen Bytecode enthält.

- 5) Starte in der DOS-Box/dem Terminalfenster den Java-Interpreter "java" mit dem Namen der öffentlichen Klasse:

```
java Bsp_01
```

Der Java-Interpreter startet die Interpretation des Bytecodes des Files "Bsp_01.class" für die Klasse "Bsp_01". Damit startet das Java-Programm.

Erläuterungen zum Java-Programm in 1):

- (1) Jedes Java-Programm ist eine Definition einer neuen Klasse. Eine Klasse ist ein neuer *Datentyp*, der Datenelemente und Funktionen definiert, die mit geschweiften Klammern (`{, }`) zusammengefasst werden müssen.
Die Klasse `Bsp_01` mit der Funktion (2) muss eine öffentliche Klasse (`public class`) sein, die man vom MSDOS-Fenster bzw. Shell-Fenster starten kann, d.h. sie darf keine interne Klasse sein.
Die Klasse `Bsp_01` definiert keine Datenelemente, sondern nur eine Funktion `main()` in (2).
- (2) Jede Java-Applikation besteht aus einer öffentlichen Klasse, die genau eine solche Funktion `main()` enthält. Diese muss aus ähnlichen Gründen öffentlich sein (`public`) und objektunabhängig (`static`).
Die Funktion schließt alle Anweisungen und Variablendefinitionen wieder in geschweifte Klammern ein.
Die Funktion hat als Rückgabetypp den Datentyp `void`, d.h. diese Funktion gibt nichts als Ergebnis zurück (an das MSDOS-Fenster oder das Shell-Fenster), deshalb fehlt eine `return`-Anweisung.
Die Funktion erwartet als Übergabeparameter ein (evtl. leeres) Array von Strings (Zeichenketten), vgl. `String []`.
Der Aufruf der Java-Applikation kann also wie folgt aussehen:

```
java Bsp_01  
(kein Argument wird als String uebergeben)
```

oder:

```
java Bsp_01 Bill  
(1. Argument ist der String Bill, wird in argv[0] gespeichert)
```

oder

```
java Bsp_01 Bill Clinton  
(1. Argument ist der String Bill, wird in argv[0] gespeichert,  
2. Argument der String Clinton, wird in argv[1] gespeichert)
```

oder

```
java Bsp_01 "Bill Clinton" 46
```

(1. Argument ist der String "Bill Clinton", wird in argv[0] gespeichert,
2. Argument der String "46", wird in argv[1] gespeichert)

- (3) Die Funktion `main()` besteht nur aus einer einzigen Ausgabeanweisung, die mit ";" abgeschlossen wird. Es wurden keine Variablen definiert.

Die Ausgabefunktion heißt `println(...)` und gibt ein Stringobjekt, hier: `Java ist einfach schoen!`, in dem MSDOS-Fenster/dem Shell-Fenster aus und beendet die Zeile mit einem Newline-Zeichen.

Die Funktion `println(...)` ist allerdings nicht global bekannt, sondern kann nur über ein Objekt der Klasse `java.io.PrintStream` aufgerufen werden (die Klasse `PrintStream` gehört also zum Paket `java.io`). `System.out` ist ein solches Objekt.

kurz gesagt: `System.out` steht für die Standardausgabe in das MSDOS-Fenster/Shell-Fenster und `println(...)` ist die Funktion, die einen String darauf ausgibt.

mögliche Fehlerquellen beim Schreiben/Starten von Applikationen:

- Die öffentliche Klasse `Bsp_01` heißt nicht genauso wie das File "`Bsp_01.java`" ohne die Endung ".java". Insbesondere ist die Klein- und Großschreibung **relevant**.
- Bei der Klassendefinition wurde "`public`" vergessen.
- Die erste Zeile der "`main()`"-Funktion wurde nicht genauso wie im Beispiel spezifiziert.
- Beim Start des Java-Interpreters wurde versehentlich die Endung "`.class`" der Klasse angegeben, d.h. er wurde mit dem Befehl

```
java Bsp_01.class
```

gestartet. Auch die Angabe des Klassennamens muss in richtiger Klein-/Großschreibung erfolgen.

- Das File "`.class`" wurde als Executable (ausführbares Programm) angesehen und ohne den Java-Interpreter gestartet:

```
Bsp_01.class
```

- Das Java-Programm ist ein Java-Applet und gar keine Java-Applikation.

Wie schreibe und starte ich eine Java-Applet?

Arbeitsschritte:

- 0) Falls nötig, installiere JDK Version 1.1 oder 1.2/1.3 (Java 2 Platform).
Das Verzeichnis mit `appletviewer.exe`, `javac.exe`, ... (bzw. unter Linux `appletviewer`, `javac`, ...)

`<Installationsverzeichnis vom JDK>\bin`

muß im Pfad liegen (Windows 95/98 in `autoexec.bat`), siehe Dokumentation zur Installation!

Test, ob Java installiert ist: `appletviewer` oder `javac` in einer MSDOS-Box (Windows) oder einem Shellfenster (Linux) aufrufen.

- 1) Erstelle mit einem Texteditor ein Textfile namens "Bsp_02.java" mit der Dateiendung ".java", z.B. im Verzeichnis "C:\work" (Linux-User: "\${HOME}/work")

```
public class Bsp_02 extends java.applet.Applet           // (1)
{
    public void paint(java.awt.Graphics g)                // (2)
    {
        g.drawString("Java startet als Applet!", 20, 50); // (3)
    }
}
```

- 2) Windows 95/NT: Start einer DOS-Box (Start → Programme → MS-DOS-Eingabeaufforderung)
Linux: Start eines Terminalfensters (abhängig vom Windowmanager bzw. starte "xterm &")

- 3) Windows 95/NT: Wechsle in der DOS-Box in das richtige Verzeichnis:

```
cd C:\work
```

Linux: Wechsle in dem Terminalfenster in das richtige Verzeichnis:

```
cd $HOME/work
```

- 4) Starte in der DOS-Box/dem Terminalfenster den Java-Compiler "javac":

```
javac Bsp_02.java
```

Es entsteht das File "Bsp_02.class", das den portablen Bytecode enthält.

- 5) Erstelle mit einem Texteditor ein Textfile beliebigen Namens, z.B. "Test.htm" oder "Test.html" mit HTML-Kommandos und der Dateiendung ".htm" bzw. ".html" im selben Verzeichnis "C:\work" (Linux-User: "\${HOME}/work"). Das HTML-File muß im BODY-Teil den APPLET-Befehl beinhalten:

```
<APPLET CODE="Bsp_02.class" HEIGHT=300 WIDTH=500>
  Start des Java-Applets gescheitert! Browser-Einstellungen kontrollieren
  (Start von Java-Programmen erlaubt?) oder einen Java-fähigen Browser
  besorgen!
</APPLET>
```

Als Wert für CODE muß dabei der Name des Files "Bsp_02.class" mit dem Java-Bytecode enthalten. WIDTH und HEIGHT spezifizieren Breite und Höhe des Applets in der HTML-Darstellung des Browsers. Der Text innerhalb des APPLET-Befehls wird nur sichtbar, wenn der Start des Java-Applets scheitert. Appletfähige Browser interpretieren nur noch den PARAM-Befehl innerhalb des APPLET-Befehls.

- 6a) Starte Netscape oder den Internet Explorer und öffne das lokale HTML-File "Test.htm" bzw. "Test.html" im Browser. Beim Darstellen des HTML-Files stößt der Browser auf den APPLET-Befehl, startet daraufhin beim ersten Mal seine Java Virtual Machine und dann das Java-Applet (genauer: lädt den Bytecode "Bsp_02.class") und setzt danach die Interpretation des HTML-Files fort.
- 6b) Starte in der DOS-Box/dem Terminalfenster (aktuelles Verzeichnis wie in 3.) den Appletviewer "appletviewer":

```
appletviewer Test.html
```

Dabei werden jedoch alle HTML-Kommandos außer dem APPLET-Befehl **ignoriert**. Das Applet startet jedoch im Appletviewer.

Erläuterungen zum Java-Programm in 1):

- (1) Auch jedes Java-Applet besteht aus einer Definition einer neuen Klasse "Bsp_02", die von der in Java bekannten Klasse `java.applet.Applet` abgeleitet ist. Damit erbt sie Eigenschaften und Funktionen von dieser Klasse, z.B. ist die Startreihenfolge bei dieser Klasse "Bsp_02" genau dieselbe wie für die vordefinierte Klasse `java.applet.Applet`. Die Klasse `Bsp_02` mit der Funktion (2) muss eine öffentliche Klasse (`public class`) sein, damit sie der Webbrowser bzw. der Appletviewer starten kann. Jedes Java-Applet ist eine von der Klasse `java.applet.Applet` abgeleitete Klasse.
- (2) Die eigene Klasse `Bsp_02` überschreibt die in der Klasse `java.applet.Applet` als nichtstehend vordefinierte Funktion

```
public void paint(java.awt.Graphics g)
```

Ein Überschreiben bedeutet ein Neuschreiben einer bereits in einer übergeordneten Klasse definierten Funktion. Die neu definierte Funktion ersetzt damit die alte in der Klasse `java.applet.Applet` vordefinierte Funktion. Diese Funktion wird beim Start des Applets im Webbrowser/Appletviewer nach einer festen Reihenfolge aufgerufen. Der Name deutet schon an, dass die Funktion hauptsächlich zum Zeichnen in das Appletfenster im Webbrowser gedacht ist. Die Anweisung (3) zeichnet also einen String. Die Funktion hat als Rückgabetyt den Datentyp `void`, d.h. diese Funktion gibt nichts als Ergebnis zurück, deshalb fehlt wieder eine `return`-Anweisung.

Die Funktion erwartet als Übergabeparameter ein Objekt der Klasse "`java.awt.Graphics`", das für den Grafikkontext des Applets innerhalb des Webbrowsers/Appletviewers steht. Etwas verwirrend für Einsteiger in die Objektorientierung ist, dass man *nicht* für den Aufruf dieser Funktion `paint(...)` und ein passendes Grafikkontext-Objekt sorgen muss. Dafür ist bereits in der vordefinierten Klasse `java.applet.Applet` gesorgt worden.

- (3) Die Funktion `paint()` besteht nur aus einer einzigen Zeichenanweisung `drawString(...)`, die mit ";" abgeschlossen wird. Die Zeichenfunktion zeichnet in einen Bereich des Webbrowsers, dessen Größe durch den `APPLET`-Befehl im HTML-File mit den Werten von `WIDTH` (Breite) und `HEIGHT` (Höhe) festgelegt wird. (20, 50) sind (x, y)-Wert im Koordinatensystem dieses Bereiches (linke untere Ecke, ab der der String gezeichnet wird). Die Funktion `drawString(...)` ist allerdings nicht global bekannt, sondern kann nur über ein Objekt der Klasse `java.awt.Graphics` aufgerufen werden. `g` ist ein solches Objekt und steht für den Grafikkontext des Appletbereichs im Webbrowser/Appletviewer.

import-Anweisung:

Das Java-Programm für das Applet kann noch folgendermaßen modifiziert werden. Um anzudeuten, daß der kürzere Name "Applet" für die Klasse "`java.applet.Applet`" steht (vgl. (6)), d.h. für die Klasse "Applet" im Paket "`java.applet`", kann man die "import"-Anweisung in (4) verwenden. Vor allem, wenn Klassennamen öfter angegeben werden müssen, ist eine kürzere Schreibweise erwünscht.

```
import java.applet.Applet;           // (4)
import java.awt.*;                   // (5)

public class Bsp_02 extends Applet   // (6)
{
    public void paint(Graphics g)     // (7)
    {
        g.drawString("Java startet als Applet!", 20, 50);
    }
}
```

Die "import"-Anweisung "import java.awt.*" in (5) führt für alle Klassen im Paket "java.awt" diese abkürzende Schreibweise ein, insbesondere für die Klasse "Graphics", vgl. (7). Damit könnte man also auch "Button" statt "java.awt.Button" schreiben. **mögliche Fehlerquellen beim Schreiben/Starten von Applets:**

mögliche Fehlerquellen beim Schreiben/Starten von Applets:

- Die öffentliche Klasse heißt nicht genauso wie das File (ohne die Endung ".java"). Klein- und Großschreibung sind wieder **relevant**.
- Der Name des Java-Bytecodes (Endung ".class") im HTML-File ist falsch.
- Bei der Klassendefinition wurde "public" vergessen.
- Bei der Klassendefinition wurde "extends java.applet.Applet" bzw. die Kurzform "extends Applet" vergessen.
- Die Kurzform "extends Applet" wurde verwendet, die zugehörige "import"-Anweisung aber vergessen (Vergleichbares gilt für "Graphics").
- Die erste Zeile der "paint()" -Funktion wurde nicht genauso wie im Beispiel spezifiziert.
- Der Browser ist nicht Java-fähig bzw. der Start von Java-Programmen wurde in der Einstellung des Browsers untersagt (Problem tritt beim Appletviewer nicht auf).
- Der Browser kommt mit der im Java-Applet verwendeten JDK-Version nicht zurecht (Browser verwendet veraltete Java-Version; Problem tritt beim Appletviewer nicht auf).
In Netscape kann man zur besseren Fehlersuche die Java-Console öffnen, im Internet Explorer muß man bei älteren Versionen das Anlegen eines Logfiles beim Start von Java-Programmen veranlassen und dieses auswerten oder ab Version 4.x die Java-Konsole aktivieren. Typische Fehlermeldungen beinhalten dann meistens die Worte "class/interface/method not found".
- Das Java-Applet verletzt die Sicherheitsbestimmungen von Java (versucht z.B. ein lokales File zu lesen oder zu schreiben) und scheitert deshalb, d.h. der Start des Applets wird abgebrochen (siehe Java-Console oder Java-Logfile). Typische Fehlermeldungen beinhalten dann meistens die Worte "security exception".

- Beim Start des Appletviewers wurde versehentlich das Java-Bytefile "Bsp_02.class" angegeben, d.h. er wurde mit dem Befehl

```
appletviewer Bsp_02.class
```

gestartet.

- Das Java-Programm ist eine Java-Applikation und gar kein Java-Applet.

Der Umgang mit Java im Browser

* Start eines Applets auf der WWW-Seite

Um ein Applet zu starten, muß der WWW-Browser javafähig sein (nicht javafähig sind z.B. Mosaic, Internet Explorer 2.0, Netscape 1.1) und es muß in den Optionen der Start von Java-Programmen erlaubt sein.

Microsoft Internet Explorer 3.0	"View → Options → Security" unter "Active Content" ist die Checkbox "Enable Java programs" angeklickt
Microsoft Internet Explorer 4.0, 5.x	"Ansicht → Internet Optionen → Sicherheit": Checkbox "Hohe Sicherheit" ist <i>nicht</i> angeklickt bzw. Checkbox "angepaßt" ist angeklickt, aber in den "Einstellungen" ist im Abschnitt "Java", Unterabschnitt "Java-Einstellungen" ist weder "hohe Sicherheit" noch "Java deaktivieren" angeklickt
Netscape 2.0	"Options → Security Preferences → General" ist die Checkbox "Disable Java" nicht angeklickt
Netscape 3.0	"Options → Network Preferences → Languages" ist die Checkbox "Enable Java" angeklickt
Netscape 4.x, 6.0	"Edit → Preferences → Advanced" ist die Checkbox "Enable Java" angeklickt bzw. "Bearbeiten → Einstellungen → Erweitert" ist die Checkbox "Java aktivieren" angeklickt

* Laden einer lokalen WWW-Seite im Browser

Die lokal vorhandene WWW-Seite "StartApplet.html" kann durch

Microsoft Internet Explorer 3.0	"File → Open File"
Microsoft Internet Explorer 4.0, 5.0, 5.5	"Datei → Datei öffnen"
Netscape 2.0	"File → Open File"
Netscape 3.0	"File → Open File"
Netscape 4.x, 6.0	"File → Open Page → Choose File"

und anschließender Selektion von "StartApplet.html" (ggf. im geeigneten Unterverzeichnis) in der File-Auswahlbox (bei Netscape 4.0 durch abschließendes Drücken von "Open in Navigator") erfolgen.

Alternativ kann bei Netscape auch von der DOS-Box/Shell aus durch

```
netscape StartApplet.html
```

und in der DOS-Box auch durch

```
"c:\Programme\Plus!\Microsoft Internet\iexplore" StartApplet.html
```

(Verzeichnis für S 81) das Applet gestartet werden.

Wenn man nur an dem Applet und nicht an der WWW-Seite interessiert ist, kann man das Applet auch mit dem Appletviewer des JDK starten:

```
appletviewer StartApplet.html
```

Der Vorteil des Appletviewer ist die größtmögliche Kompatibilität zur aktuellsten JDK-Version und die einstellbaren Sicherheitsattribute (Netz- und Dateizugriff).

*** Die Java-Console/das Java-Logfile**

Mögliche Ausgaben des Java-Applets mit `"System.out.print()/System.out.println()"` (verwendet Standardausgabekanal) oder mit `"System.err.print()/System.err.println()"` (verwendet Standardfehlerkanal) können auf der Java-Console bzw. in einem Java-Logfile angesehen werden. Eine Console ist ein Fenster, in dem man Ausgaben lesen kann, ein Schreiben in der Java-Console ist nicht möglich. Ein Logfile ist eine Datei, in die nacheinander alle Ausgaben mit `"System.out"` bzw. `"System.err"` angehängt werden, die das Java-Applet erzeugt. Dies erfolgt durch:

Microsoft Internet Explorer 3.0	Anklicken der Checkbox "Enable Java logging" unter "View → Options → Advanced" Beenden und Wiederaufrufen des Internet Explorers; alles findet sich unter der Datei "javalog.txt" im Unterverzeichnis "java" des Windows-Hauptverzeichnisses.
Microsoft Internet Explorer 4.0	"Ansicht → Internet Optionen → Erweitert": im Abschnitt "Java VM" ist "Java-Konsole aktiviert" angeklickt (Neustart nötig, nicht aktivierbar im S 81)
Microsoft Internet Explorer 5.5	"Ansicht → Java Befehlszeile" oder bei Verwendung des Sun Java Plugins: aktiviere in der Systemsteuerung mit dem Java-Plugin-Control die Java-1.3-Konsole
Netscape 2.0	Anklicken der Checkbox "Show Java Console" im Menü "Options"
Netscape 3.0	Anklicken der Checkbox "Show Java Console" im Menü "Options"
Netscape 4.0	Aufruf von "Java Console" im Menü "Window" (Navigator) bzw. "Communicator" (Communicator)
Netscape 4.5	Aufruf von "Java Console" im Menü "Communicator" (Communicator) im Untermenü "Extras"
Netscape 4.7	Aufruf von "Java Konsole" im Menü "Communicator" im Untermenü "Werkzeuge"
Netscape 6.0	Aufruf von "Java Konsole" im Menü "Aufgaben" im Untermenü "Extras" oder bei Verwendung des Sun Java Plugins: aktiviere in der Systemsteuerung mit dem Java-Plugin-Control die Java-1.3-Konsole

0.7 * Koordinatensystem eines Applets

vgl. zusätzlich die Vorlesung!

Die im HTML-File mittels des APPLET-Befehls mit WIDTH bzw. HEIGHT festgelegte Breite bzw. Höhe des Appletbereichs im Browserfenster kann im Java-Programm ermittelt werden. Ein Objekt der Klasse Dimension im Paket java.awt speichert dabei Breite und Höhe in seinen Datenelementen width bzw. height.

```
public void paint(Graphics g)
{
    ...
    // Ermitteln der mit WIDTH/HEIGHT festgelegten Anfangsgrösse
    Dimension dim = getSize();
    int breite = dim.width;
    int hoehe = dim.height;
    System.out.println("Appletgrösse: " + width + " x " + height
        + " Pixel");
    ...
}
```

Mit der Methode "void setSize(int neue_breite, int neue_hoehe);" bzw. "void setSize(Dimension neue_dimension);" kann eine neue Größe festgelegt werden.

Als Koordinaten sind dann

x-Werte zwischen 0 und breite-1 und

und

y-Werte zwischen 0 und hoehe-1

erlaubt. Andere Koordinatenangaben werden ignoriert und die Pixel nicht gezeichnet.

Beispiel:

Im HTML-File steht der Applet-Befehl:

```
<APPLET CODE="Bsp_Applet.class" WIDTH=500 HEIGHT=300>
  Fehler! Applet kann nicht starten!
  Der Browser kann kein Java, oder es ist abgeschalten!
</APPLET>
```

Wenn das Applet mit obiger init()-Routine startet, enthält

```
breite = dim.width = 500,
hoehe = dim.height = 300.
```

Verwendet man die Methode "getSize()" in der "paint()" -Methode, steht beim ersten Aufruf von "paint()" in dem Objekt dim die im HTML-File festgelegte Größe 500x300, bei wiederholten Aufrufen findet sich allerdings die aktuelle Größe des Appletfensters wieder (kann verändert werden durch Größer-/Kleinerziehen des Fensters vom Appletviewer oder vom WWW-Browser).

```

public void paint(Graphics g)
{
    ...
    // 1. Aufruf: Ermitteln der mit WIDTH/HEIGHT festgelegten Anfangsgroesse
    // weitere Aufrufe: Ermitteln der aktuellen Groesse des Appletfensters

    Dimension dim = getSize();
    int breite = dim.width;
    int hoehe = dim.height;
    System.out.println("Appletgroesse: " + width + " x " + height
        + " Pixel");
    ...
}

```

Achtung: *Größere* y-Werte für den Appletbereich sind in diesem Koordinatensystem weiter *unten* und nicht wie im üblichen Koordinatensystem weiter oben. Der Ursprung des Koordinatensystems mit Koordinaten (0,0) ist also in der *linken oberen* Ecke!

Beispiel: Es sollen Strings untereinander im Applet ausgegeben werden.

```

public void paint(Graphics g)
{
    g.drawString("Oben steht das relativ weit links!", 20, 50);
    g.drawString("Weiter unten steht das relativ weit links!", 20, 100);
    g.drawString("Recht weit oben und rechts steht das!", 150, 10);
    g.drawString("Recht weit unten und rechts steht das!", 150, 200);
}

```

Professioneller ermittelt man die Größe des Appletbereichs und die Fonthöhe bzw. Breite in Pixels zur Ausgabe von Strings und arbeitet mit diesen Werten, anstatt die Koordinaten irgendwie passend zu wählen.

0.8 * Der Lebenszyklus eines Applets

vgl. zusätzlich die Vorlesung!

Ein Applet (im Paket `java.applet` definiert) ist eine Spezialisierung (auch Ableitung, Unterklasse genannt) von `Panel`. Beides sind Klassen, die in Java bereits vordefiniert sind. Damit sind bereits alle Methoden festgelegt und definiert. Da es Interaktionen zwischen Applet und dem WWW-Browser gibt, werden abhängig von Aktionen mit dem WWW-Browser bzw. dessen Darstellungsfenster bestimmte Methoden des Applets aufgerufen.

Diese Methoden sind jedoch in der Klasse `Applet` so definiert, daß sie nichts tun. Um ein eigenes Applet zu entwerfen, definiert man als erstes eine eigene Spezialisierung (Ableitung, Unterklasse) der vordefinierten Klasse `Applet` und überschreibt nur diejenigen Methoden, die man ändern möchte. Man muß daher die Ablaufreihenfolge eines Applets kennen, um zu wissen, was man ändern muß und was man von der vordefinierten Klasse `Applet` unverändert übernehmen kann.

Reihenfolge des Aufrufs einer selbstgeschriebenen Klasse `HelloApplet`:

- (1) Konstruktor (sehr selten verwendet)

```
public HelloApplet()
```

Ein Konstruktor ist eine spezielle Methode, deren Name der Name der Klasse ist, die keinen Rückgabebetyp hat und ansonsten beliebig viele Argumente hat. In diesem Fall hat der Konstruktor keine Argumente und heißt Standardkonstruktor.

Er wird sehr selten verwendet, weil es Vorteile hat, die Initialisierung in (2) zu erledigen. Aufruf: Der Konstruktor wird noch vor (2) genau einmal beim Start des Applets aufgerufen (Appletviewer: auch bei Menüpunkt Reload nach (6)).

- (2) Initialisierungsmethode `init()` (sehr häufig verwendet)

```
public void init()
```

Hier wird der Bytecode geladen und verifiziert. Aufruf: Diese Methode wird genau einmal beim Start des Applets vom Browser aufgerufen (Ausnahme: Reload des Applets). Im Appletviewer: bei den Menüpunkten Restart nach (6) und bei Reload nach (1)

Zweck: Initialisierung von Datenelementen, Aufbau der GUI-Elemente, Start der Ereignisbehandlung, ...

Damit der Browser nicht "hängen" bleibt, sollten lange Berechnungen in dieser Methode vermieden werden (Ausweg: (3)).

- (3) Startmethode `start()` (nicht so häufig verwendet)

```
public void start()
```

Hier sollten parallel auszuführende Programmteile (sog. Threads oder parallele Prozesse) gestartet werden. Langwierige Berechnungen müssen in einem zusätzlichen Thread gestartet werden.

Aufruf: Diese Methode wird immer dann aufgerufen, wenn das Browserfenster neu aktiviert wird (Restart des Applets, z.B. nach Wiederaufruf der HTML-Seite durch Vorwärts-/Rückwärtsblättern, Wiederherstellen des Fensters nach Iconifizierung, ggf. Hinstrollen zur Appletausgabe auf langer HTML-Seite, ...). Im Appletviewer: bei den Menüpunkten Reload bzw. Restart nach (2) und bei Start direkt

Zweck: Start zusätzlicher Threads, Aufbau weiterer Fenster neben dem Browserfenster

- (4) Zeichenmethode `paint()` (fast immer verwendet)

```
public void paint(Graphics g)
```

Hier müssen alle Grafikausgaben für das Applet eingefügt werden (Zeichnen von Linien/Kreisen/Rechtecken/..., Ausgabe von Text als Grafik, Löschen/Neuzeichnen von Bildschirmteilen, Setzen von Vorder-/Hintergrundfarbe, ...). Ein direktes Zeichnen in das Appletfenster ist nur über den Umweg seines Grafikkontextes (Klasse `Graphics`) möglich.

Aufruf: direkt nach `start()` und dann immer wieder, wenn ein Neuzeichnen des Browserfensters initiiert wird (z.B. durch Größenveränderung des Browserfensters, teilweises

Verdecken des Appletanteils im Browserfenster durch andere Fenster, indirektes Anfordern mit Aufruf der Methode `repaint()`, ...). Im Appletviewer: bei den Menüpunkten Reload, Restart und Start nach (3)

Zweck: obige Grafikoperationen, nicht Aufbau der GUI

- (5) Stoppmethode `stop()` (Gegenstück zu `start()`)

```
public void stop()
```

Hier sollten parallel auszuführende Programmteile (sog. Threads oder parallele Prozesse) gestoppt werden. Langwierige Berechnungen sollten angehalten werden, wenn die HTML-Seite verlassen wird bzw. iconisiert wird.

Aufruf: Diese Methode wird immer dann aufgerufen, wenn das Browserfenster deaktiviert wird (Stopp des Applets, z.B. nach Verlassen der HTML-Seite durch Vorwärts-/Rückwärtsblättern/Aufruf anderer URL, Iconifizierung des Fensters, ggf. Wegscrollen von der Appletausgabe auf langer HTML-Seite, ...). Im Appletviewer: bei den Menüpunkten Reload bzw. Restart als erste Aktion, bei Close oder Quit vor (6) und bei Stop als alleinige Aktion

Zweck: Stopp zusätzlicher Threads, Verschwinden weiterer Fenster neben dem Browserfenster

- (6) Beendigungsmethode `destroy()` (Gegenstück zu `init()`, sehr selten verwendet)

```
public void destroy()
```

Hier müssen alle wichtigen Aufräumarbeiten durchgeführt werden, die eine Beendigung des Applets ggf. beschleunigen.

Aufruf: Diese Methode wird genau einmal bei Beendigung des Applets vom Browser aufgerufen, z.B. wenn der Bytecode des Applets aus dem browserinternen Cache wegen Speicherbedarfs gelöscht wird. Im Appletviewer: bei den Menüpunkten Reload bzw. Restart als zweite Aktion nach (5) und bei Close oder Quit nach (5)

Zweck: Ermöglichen eines schnelleren Beenden des Applets, z.B. durch explizite Freigabe von Memory/Daten/... oder durch explizites Beenden von weiteren Fenstern

Beispiele:

Soll ein eigenes Applet nur etwas zeichnen und hat es keine eigenen Datenelemente und daher auch keine GUI-Elemente, überlädt (auch überschreibt genannt) man nur die Methode `paint()`.

```
import java.applet.Applet;
import java.awt.Graphics;

public HelloApplet extends Applet
{
    // nur paint() wird neu definiert
    public void paint(Graphics g)
    {
        g.drawString("Hello, world", 20, 40);
    }
}
```

```
}
```

Der Konstruktor wird weggelassen und die Methoden `init()`, `start()`, `stop()` und `destroy()` von der Klasse `Applet` übernommen.

Zeichnet ein Applet nichts, sondern gibt nur etwas aus oder verwendet es GUI-Elemente, dann schreibt man nur die `init()`-Methode neu. Alle anderen Methode wie auch `paint()` werden von der Klasse `Applet` übernommen.

```
import java.applet.Applet;
import java.awt.Graphics;

public AusgabeApplet extends Applet
{
    // nur init() wird neu definiert
    public void init()
    {
        // festgelegt Groesse des in der HTML-Seite im APPLET-Befehl
        // in den Parametern WIDTH bzw. HEIGHT

        Dimension dim = getSize();
        System.out.println("Breite des Applets: " + dim.width);
        System.out.println("Hoehe des Applets: " + dim.height);
    }
}
```

1 * Grundsätzliches in Java

1.1 * Kommentare

Für Kommentare gibt es in Java 3 Möglichkeiten, wobei zwei genauso sind wie in C++.

```
//          : leitet einen einzeiligen Kommentar ein
            : kann auch mitten in einer Zeile verwendet werden
/*         : leitet einen ein-/mehrzeiligen Kommentar ein
...
*/
/**        : leitet einen Kommentar ein, den das Java-Tool javadoc verwenden kann,
            : um HTML-Dokumentationen aus dem Source-File zu generieren
...
*/
```

Beispiel:

```
// import der AWT-Klassen
import java.awt.*;
import java.applet.Applet; // import der Klasse Applet
```

```

/*
   XYZ ist abgeleitet von der Java-Klasse
   Applet
*/
public class XYZ extends Applet
{
    ...
}

```

1.2 * Variablen

Java erlaubt als Namen von Variablen, Klassen, Methoden, ...

- bel. lange Folge von Java-Buchstaben und -Ziffern
- 1. Zeichen muß Java-Buchstabe sein (a..z, A..Z, -, \$, zusätzlich alle Zeichen, die beim Aufruf von `Character.isUnicodeIdentifizierStart() == true` liefert)

Außerdem gilt:

- Bezeichner können Unicode-Sequenzen enthalten, z.B.

```
int R\u00fcben = 10; // Rüben
```

- Bezeichner sind gleich, falls die Unicode-Zeichenfolge gleich sind.

`R\u00fcben` und `Rüben` sind gleich!

1.2.1 * Namenskonventionen

- Namen von Klassen und Schnittstellen beginnen mit einem Großbuchstaben.
Bsp.: `Applet`, `Date`, `ActionListener`, ...
- Namen von Variablen und Methoden beginnen mit kleinen Buchstaben.
Bsp.: `anzahl`, `add()`, `getLabel()`, ...
- Alle Package-Namen (außer in Packages von Fremdherstellern bzw. im Top-Level-Internet-Domain) werden kleingeschrieben.
Bsp.: `java.awt`, `java.applet`, `java.lang`, ..., aber `org.omg.CORBA`, `DE.suse.www`, ...
- Namen von Konstanten werden vollständig groß geschrieben.
Bsp.: `MAX_VALUE` in `java.lang.Double`, `PI` in `java.lang.Math`, `CENTER` in `java.awt.Label`, ...
wenige Ausnahmen, z.B. `NaN` in `java.lang.Double`
- Anfangsbuchstaben von mehrteiligen Bezeichnern werden (bis auf den 1. Buchstaben) großgeschrieben; bei Konstanten wird der Unterstrich "_" verwendet.
Bsp.: `ActionListener`, `MIN_VALUE` in `java.lang.Double`, `FlowLayout`, ...

Beispiele:

```
int getX()
{
    return x;
}

void setRadius(double r)
{
    radius = r;
}

static final int MAX_NUM = 3;

java.util.Vector monatsUmsaetze;

public class EineBesondereKlasse
{
    ...
}

double einGrosserWert = 1.0E+100;
```

1.2.2 * Schlüsselwörter

Dies sind Bezeichner, die nicht anderweitig verwendet werden dürfen:

abstract	double	int	static
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float		throw
char	for	package	throws
class	goto ¹	private	transient ¹
const ¹	if	protected	try
continue	implements	public	void
default	import	return	volatile
do	instanceof	short	while

¹ sind zur Zeit nicht von Java benutzt, die Klassifikation als Schlüsselwörter erlaubt aber teilweise eine bessere Fehlererkennung.

false, null, true sind Literale (d.h. unbenannte Konstanten), die aber wie Schlüsselwörter nicht verwendet werden dürfen.

1.2.3 * Geltungsbereich von Variablen/Datenelementen

siehe Vorlesung

1.3 * Elementare Datentypen

Elementare Datentypen (auch Standarddatentypen genannt) sind die einzigen Datentypen in Java, die nicht als Klassen realisiert sind.

Typ	Wrapperklassen	Wertebereich	Defaultwert	Speicherplatz
byte	Byte	-128 ... 127	0	1 Byte
short	Short	-32768 ... 32767	0	2 Bytes
int	Integer	$-2^{31} \dots 2^{31} - 1$	0	4 Bytes
long	Long	$-2^{63} \dots 2^{63} - 1$	0L	8 Bytes
boolean	Boolean	false, true	false	1 Bit
char	Character	\u0000 ... \uFFFF	\u0000	2 Bytes
float	Float	$\pm 3.40282E+38$	0.0F	4 Bytes
double	Double	$\pm 1.79769E+308$	0.0D	8 Bytes

Eine Variable eines Standarddatentyps legt man nicht mit einem `new`-Aufruf an, sondern gibt vor der ersten Verwendung an, welchen Datentyp sie hat. Eine Reihenfolge bei der Definition verschiedener Standarddatentypen muß man nicht einhalten. Man kann durchaus mehrere `int`-Variablen nacheinander anlegen.

```
// Variable "breite" vom Datentyp int angelegt
int breite;

// zwei Variablen "x" und "y" vom Datentyp double angelegt
double x, y;

// nochmal eine int-Variable angelegt
int hoehe;
```

Gibt man keinen Initialisierungswert vor, werden die Variablen automatisch mit dem Standardwert vorinitialisiert.

```
// Variable "breite" hat als Anfangswert 0
int breite;

// "x" hat Anfangswert 3.0, "y" den Anfangswert -4.0
double x = 3.0, y = -4.0;

breite = 200;

// hoehe erhaelt als Anfangswert 2*200 = 400
int hoehe = 2*breite;
```

ganzzahlige Konstanten (Details):

Für Integer-Typen sind die

- dezimale Schreibweise
- oktale Schreibweise (führende Null)
- hexadezimale Schreibweise (0x oder 0X am Anfang)

für Konstanten möglich.

Fließkommazahlen können als Exponentenzeichen e bzw. E enthalten.

Genauigkeitsangaben am Ende eines Literals:

l/L	Datentyp long
f/F	Datentyp float
d/D	Datentyp double
int	Standard für Literale von Integerdatentypen
double	Standard für Literale von Fließkommadatentypen

1.3.1 * explizite Datentypumwandlung (Cast)

Java schränkt die automatische Typumwandlung (wie in C/C++ weit verbreitet) sehr stark ein. Automatisch wird nur umgewandelt, wenn der Wertebereich des umzuwandelnden Typs im Wertebereich des Typs des zugewiesenen Objekts enthalten ist. Erzwingen kann man jedoch eine Umwandlung in elementare Datentypen durch explizites Anwendung von Casts. Beispiele:

a) automatische Umwandlung

```
byte b = 64;
int i;

...

i = b;
```

b) Fehlermeldung, obwohl Wert im Wertebereich von byte

```
byte b;
int i = 64;

...

b = i;
```

c) explizites Casten

```
int i = 64;
byte b;
```

...

```
b = (byte) i;
```

- d) `short` und `char` belegen zwar beide 2 Bytes im Speicher, trotzdem sind sie nicht zuweisbar

```
char c = 'A';  
short s;
```

...

```
s = c;           // Fehler!  
s = (short) c;  // richtig! s speichert 65 (ASCII-Wert von 'A')
```

```
s = 48;  
c = s;          // Fehler!  
c = (char) s;   // richtig! c speichert jetzt Zeichen '0'  
                // 48 ist ASCII-Wert von '0'
```

Es gibt die folgende Hierarchie für automatische Umwandlungen:

`byte` → `short` → `int` → `long` → `float` → `double`

Bei der Umwandlung von `long` und `float/double` bleibt zwar die Größenordnung der Zahl erhalten, jedoch nicht alle Stellen (wegen der fehlenden Genauigkeit der Fließkommazahlen). Die Umwandlungen eines sog. skalaren Datentyps in `boolean` werden *nicht* unterstützt in Java.

- a) `int` in `boolean`

```
int erg, zaehler = 3, nenner = 2;  
boolean ungleichNull;  
  
ungleichNull = nenner;           // falsch  
ungleichNull = (boolean) nenner; // auch falsch  
ungleichNull = (nenner != 0) ? true : false; // richtig
```

- b) `boolean` in `int`

```
boolean b = true;  
int zahl;  
  
zahl = b;           // falsch  
zahl = (int) b;     // auch falsch  
zahl = b ? 1 : 0;   // richtig
```

Bei expliziter Typumwandlung (Cast) werden ggf. die passenden niederwertigen Bytes kopiert und die höherwertigen ignoriert (→ evtl. Datenverlust).

1.4 * Portable Zeichen mit Unicode

Escape-Sequenz	Unicode-Sequenz	Bedeutung	englischer Name
<code>\b</code>	<code>\u0008</code>	Backspace	backspace
<code>\f</code>	<code>\u000C</code>	Seitenvorschub	form feed
<code>\n</code>	<code>\u000A</code>	Zeilenvorschub	line feed/newline
<code>\r</code>	<code>\u000D</code>	Wagenrücklauf	carriage return
<code>\t</code>	<code>\u0009</code>	horizontaler Tabulator	horizontal tab
<code>\\</code>	<code>\u005C</code>	inverser Schrägstrich	backslash \
<code>\'</code>	<code>\u0027</code>	Apostroph '	single quote
<code>\"</code>	<code>\u0022</code>	Doppelanführungszeichen oben "	double quote
<code>\d</code> <code>\dd</code> <code>\cdd</code>	$\leq \text{\u00FF}$	Escape-Sequenz mit hexadezimalen Ziffern $c \in \{0, \dots, 3\}, d \in \{0, \dots, 7\}$	
<code>\uxxxx</code>	$\leq \text{\uFFFF}$	Escape-Sequenz mit hexadezimalen Ziffern $x \in \{0, \dots, 9, a, \dots, f, A, \dots, F\}$	

Beispiele des Unicode-Zeichensatzes (Version 2.0), vgl. die Information des Unicode Consortiums in <http://www.unicode.org/charts/>:

Integer-Bereich	Unicode-Bereich	Zeichensatz
0 - 127	<code>\u0000</code> - <code>\u007F</code>	ASCII-Zeichensatz
128 - 255	<code>\u0080</code> - <code>\u00FF</code>	ISO-8859-Latin 1-Zeichensatz
880 - 1023	<code>\u0370</code> - <code>\u03FF</code>	griechische Zeichen
1024 - 1279	<code>\u0400</code> - <code>\u04FF</code>	kyrillische Zeichen
1424 - 1535	<code>\u0590</code> - <code>\u05FF</code>	hebräische Zeichen
1536 - 1791	<code>\u0600</code> - <code>\u06FF</code>	arabische Zeichen

deutsche Sonderzeichen:

Integer-Wert	Unicode-Zeichen	Bedeutung
228	<code>\u00E4</code>	deutscher Umlaut "ä"
246	<code>\u00F6</code>	deutscher Umlaut "ö"
252	<code>\u00FC</code>	deutscher Umlaut "ü"
223	<code>\u00DF</code>	deutsches scharfes S "ß"
196	<code>\u00C4</code>	deutscher Umlaut "Ä"
214	<code>\u00D6</code>	deutscher Umlaut "Ö"
220	<code>\u00DC</code>	deutscher Umlaut "Ü"

Beachte:

Nationale Sonderzeichen direkt ohne Unicode-Angabe in einem Java-Sourcecode sind nicht portabel (z.B. deutsche Umlaute mit DOS-Zeichensatz). Damit man jedoch nicht immer die umständlichen Unicode-Zeichen verwenden muß, kann man zunächst mit direkter Eingabe von Sonderzeichen im jeweiligen lokalen Zeichensatz arbeiten. Dann sollte man aber

Java bricht ab, wenn bei ganzzahliger Division oder Restbildung der Nenner 0 ist. Dagegen liefert Java einen der 3 Konstanten

```
Double.POSITIVE_INFINITY // z.B. als Wert von 3.0/0.0
Double.NEGATIVE_INFINITY // z.B. als Wert von -3.0/0.0
Double.NaN                // z.B. als Wert von 0.0/0.0,
```

die in der Klasse Double im Package java.lang definiert sind, wenn der Nenner 0 ist bei einer Fließkommadivision.

1.5.2 * Vergleichsoperatoren

Operator	Bezeichnung	Beispiel	Bemerkungen
<	kleiner	$x < y$	true, falls $x < y$, false sonst
<=	kleiner gleich	$x \leq y$	true, falls $x \leq y$, false sonst
>	größer	$x > y$	true, falls $x > y$, false sonst
>=	größer gleich	$x \geq y$	true, falls $x \geq y$, false sonst
==	gleich	$x == y$	true, falls x und y gleich sind, false sonst
!=	ungleich	$x != y$	true, falls x und y ungleich sind, false sonst

1.5.3 * logische Operatoren

Operator	Bezeichnung	Beispiel	Bemerkungen
&&	logisches Und	$x \&\& y$	true, falls x und y gleich true
	logisches Oder	$x \ \ y$	true, falls x oder y gleich true
!	logisches Nicht	$! x$	true, falls x gleich false

1.5.4 * Bitoperatoren

Operator	Bezeichnung	Beispiel	Bemerkungen/ Wert für jede Bitposition
&	bitweises Und	$x \& y$	1, falls zugeh. Bit von x und y eine 1 ist
	bitweises Oder	$x \ \ y$	1, falls zugeh. Bit von x oder y eine 1 ist
^	bitweises Exklusives Oder	$x \wedge y$	1, falls entweder zugeh. Bit von x oder das von y eine 1 ist
~	bitweises Nicht	$\sim x$	1, falls zugeh. Bit von x eine 0 ist
<<	Links-Shift	$x \ll y$	x wird um y Bitpositionen nach links geschoben
>>	(arithm.) Rechts-Shift	$x \gg y$	x wird um y Bitpositionen nach rechts geschoben, links wird mit höchstem Bit (Vorzeichenbit) aufgefüllt
>>>	(log.) Rechts-Shift	$x \ggg y$	x wird um y Bitpositionen nach rechts geschoben, links wird mit 0 aufgefüllt

1.5.5 * Zuweisungsoperatoren

Operator	Bezeichnung	Beispiel	Bemerkungen
=	Zuweisung	$x = y$	x wird der Wert des Ausdrucks y zugewiesen
$\langle \text{Operand} \rangle =$	zusammengesetzte Zuweisung	$x \langle \text{Operand} \rangle = y$	entspricht der Wertzuweisung $x = x \langle \text{Operand} \rangle (y)$ Der Operand ist ein binärer arithmet. oder ein Bit-Operator.

Beispiele:

```
int n = 3;
...
n += 2;    // entspricht: n = n + 2;
           // n hat danach den Wert 5
n *= 2;    // entspricht: n = n * 2;
           // n hat danach den Wert 10
n *= 2 + 3; // entspricht: n = n * (2 + 3);
           // und nicht: n = n*2 + 3;
           // n hat danach den Wert 50 (nicht 23!)
```

1.5.6 * sonstige Operatoren

Operator	Bezeichnung	Beispiel	Bemerkungen
$\langle \text{Datentyp} \rangle$	explizite Typumwandlung	$\langle \text{Datentyp} \rangle x$, z.B. $((\text{float}) 1)/3$	Wert von x wird in angegebenen Datentyp umgewandelt
?:	bedingte Bewertung	$x ? y : z$, z.B. $(x > y) ? x : y$	liefert den Wert von y , falls x gleich <code>true</code> ist, ansonsten den von z
.	Zugriff auf Klassenkomponente, Paket-/Klassenuntername	$\langle \text{Objektname} \rangle . \langle \text{Datenelement einer Klasse} \rangle$ $\langle \text{Paketname} \rangle . \langle \text{Unterpaketname} \rangle$	
instanceof	prüft auf Zuweisbarkeit von Objekten	$\langle \text{Objekt} \rangle \text{ instanceof } \langle \text{Klassenname} \rangle$	liefert <code>true</code> , falls das Objekt zuweisbar an ein anderes Objekt der Klasse ist
new	erzeugt neues Objekt	<code>new</code> $\langle \text{Konstruktoraufwurf einer Klasse} \rangle$	liefert neu erzeugtes Objekt einer Klasse

1.5.7 * Vorrang und Assoziativität von Operatoren

Prioritätsstufe	Operator	Bedeutung	Auswertungsreihenfolge
14	$\underbrace{++, --}_{\text{Postfix}}, [], \underbrace{()}_{\text{Funktionsaufruf}}$	Erhöhung/Verminderung von Variablen, Zugriff auf Objektkomponente, Funktionsaufruf	von links
14	$!, \sim, \underbrace{++, --}_{\text{Präfix}}, \underbrace{+, -}_{\text{Vorzeichen}}$	Verneinung (logisch, Bit-), Erhöhung/Verminderung von Variablen	von rechts
13	<code>new, (<Datentyp>)</code>	Erzeugung von Objekten, Typumwandlung	von rechts
12	<code>*, /, %</code>	Multiplikation, Division, Restbildung	von links
11	$\underbrace{+, -}_{\text{Addition, Subtraktion}}$	Addition, Subtraktion	von links
10	<code><<, >>, >>></code>	Bitshifts nach links und rechts	von links
9	<code><, <=, >, >=, instanceof</code>	Vergleiche (kleiner, kleiner gleich, ...), Zuweistest von Objekt	von links
8	<code>==, !=</code>	Gleichheit, Ungleichheit	von links
7	<code>&</code>	bitweises Und	von links
6	<code>^</code>	bitweises Exklusives Oder	von links
5	<code> </code>	bitweises Oder	von links
4	<code>&&</code>	logisches Und	von links
3	<code> </code>	logisches Oder	von links
2	<code>?:</code>	bedingte Bewertung	von rechts
1	<code>=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, =</code>	Wertzuweisung, Zuweisung verknüpft mit Operator	von rechts

Dabei steht die Prioritätsstufe 15 als höchste Priorität und 1 für die niedrigste Priorität bei der Auswertung von Termen.

1.6 * Anweisungen

1.6.1 Blockanweisung

Blockanweisungen

1.6.2 * bedingte Anweisungen (if, switch)

Beispiele: vgl. "IfTest_1.java" und "IfTest_2.java"

	Anweisungsname	Beispiel	Bedeutung
a)	if-Anweisung	<pre>double x; // double-Zufallszahl zwischen 0.0 und kleiner als 1 x = Math.random(); if (x > 0.0) System.out.println("x ist positiv!");</pre>	<p>nur wenn x größer als 0.0 ist, wird der String "x ist positiv!" ausgegeben</p>
b)	if-else-Anweisung	<pre>double erg, x; // double-Zufallszahl zwischen -0.5 und kleiner als 0.5 x = Math.random() - 0.5; if (x >= 0.0) erg = x; else erg = -x;</pre> <p>kürzer:</p> <pre>erg = (x >= 0.0) ? x : -x;</pre>	<p>Der Betrag der Zahl x wird in erg gespeichert. Wenn x größer als 0.0 ist, wird in erg der Wert x gespeichert, ansonsten (für negative x) wird der Wert -x, also die positive Zahl, abgespeichert.</p>
c)	if-else if-else-Anweisung	<pre>// Achtung! Keine Unicode-Zeichen verwendet werden! String s = "rgerlich"; // 1. Buchstabe 'r' von String s abspeichern in c char c = s.charAt(0); if (c == 'ä') System.out.println("String beginnt mit Umlaut ä"); else if (c == 'ö') System.out.println("String beginnt mit Umlaut ö"); else if (c == 'ü') System.out.println("String beginnt mit Umlaut ü"); else if (c == 'A') System.out.println("String beginnt mit Umlaut Ä"); else if (c == 'O') System.out.println("String beginnt mit Umlaut Ö"); else if (c == 'U') System.out.println("String beginnt mit Umlaut Ü"); else System.out.println("String beginnt ohne Umlaut."); // Leerzeile ausgeben System.out.println();</pre>	<p>falls in der char-Variable c der Buchstabe ä steht, wird der String "String beginnt mit Umlaut ä" ausgegeben und anschließend übersprungen und die Variable c mit dem nächsten Zeichen von s befüllt und die Anweisung nach dem nächsten if-else if ausgegeben.</p> <p>Falls in ihr der Buchstabe ö steht, wird der Test in der 1. if-Anweisung mit false und es wird zum nächsten else if übergegangen. Dieser liefert true, es wird der String "String beginnt mit Umlaut ö" ausgegeben und diese Anweisung übersprungen und die Variable c mit dem nächsten Zeichen von s befüllt und die Anweisung nach dem 1. else if ausgegeben. ... und wieder erfolgt dies bis zur nächsten Anweisung übergegangen. Wenn der Buchstabe ü steht, wird der String "String beginnt mit Umlaut ü" ausgegeben und diese Anweisung übersprungen und die Variable c mit dem nächsten Zeichen von s befüllt und die Anweisung nach dem letzten else if ausgegeben. (und dann die Leerzeile ausgegeben).</p> <p>Übersichtlicher wäre da ein</p>

Hinweise:

- log. Ausdruck muß vom Typ `boolean` sein
- log. Ausdruck muß geklammert sein
- Vor `else` muß die Anweisung (Ausnahme: Blockanweisung) mit `”;` beendet werden.
- Ein `else`-Zweig wird bei Mehrdeutigkeit der direkt vorstehenden `if`-Anweisung zugeordnet.

Beispiel:

```
if (x >= 0.0)
    if (x > 0.0)
        System.out.println("x gr\u00F6\u00DFer als Null!");
    else
        System.out.println("x gleich Null!");
```

Bei anderer gewünschter Zuordnung muß eine leere Anweisung verwendet oder geklammert werden.

Beispiel:

```
if (x >= 0.0)
    if (x > 0.0)
        System.out.println("x gr\u00F6\u00DFer als Null!");
    else
        ;
else
    System.out.println("x kleiner als Null!");
```

oder

```
if (x >= 0.0)
{
    if (x > 0.0)
        System.out.println("x gr\u00F6\u00DFer als Null!");
}
else
    System.out.println("x kleiner als Null!");
```

*** switch-Anweisung**

Beispiel: vgl. `if-else` `if-else`-Anweisung

```

// Achtung! Keine Unicode-Zeichen verwendet worden! Nicht portabel!

String s = "    ";
// 1. Buchstabe ' ' von String s abspeichern in c
char c = s.charAt(0);

switch(c)
{
    case 'ä':
        System.out.println("String beginnt mit Umlaut ae");
        break;
    case 'ö':
        System.out.println("String beginnt mit Umlaut oe");
        break;
    case 'ü':
        System.out.println("String beginnt mit Umlaut ue");
        break;
    case 'Ä':
        System.out.println("String beginnt mit Umlaut Ae");
        break;
    case 'Ö':
        System.out.println("String beginnt mit Umlaut Oe");
        break;
    case 'Ü':
        System.out.println("String beginnt mit Umlaut Ue");
        break;
    default:
        System.out.println("String beginnt ohne Umlaut!");
}

// Leerzeile ausgeben
System.out.println();

```

Da der Buchstabe in `c` ein `'ä'` ist ("`s.charAt(0)`" liefert 1. Buchstaben von `s`), wählt die `switch`-Anweisung (Argument ist der Wert `c` selbst) die Marke "`case 'ä':`" an als Sprungziel und führt die Anweisung zur Ausgabe des Umlauts `ae` aus. Die nachfolgende `break`-Anweisung verhindert, dass alle darauffolgenden Anweisungen (mit anderen `case`-Marken auch noch ausgeführt werden). Es springt zum Ende der `switch`-Anweisung und daher zur Ausgabe der Leerzeile.

Wäre `c` durch die Zuweisung "`c = s.charAt(4);`" initialisiert worden (also mit `'ö'`), würde die `switch`-Anweisung die Marken "`case 'ö':`" und "`case 'ü':`" überspringen und direkt zu "`case 'ä':`" verzweigen. Dann würde die Anweisung zur Ausgabe des Umlauts `ue` ausgegeben werden. Das nachfolgende `break` verhindert wie oben die anderen folgenden Ausgaben. Wäre `c` durch die Zuweisung "`c = s.charAt(1);`" initialisiert worden (also mit dem Blankzeichen `' '`), würde die `switch`-Anweisung keine `case`-Marke finden mit dem Wert von `c`, deshalb wird die `default`-Marke angesprungen. Würde sie fehlen, würde gar nichts ausge-

geben und sofort die anschließende Leerzeile ausgegeben werden.

Beispiel: vgl. "SwitchRoman.java"

Abhängig vom Wert `c` wird zu einer der `case`-Marken verzweigt und alle folgenden Anweisungen ausgeführt (auch die mit anderen `case`-Marken), bis das Ende der `switch`-Anweisung oder eine `break`-Anweisung erreicht ist. Danach werden die auf die `switch`-Anweisung folgenden Anweisungen ausgeführt. Kommt es zu keiner Übereinstimmung, wird der `switch`-Block ignoriert.

```
// 2 Methoden in einer Appletklasse

public void init()
{
    // roemische Zahl 67
    String s = "LXVII";
    int dez_zahl;

    // klappt nur, da keine Zahlen der Form IX oder IV oder ... vorkommen

    // s.charAt(0) == 'L', 50 ist der Funktionswert
    dez_zahl = roman(s.charAt(0));

    // s.charAt(1) == 'X', 10 ist der Funktionswert,
    // der zu dez_zahl addiert wird
    dez_zahl = dez_zahl + roman(s.charAt(1));
    dez_zahl = dez_zahl + roman(s.charAt(2));
    dez_zahl = dez_zahl + roman(s.charAt(3));
    dez_zahl = dez_zahl + roman(s.charAt(4));
}

public int roman(char c)
{
    int erg;

    switch(c)
    {
        case 'I': erg = 1;
                 break;
        case 'V': erg = 5;
                 break;
        case 'X': erg = 10;
                 break;
        case 'L': erg = 50;
                 break;
        case 'C': erg = 100;
                 break;
        case 'D': erg = 500;
    }
}
```

```

        break;
    case 'M': erg = 1000;
        break;
}

return erg;
}

```

Beispiel: vgl. "SwitchCountdown.java"

Hier wird ausgenutzt, daß die `case`-Marke nur ein Startpunkt für die weitere Ausführung ist. Da kein `break` verwendet worden ist, werden *alle* Anweisung in der `switch`-Anweisung nach der gefundenen `case`-Marke ausgeführt bis zur `break`-Anweisung für den Wert 0.

// 2 Methoden in einer Appletklasse

```

public void init()
{
    // double-Zufallszahl zwischen 0.0 und kleiner als 11.0 wird erzeugt
    double zufall = Math.random()*11.0;

    // durch Abschneiden der Dezimalstellen wird eine int-Zufallszahl
    // zwischen 0 und 10 (einschliesslich!) erzeugt
    int sek = (int) zufall;

    // ist der Wert von sek z.B. 8,
    // wird zuerst "8..", dann "7..", dann ... und dann "1.." und "take off!"
    // ausgegeben
    // da nicht println(..), sondern print(..) verwendet wurde, erscheint
    // alles in einer Zeile als Ausgabe des Strings
    // "8..7..6..5..4..3..2..1..take off!"

    // ist der Wert von sek dagegen z.B. 3, wird der String
    // "3..2..1..take off!" ausgegeben

    countdown(sek);
}

public void countdown(int secs)
{
    System.out.print("Countdown: ");
    switch(secs)
    {
        case 10: System.out.print("10..");
        case 9: System.out.print("9..");
        case 8: System.out.print("8..");
        case 7: System.out.print("7..");
    }
}

```

```

        case 6: System.out.print("6..");
        case 5: System.out.print("5..");
        case 4: System.out.print("4..");
        case 3: System.out.print("3..");
        case 2: System.out.print("2..");
        case 1: System.out.print("1..");
        case 0: System.out.println("take off!");
                break;
        default: System.out.println("cancelled!");
    }
}

```

Hinweise:

- Testausdruck für `switch` muß vom Typ `byte`, `char`, `short` oder `int` sein. `boolean`, `long` oder `double` ... sind nicht erlaubt.
- Testausdruck für `switch` muß geklammert sein.
- Die Ausführung bei einer Verzweigung zu einer `case`-Marke endet nur vor der nächsten, falls eine `break`-Anweisung kommt.
- Mehrere `case`-Marken hintereinander sind erlaubt, um denselben Satz an Anweisungen auszuführen.
- Die Marke `default` wird angesprungen, wenn keine Übereinstimmung mit den anderen `case`-Marken erfolgt ist.

Beispiel: vgl. "SwitchJahreszeit.java"

```

// 2 Methoden in einer Appletklasse

public void init()
{
    // double-Zufallszahl zwischen 1.0 und kleiner als 13.0 wird erzeugt
    double zufall = Math.random()*12.0 + 1.0;

    // int-Zufallszahl liegt zwischen 1 und 12 wird erzeugt
    int monat = (int) zufall;

    String name = jahreszeit(monat);
    System.out.println("Der Monat " + monat + " geh\u00F6rt zur Jahreszeit "
        + name);
}

public String jahreszeit(int monat)
{
    String name;

```

```

switch(monat)
{
    case 3:
    case 4:
    case 5:
        name = "Fr\u00fchling";
        break;

    case 6:
    case 7:
    case 8:
        name = "Sommer";
        break;

    case 9:
    case 10:
    case 11:
        name = "Herbst";
        break;

    case 12:
    case 1:
    case 2:
        name = "Winter";
        break;

    default:
        name = "nicht zu entscheiden, welche Jahreszeit es ist";
}
return name;

```

Hier wird ausgenutzt, dass *eine* Anweisung (z.B. die Anweisung 'name = "Sommer";' mit *mehreren* case-Marken versehen werden kann. Im Fall, dass *monat* *einer* der Werte 6, 7 oder 8 ist, wird der String *name* auf "Sommer" gesetzt. Wie man bei den hintereinander geschriebenen case-Marken zu den Werten 12, 1 und 2 sieht, muss keine Reihenfolge eingehalten werden.

1.6.3 * Wiederholungsanweisungen

* while-Schleife

Die *while*-Anweisung wiederholt solange die Anweisung, wie der Testausdruck den Wert *true* beinhaltet.

Beispiel: Die *while*-Schleife wird solange durchlaufen, bis die Zufallszahl in *n* größer oder gleich 25 wird. In diesem Fall wird keine weitere Zufallszahl bestimmt (d.h. der *while*-Block mit der Ausgabe einer nächsten Zufallszahl nicht mehr ausgeführt).

```

int n = 0;
double zufall;

while (n < 25)

```

```

{
    zufall = Math.random()*31.0;
    n = (int) zufall;
    System.out.println(" Zufallszahl zwischen 0 und 30 ist " + n);
}

```

Beispiel: Die `while`-Schleife wird solange durchlaufen, bis die Zufallszahl in `zufall` kleiner oder gleich 0.3 wird. In diesem Fall wird keine weitere Zufallszahl ausgegeben (d.h. der `while`-Block mit der Ausgabe der Zufallszahl und die Bestimmung einer neuen Zufallszahl wird nicht mehr ausgeführt). U.U. wird auch die `while`-Schleife überhaupt nicht durchlaufen, wenn die 1. Zufallszahl schon kleiner oder gleich 0.3 ausfällt.

```

double zufall;

// Zufallszahl zwischen 0.0 und kleiner als 1.0
zufall = Math.random();

while (zufall > 0.3)
{
    System.out.println(" Zufallszahl zwischen 0.0 und kleiner als 1.0 ist "
        + zufall);
    zufall = Math.random();
}

```

Beispiel: Es wird eine ganze Zeile von der Tastatur eingelesen und der logische Wert `weiter` auf `true` gesetzt, wenn diese Zeile mit dem Buchstaben "j" als String übereinstimmt (ansonsten auf `false`). Es wird solange wiederholt, bis der Anwender einmal nicht "j" eingibt in der Zeile. Die logische Variable `weiter` muss nicht vor der `do-while`-Schleife initialisiert worden sein (passiert im `do-while`-Block durch die Anweisung `'weiter = "j".equals(antw);'`).

```

boolean weiter = true;
String antw;
BufferedReader is = new BufferedReader(
    new InputStreamReader(System.in));

while (weiter)
{
    ...
    System.out.print("Weiter (j/n): ");
    antw = is.readLine();           // Einlesen einer Zeile
    weiter = "j".equals(antw);     // Vergleich von "j" und antw
}

```

Hinweise:

- Der Testausdruck muß vom Typ `boolean` sein.

- Der Testausdruck muß geklammert werden.
- Alle Variablen im Testausdruck der `while`-Schleife müssen vor den ersten Auswerten initialisiert sind.
- Der Testausdruck der `while`-Schleife wird mindestens einmal ausgewertet, die Anweisungen des Schleifenrumpfes können ggf. auch gar nicht ausgeführt werden.
- Durch den Testausdruck wird ggf. eine Anweisung wiederholt, für die Wiederholung mehrerer Anweisungen muß die Blockanweisung verwendet werden.

* `do-while`-Schleife

Die `do-while`-Anweisung wiederholt mindestens einmal die Anweisung und ggf. solange, wie der Testausdruck den Wert `true` besitzt.

Beispiel: Die `do-while`-Schleife wird solange durchlaufen, bis die Zufallszahl in `n` größer oder gleich 25 wird. In diesem Fall wird keine weitere Zufallszahl bestimmt (d.h. der `do-while`-Block mit der Ausgabe einer nächsten Zufallszahl nicht mehr ausgeführt). Die `do-while`-Schleife wird immer mindestens einmal durchlaufen!

```
// n muss nicht wegen do-while initialisiert werden
// (dies geschieht innerhalb des do-while-Blocks)
int n;

double zufall;

do
{
    zufall = Math.random()*31.0;
    n = (int) zufall;
    System.out.println(" Zufallszahl zwischen 0 und 30 ist " + n);
}
while (n < 25);
```

Beispiel: Die `do-while`-Schleife wird solange durchlaufen, bis die Zufallszahl in `zufall` kleiner oder gleich 0.3 wird. In diesem Fall wird keine weitere Zufallszahl ausgegeben (d.h. der `do-while`-Block mit der Ausgabe der Zufallszahl und die Bestimmung einer neuen Zufallszahl wird nicht mehr ausgeführt). Auch wenn die 1. Zufallszahl schon kleiner oder gleich 0.3 ausfällt, wird die `do-while`-Schleife einmal durchlaufen.

```
double zufall;

do
{
    zufall = Math.random();
    System.out.println(" Zufallszahl zwischen 0.0 und kleiner als 1.0 ist "
        + zufall);
}
```

```
while (zufall > 0.3);
```

Beispiel: Es wird eine ganze Zeile von der Tastatur eingelesen und der logische Wert `weiter` auf `true` gesetzt, wenn diese Zeile mit dem Buchstaben "j" als String übereinstimmt (ansonsten auf `false`). Es wird solange wiederholt, bis der Anwender "j" eingibt in der Zeile. Die logische Variable `weiter` muss vor der `while`-Schleife durch "`boolean weiter = true;`" initialisiert worden sein.

```
boolean weiter;
String antw;
BufferedReader is = new BufferedReader(
    new InputStreamReader(System.in));

do
{
    ...
    System.out.print("Weiter (j/n): ");
    antw = is.readLine();           // Einlesen einer Zeile
    weiter = "j".equals(antw);     // Vergleich von "j" und antw
}
while (weiter);
```

Hinweise:

- Der Testausdruck muß vom Typ `boolean` sein.
- Der Testausdruck muß geklammert werden.
- Die Variablen im Testausdruck der `do-while`-Schleife können auch im Schleifenrumpf noch initialisiert werden.
- Sowohl der Testausdruck der `do-while`-Schleife wird mindestens einmal ausgewertet, als auch die Anweisungen des Schleifenrumpfes.
- Durch den Testausdruck wird ggf. eine Anweisung wiederholt, für die Wiederholung mehrerer Anweisungen muß die Blockanweisung verwendet werden.

Eine `repeat-until`-Schleife gibt es nicht, sie muß mit der `do-while`-Schleife emuliert werden (logische Verneinung des Testausdrucks).

Beispiel:

```
integer i;           int i;
...                 ...
repeat              do
    ...             {
                    ...
    i = i + 1        i++;
                    }
until (i > 10);     while (i <= 10);
```

* for-Schleife

Die `for`-Schleife besteht aus 4 Teilen, dem Initialisierungsausdruck, der Abbruchbedingung vom Typ `boolean`, dem Schleifenrumpf (oder Schleifenblock) und dem Updateausdruck. Die Teile werden durch `;` voneinander getrennt und können auch leer sein (vgl. `For_1.java`). Ein leerer Testausdruck wird wie der Wert `true` behandelt.

Beispiel: die Schleifenvariable `i` durchläuft die Werte 0, 1, 2, ..., 10
Gestartet wird mit dem Wert 0, dann kommt der Test `"i <= 10"`, der für `i` gleich 0 `true` liefert, die Ausgabe des Wertes von `i` wird ausgeführt und dann `i` durch `i++` um 1 erhöht. Danach wird für `i` gleich 1 wieder der Test `"i <= 10"` durchgeführt, wieder der Schleifenblock (hier nur eine Ausgabeanweisung), `i` erhöht, Dies geht solange gut, bis `i` durch die Erhöhung mittels `i++` den Wert 11 erhält. Der Test `"i <= 10"` liefert `false` und die Schleife wird ohne Beachtung des Schleifenblocks (d.h. ohne weitere Ausgabe) beendet.

```
int i;

for (i = 0; i <= 10; i++)
    System.out.println(" i = " + i);
```

Beispiel: die Schleifenvariable `i` durchläuft die Werte 10, 9, 8, ..., 0.
Gestartet wird mit dem Wert 10, dann kommt der Test `"i >= 0"`, der für `i` gleich 10 `true` liefert, die Ausgabe des Wertes von `i` wird ausgeführt und dann `i` durch `i--` um 1 erniedrigt. Danach wird für `i` gleich 9 wieder der Test `"i >= 0"` durchgeführt, wieder der Schleifenblock (hier nur eine Ausgabeanweisung), `i` erniedrigt, Dies geht solange gut, bis `i` durch die Subtraktion von 1 mittels `i--` den Wert -1 erhält. Der Test `"i >= 0"` liefert `false` und die Schleife wird ohne Beachtung des Schleifenblocks (d.h. ohne weitere Ausgabe) beendet.

```
int i;

for (i = 10; i >= 0; i--)
    System.out.println(" i = " + i);
```

Beispiel: die Schleifenvariable `i` durchläuft die Werte 0, 2, 4, ..., 10
Wie das 1. Beispiel zu `for`, nur wird hier die Update-Anweisung `i++` durch `i=i+2` ersetzt. Deshalb werden die ungeraden Zahlen nicht ausgegeben und durch diesen Schleifendurchlauf übersprungen.

```
int i;

for (i = 0; i <= 10; i=i+2)
    System.out.println(" i = " + i);
```

Hinweise:

- Der Durchlaufsinne einer `for`-Schleife ist beliebig, da die Ausdrücke beliebig sind.

- Die Schleifenvariable muß keinen ganzzahligen Datentyp besitzen (vgl. "For_2.java").

```
double x, h = 0.01;

for (x = 0.0; x <= 1.0; x += h)
    ...
```

Vorsicht bei der Verwendung von Gleitpunktvariablen in `for`-Schleifen, da Rundungsfehler auftauchen können und daher die `for`-Schleife einmal zu oft oder zu wenig durchlaufen werden kann. In obigem Beispiel sollte man besser schreiben:

```
double x, h = 0.01;

for (x = 0.0; x <= 1.0 + 1E-13; x += h)
    ...
```

- Der Schleifenrumpf von `for` wird nicht durchlaufen, wenn der Testausdruck sofort den Wert `false` liefert. Die Initialisierung wird aber in jedem Fall durchgeführt.

```
int i, i_max = 10;

for (i = 20; i <= i_max; i++)
    ...
```

Nur die Initialisierung `i = 20` wird ausgeführt, der Schleifenrumpf nicht.

- Die Schleifenvariable behält nach Abarbeitung der `for`-Schleife ihren durch die letzte Update-Ausdruck erhaltenen Wert.

```
int i;

for (i = 0; i <= 10; i++)
    ...
```

`i` hat den Wert 11 nach Abarbeitung der Schleife.

- Schleifen mit „ungewöhnlichen“ Datentypen sind möglich (vgl. "For_3.java"):

```
char c;

for (c = 'a'; c <= 'z'; c++)
    ...
```

Hier wird der zugehörige ASCII-Wert von `c` erhöht, bis der ASCII-Wert von `'z'` erreicht wird. Damit werden alle Kleinbuchstaben durchlaufen.

```
String wort;  
System.out.println("");  
for (wort = "immer weniger"; !wort.equals("");  
     wort = wort.substring(0, wort.length()-1))
```

Hier wird die Ausgabe des Strings `wort` stets um ein Zeichen verkürzt, die Schleifenvariable ist hier ein String.

- In einer `for`-Schleife kann man mehrere Initialisierungen oder Updates vornehmen, die durch Kommata voneinander getrennt werden (vgl. `For_3.java`):

```
int anz, sum, sum2, zahl;  
  
for (sum = sum2 = 0, anz = zahl = 0; zahl <= 10; zahl+=2, anz++)  
{  
    sum += zahl;  
    sum2 += zahl*zahl  
}
```

Die Variablen `sum` und `sum2` für die Summen werden ebenfalls wie die Durchlaufzähler `anz` und die Schleifenvariable `zahl` auf 0 initialisiert, beim Update wird `anz` um 1 und `zahl` um 2 erhöht. Damit enthält `sum` die Summe der geraden Zahlen von 0 bis 10, `sum2` die Summe der Quadrate der geraden Zahlen von 0 bis 10 und `anz` die Anzahl dieser geraden Zahlen.

- Jede `while`- oder `do while`-Schleife kann durch eine `for`-Schleife realisiert werden.
- In einer `for`-Schleife kann man nur dort definierte Variablen deklarieren und initialisieren (vgl. `ForVar_1.java`).

1.6.4 * Sprunganweisungen

* `break`-Anweisung

Die `break`-Anweisung bewirkt das unmittelbare Verlassen des `switch`-Blockes oder der `for`-, `while`-, `do-while`-Schleife. Dabei wird stets nur der innerste Block/Schleifenrumpf verlassen. Vgl. `Break_1.java` und `Break_2.java`.

Die `break`-Anweisung mit einem Label kann auch einen mit diesem Label markierten Block verlassen, falls dieser die `break`-Anweisung umfaßt, so daß auch ein gleichzeitiges Verlassen mehrerer Blöcke möglich ist (vgl. `Break_3.java` und `Break_4.java`).

* continue-Anweisung

Die `continue`-Anweisung bewirkt den unmittelbaren Sprung zum Ende des Schleifenrumpfes, jedoch nicht unbedingt das Verlassen der Schleife. Der normale Ablauf der Schleife wird durch `continue` in dem Sinn geändert, daß einige Anweisungen im Rumpf übersprungen werden. Dabei werden stets nur Anweisungen im innersten Block/Schleifenrumpf übersprungen. Sie kann nur in der `for`-, `while`- oder `do-while`-Schleife auftreten, vgl. `Continue_1.java` und `Continue_2.java`. Die `continue`-Anweisung mit einem Label kann auch am Ende eines mit diesem Label markierten Schleife mit der Ausführung fortfahren, falls diese die `continue`-Anweisung umfaßt, so daß auch ein gleichzeitiges Überspringen von Anweisungen mehrerer Blöcke möglich ist (vgl. `Continue_3.java`).

* return-Anweisung

Eine `return`-Anweisung bewirkt die unmittelbare Beendigung der momentanen Funktion und einen Rücksprung zum aufrufenden Programmteil. Wird eine `return`-Anweisung mit einem Ausdruck angegeben, wird der Wert dieses Aufrufs als Funktionsergebnis an den aufrufenden Programmteil zurückgegeben. Sollte der Wert des Ausdrucks nicht mit dem Datentyp des Funktionswertes übereinstimmen, erfolgt eine implizite Typumwandlung. Das Erreichen des Endes der Funktion ist gleichbedeutend mit einer `return`-Anweisung ohne Ausdruck.

Hinweise:

- Die `main()`-Methode in der Applikation hat keinen Rückgabewert. Es kann jedoch mit der Methode `exit()`-Methode der Klasse `System` und einem `int`-Argument der Rückgabewert an das Betriebssystem spezifiziert werden. 0 steht dabei für ordnungsgemäßen Ablauf, ein Wert ungleich 0 für einen Ablauf mit einer Fehlersituation.
- Es können durchaus mehrere `return`-Anweisungen in einer Funktion stehen (z.B. in mehreren `if`-Alternativen), vgl. `Return_1.java`.

1.7 * Arrays

Ein *Array* (= *Feld*, *Vektor*) gehört zu den strukturierten/zusammengesetzten Datentypen.

```
double [] umsatz = { 1000.0, 2000.0, 1500.0, 2500.0 };

// schlechtere Form
int wuerfel_zahlen [] = { 1, 2, 3, 4, 5, 6 };

// bessere Form
int [] wuerfel_zahlen = { 1, 2, 3, 4, 5, 6 };

// schlechtere Form, da wurf eine int-Variable und kein int-Array
int wuerfel2 [], wurf;

// bessere Form, da umsaetze_sun und umsaetze_microsoft Arrays
double [] umsaetze_sun, umsaetze_microsoft;
```

Das Array mit dem Namen `umsatz` besteht aus vier `double`-Werten. Ein eindimensionales Array kann man als Vektor von Werten eines *gemeinsamen* Datentyps (hier: `double` bzw. `int`) ansehen. Die Werte (= Komponenten des Arrays) werden unter einem gemeinsamen Namen mit Indizes angesprochen. Die Komponenten des Arrays liegen unmittelbar aufeinanderfolgend im Speicher.

```
System.out.println("1. Komponente: " + umsatz[0]);
System.out.println("2. Komponente: " + umsatz[1]);
System.out.println("3. Komponente: " + umsatz[2]);
System.out.println("4. Komponente: " + umsatz[3]);
```

Im Gegensatz zu anderen Sprachen kann man in Java die Länge eines Arrays ermitteln:

```
System.out.println("L\u00E4nge des Arrays: " + umsatz.length);
```

Dies kann man also gut mit einer `for`-Schleife verbinden, um alle Komponenten zu initialisieren, auszugeben, zu ändern,

```
for (int i = 0; i < umsatz.length; i++)
{
    umsatz[i] = i*1000.0 + 2500.0;
    System.out.println(i + ". Komponente = " + umsatz[i]);
}
```

Der Datentyp der Komponenten kann jeder in Java bekannte Datentyp sein:

- elementare Datentypen (arithmetische wie z.B. `int`, `char`, `double`)
- (selbst wieder) Arrays
- Strukturen
- Zeiger
- Klassen

Ein Array ist also eine Zeile mit (mehreren) Spalten (= Komponenten). Die Anzahl der Komponenten des Arrays wird als Dimension bzw. Länge des Arrays bezeichnet. Standardmäßig beginnt ein Array bei dem Index 0 und hat als letzten Index die um 1 verminderte Länge des Arrays. Es kann keine andere Numerierung vorgegeben werden (z.B. startend beim Index 1).

Ein Array muß wie eine Variable auch in dem Deklarationsteil definiert werden, eine Längenangabe darf dabei nicht erfolgen. Ein Array kann entweder durch Aufzählung aller Komponenten

```
int [] wuerfel_zahlen = { 1, 2, 3, 4, 5, 6 };
```

oder dynamisch

```
int [] wuerfel_zahlen = new int[6];
for (int i = 0; i < wuerfel_zahlen.length; i++)
    wuerfel_zahlen[i] = i + 1;
```

(d.h. die Länge des Arrays muss nicht beim Compilieren feststehen) erzeugt werden.

Beachte: Java verbietet den Zugriff auf gar nicht vorhandene Komponenten, so daß in obigem Beispiel ein Zugriff auf `umsatz[-1]` bzw. `umsatz[4]` eine Fehlermeldung nach sich zieht. Diese Fehlermeldung meldet sich als

```
java.lang.ArrayIndexOutOfBoundsException: 4
    at ArrayTestProg.main(ArrayTestProg.java:67)
```

4 ist dabei der falsche Index, der zum Abbruch des Java-Programmes geführt hat. Nebenbei werden der Methodename (hier: `main()`) und die Zeile im Sourcefile (hier: Zeile 67 im File `ArrayTestProg.java`) mit genannt.

Arrays werden in Anwendung gebraucht zur Abspeicherung

- aller Iterierten einer Iteration (Folgliedern)
- von Zahlen für mehrere Tage/Monate/Jahre
- mehrerer Zeichen (Strings) oder mehrerer Zeilen von Strings
- von Vektoren oder Matrizen
- von Häufigkeiten mehrerer gleichartiger Daten
- von gleichartigen Datensätzen (Datenbanken, Arrays von Strukturen)
- von gleichartigen GUI-Elementen (alle Buttons, alle Labels, ...) eines Java-Applets
- ...

Arrays müssen vor ihrer Verwendung definiert werden. Alle Arraykomponenten werden von Java automatisch beim Anlegen des Arrays mit dem Standardwert ihres Komponentendatentyps initialisiert. Die Komponenten können wie normale Variablen des Komponentendatentyps verwendet werden.

Intern speichert Java für Arrays nur die Anfangsadresse der Komponenten ab. Aus Sicherheitsgründen kann man diese Adresse jedoch nicht abfragen. Man spricht davon, dass Arrays in Java als Referenzen behandelt werden (Referenz = interner Hinweis, wo die 1. Komponente abgespeichert ist).

```
double [] umsatz = { 1000.0, 2000.0, 1500.0, 2500.0 };

// In umsatz wird die Anfangsadresse von umsatz[0] mit Wert 1000.0
// abgespeichert.
```

```
int [] wuerfel_zahlen = new int[6];

// In wuerfel_zahlen wird die Anfangsadresse des neu angelegten int-Arrays
// der Laenge 6 abgespeichert.
```

Arrays können im Laufe des Programmes ihre Länge ändern, die Arraylänge steckt also nicht im Datentyp der Arrays.

```
// Array hat Laenge 3
int [] zahlen = { 1, 2, 3 };
for (int i = 0; i < zahlen.length; i++)
    System.out.println(i + ". Zahl = " + zahlen[i]);
System.out.println();

// Array hat Laenge 5
zahlen = new int[5];
for (int i = 0; i < zahlen.length; i++)
    System.out.println(i + ". Zahl = " + zahlen[i]);
System.out.println();

// Array hat Laenge 10
zahlen = new int[10];
for (int i = 0; i < zahlen.length; i++)
{
    zahlen[i] = i*10;
    System.out.println(i + ". Zahl = " + zahlen[i]);
}

// Array hat Laenge 2
zahlen = new int[2];
for (int i = 0; i < zahlen.length; i++)
{
    zahlen[i] = i+1;
    System.out.println(i + ". Zahl = " + zahlen[i]);
}
System.out.println();
```

Arrays selbst werden – falls man keine Initialisierung angibt – mit der ungültigen Adresse null initialisiert.

```
// Fehler: Compiler beschwert sich, dass tage_im_monat evtl. uninitialized
// int [] tage_im_monat;
int [] tage_im_monat = null;

// tage_im_monat wird mit null initialisiert
```

```

if (tage_im_monat == null)
    System.out.println("Array zeigt auf Nulladresse!");
else
    System.out.println("Java-Version ist katastrophal!");

// Fehler: Array hat keine Laenge!
System.out.println("L\u00E4nge des Arrays: " + tage_im_monat.length);

// Fehler: Array hat keine Komponenten!
System.out.println("1. Komponente: " + tage_im_monat[0]);

```

Arrays kann man weder (sinnvoll) vergleichen noch (einfach) kopieren, addieren, subtrahieren,

```

double [] umsatz = { 1000.0, 2000.0, 1500.0, 2500.0 };
double [] kopie;

// keine gute Kopie
kopie = umsatz;

// wuerde auch umsatz[0] aendern (!!!)
// kopie[0] = -50.99;

// richtig:
kopie = new double[umsatz.length];
System.arraycopy(umsatz, 0, kopie, 0, umsatz.length);

// aendert umsatz[0] nicht!!!
kopie[0] = -50.99;

```

richtiger/falscher Vergleich von Arrays:

```

double [] umsatz = { 1000.0, 2000.0, 1500.0, 2500.0 };
double [] kopie;

// keine gute Kopie
kopie = umsatz;

if (kopie == umsatz)
    System.out.println("PROBLEM! Referenzwerte sind gleich, damit auch die Arrays")
else
    // darf nicht ausgegeben werden, da Referenzen gleich
    System.out.println("GUT! Referenzwerte sind verschieden, evtl. die Arrays gleich");

// gute Kopie, neuen Speicherplatz anlegen, einzeln kopieren
kopie = new double[umsatz.length];

```

```

System.arraycopy(umsatz, 0, kopie, 0, umsatz.length);

if (kopie == umsatz)
    // darf nicht ausgegeben werden, da Referenzen verschieden sein muessen
    System.out.println("PROBLEM! Referenzwerte sind gleich, damit auch die Arrays")
else
    System.out.println("GUT! Referenzwerte sind verschieden, die Arrays sind evtl.

```

Es können auch höherdimensionale Arrays (z.B. Werte in einer Tabelle mit Zeilen und Spalten) angelegt werden (siehe Übungen). Dabei kann man beim Erzeugen des Arrays entweder beide Dimensionen sofort angeben (bei der Tabelle: Anzahl der Zeilen und der Spalten) oder erst einmal nur die Anzahl der Zeilen festlegen und die Anzahl der Spalten pro Zeile noch offenlassen. Da eine Zeile einer Tabelle dann ein eindimensionales Array ist, kann man später die Anzahl der Werte pro Zeile nachträglich festlegen (siehe Übung). Auf die Komponenten des höherdimensionale Arrays kann man durch mehrere Indizes zugreifen.

```

// zweidim. Array von int-Werten definieren
int [][] zahlen_in_tabelle;

// int-Array hat 5 Zeilen und 3 Spalten,
// d.h. zahlen_in_tabelle[0], ..., zahlen_in_tabelle[4]
//      sind eindim. int-Arrays der Laenge 3
//      zahlen_in_tabelle[i][j]
//      ist eine int-Variable mit dem Tabellenwert
//      in Zeile i+1 und Spalte j+1

zahlen_in_tabelle = new int[5][3];

for (int i = 0; i < 5; i++)    // Durchlauf aller Zeilen
    for (int j = 0; j < 3; j++) // Durchlauf aller Spalten der Zeile i+1
        zahlen[i][j] = 1;    // besetze Element an Position (i+1,j+1)

```

Alle Arrays werden dynamisch erzeugt, sie müssen aber nicht freigegeben werden. Diese Speicherverwaltungsaufgabe erledigt Java selbst durch die sogenannte "garbage collection". Der Speicherplatz eines Arrays kann dann von Java freigegeben werden, wenn es nicht mehr benötigt wird, d.h. keine Referenz/kein Verweis auf dieses Array mehr existiert.

1.8 * Grundbegriffe in der Objektorientierung

1.8.1 Paket

Ein Paket (Package) ist eine Ansammlung von sinnverwandten Klassen und Schnittstellen (Interfaces).

Die in Java vordefinierte Klasse

String

für Strings (Zeichenketten) befindet sich im Paket

`java.lang`

Zu dem Paket "`java.lang`" gehören außerdem die vordefinierten Klassen

- "System": definiert wichtige Systemeigenschaften wie z.B. das Objekt "`out`" für die Standardausgabe, "`in`" für die Standardeingabe, die Methode "`exit()`" zum Beenden einer Java-Applikation, ...
- "Object": die "Superklasse", von der alle anderen Java-Klassen abgeleitet sind
- "Double": die Klasse zur objektorientierten Verwaltung des Standarddatentyps "`double`", die zudem Methoden zum Abprüfen auf +Unendlich oder -Unendlich erlaubt, Umwandlung von Strings in "`Double`"-Objekte erlaubt, ...
- "Math": die Klasse, in der wichtige mathematische Funktionen wie Sinus, Cosinus, Quadrieren, Maximum/Minimum, ... und Konstanten wie π oder e definiert sind
-

sowie die beiden Interfaces (Kennzeichen: `public interface`)

- "Cloneable": Schnittstelle zum Vervielfältigen von Objekten
- "Runnable": Schnittstelle zum Start von parallelen Programmteilausführungen

(vgl. die von Sun mitgelieferte Dokumentation!).

Die Klassen/Schnittstellen des Pakets "`java.lang`" müssen nicht explizit durch eine `import`-Anweisung

```
import java.lang.*;
// oder:
// import java.lang.String;
```

dem Java-Compiler bekannt gemacht werden.

1.8.2 Klasse

Eine Klasse entspricht einem Datentyp, der eigene Datenelemente (Variablen) und eigene Methoden (Funktionen) enthalten kann. In einer prozeduralen Sprache wie C, Pascal, ... gibt es nur globale oder in Funktionen definierte (lokale) Variablen und globale Funktionen. In Java oder in anderen objektorientierten Sprachen gibt es dagegen eine Bündelung von Variablen und Funktionen zu einer Klasse. In Java gibt es keine globalen Funktionen mehr, alle Funktionen gehören zu irgendeiner Klasse. Es gibt in Java auch keine globalen Variablen, sondern nur Datenelemente, die in Klassen definiert sind. Der in Java vorgesehene Datentyp für Strings ist die Klasse "`java.lang.String`".

Eine Klasse ist ein Prototyp für Variablen eines Datentyps, der durch Datenelemente und Methode festgelegt wird. Die Klasse legt damit gemeinsame Bestandteile und Fähigkeiten von Instanzen (Objekte, Variablen dieser Klasse) fest. Neben objektabhängigen Datenelementen und Methoden kann eine Klasse auch objektunabhängige Datenelementen und Methoden

festlegen, zu deren Benutzung kein Objekt dieser Klasse existieren muß.

In der Dokumentation findet man für in Java vordefinierte Klassen wie z.B. "String" keine aufgeführten Datenelemente (diese sind "versteckt"/zugriffsgeschützt und daher nicht ansprechbar). Eine Implementation wird jedoch etwas Vergleichbares zu einem Array von Zeichen verwenden und ggf. die Länge des Strings in einem Datenelement speichern.

In der Dokumentation findet man zahlreiche Methoden der Klasse "String" wie z.B.

"public int length();"	ermittelt die Länge des Strings
"public char charAt(int i);"	gibt das Zeichen des Stringobjekts an Position (i+1). Zeichen
"public boolean equals(Object anObject);"	testet zwei Stringobjekte auf Gleichheit (d.h. enthalten und gleich lang sind)
"public boolean startsWith(String prefix);"	testet, ob das Stringobjekt mit einem Anfangssuffix beginnt
"public String toLowerCase();"	gibt das Stringobjekt in Kleinbuchstaben zurück
...	...

In Java kann man neben den von Sun vordefinierten Klassen (vgl. Dokumentation) auch eigene Klassen definieren (im folgenden Beispiel die Klasse "VerkehrsZeichen").

```
...
public class VerkehrsZeichen extends Applet
{
    // Datenelemente der Klasse
    int breite, hoehe;

    // Methoden der Klasse
    public void init()
    {
        info("VerkehrsZeichen");
    }

    public void paint(Graphics g)
    {
        Dimension dim = getSize();
        breite = dim.width;
        hoehe = dim.height;
        g.setColor(Color.red);
        ...
    }

    // Beginn der Methode info(...)
    public void info(String name)
    {
        System.out.println("Die Klasse " + name + " zeigt Verkehrszeichen an.");
        ...
    }
    // Ende der Methode info(...)
```

```
}  
// Ende der Klassendefinition
```

Eine Klassendefinition beginnt mit einer der Formen

```
public class KlassenName  
public class KlassenName extends AndererKlassenName  
class KlassenName  
class KlassenName extends AndererKlassenName
```

und schließt Datenelemente und Methoden in geschweifte Klammern ein.

Bsp.:

```
class Uhrzeit  
{  
    public int stunden, minuten, sekunden;  
    public static int max_stunden = 24;  
  
    public Uhrzeit()  
    {  
        stunden = minuten = sekunden = 0;  
    }  
  
    public Uhrzeit(int h, int min, int sec)  
    {  
        stunden = h;  
        minuten = min;  
        sekunden = sec;  
    }  
  
    public String zeigeUhrzeit()  
    {  
        String ausgabe = new String(stunden + ":" + minuten + "." + sekunden);  
        return ausgabe;  
    }  
  
    public void addiereMinuten(int min)  
    {  
        minuten += min;  
  
        int neue_stunden = minuten/60;  
        minuten = minuten % 60;  
  
        stunden += neue_stunden;  
        stunden = stunden % 24;  
    }  
}
```

```

static public Uhrzeit mittagszeit()
{
    Uhrzeit mittag = new Uhrzeit(12, 0, 0);
    return mittag;
}

static public void zeigeMittag()
{
    System.out.println("Mittagszeit ist 12:00.00");
}
}

```

Diese Klasse kann man z.B. in eigenen Applets verwenden:

```

Uhrzeit standard = new Uhrzeit();

String ausgabe;
ausgabe = standard.zeigeUhrzeit();
System.out.println("Standardzeit: " + ausgabe);
System.out.println("    Stunde der Standardzeit: " + standard.stunden);

Uhrzeit vorlesungsbeginn = new Uhrzeit(16, 15, 0);
ausgabe = vorlesungsbeginn.zeigeUhrzeit();
System.out.println("Vorlesungsbeginn: " + ausgabe);
System.out.println("    Stunde des Vorlesungsbeginns: "
    + vorlesungsbeginn.stunden);

vorlesungsbeginn.addiereMinuten(90);
int max_h = Uhrzeit.max_stunden;
System.out.println("max. Stundenzahl: " + max_h);

Uhrzeit.zeigeMittag();

Uhrzeit mittag = Uhrzeit.mittagszeit();
ausgabe = mittag.zeigeUhrzeit();
System.out.println("Mittag: " + ausgabe);

```

Vor der Klassendefinition stehen die "import"-Anweisungen wie z.B.

```

import java.applet.Applet;
import java.awt.*;

// ueberfluessig:
// import java.lang.*;
// oder
// import java.lang.String;

```

In einem Java-Sourcefile können durchaus mehrere Klassen definiert sein, jedoch maximal eine öffentliche (durch `public` spezifiziert) Klasse.

Eine Klasse kann beliebig viele Datenelemente und Methoden aufnehmen.

1.8.3 Objekt

Ein Objekt ist eine Variable, die als Datentyp eine bestimmte Klasse besitzt. Bevor man ein Objekt/eine Variable in Java verwenden kann, muß dem Compiler einmal die zugehörige Klasse/sein Datentyp verraten werden.

Ein Anlegen eines neuen Objekts geschieht im Gegensatz zu Variablen von einem Standarddatentyp immer durch dynamisches Erzeugen mit "new", das Löschen übernimmt Java automatisch (garbage collection). Objektvariablen speichern eigentlich Verweise, mit "new" und einem Konstruktoraufwurf wird also in der Objektvariable ein Verweis (eine Art Speicheradresse) auf das neuerzeugte Objekt gespeichert. Konstruktoren sind in der Klasse vordefinierte Methoden, um ein Objekt zu erzeugen und alle Datenelemente zu initialisieren. Konstruktoren erkennt man daran, dass sie als Methodennamen den Klassennamen haben. In Objektvariablen können auch Verweise auf bereits erzeugte oder durch Funktionsaufrufe zurückgegebene Objekte abgespeichert werden.

```
// in Klassendefinition von "TestApplet":

public class TestApplet extends Applet
{
    ...

    public void init()
    {
        // Objekt gruss von Klasse String wird angelegt
        // aufgerufen wird sog. Konstruktor "public String(String s)"
        String gruss = new String("Hello");

        // greeting enthaelt nur Verweis auf anderes Objekt gruss
        // es wird kein neues Objekt erzeugt
        String greeting = gruss;

        // anrede speichert Verweis auf Null-Objekt (ungueltige Referenz)
        String anrede;

        // anrede erhaelt Verweis vom Teilstring (ersten beiden Buchstaben)
        // von gruss
        anrede = gruss.substring(0, 2);

        System.out.println(gruss + "!");
        System.out.println("Nice to meet you!");

        // beim Erreichen des Endes der Methode init()
```

```

    // wird das Objekt gruss automatisch geloescht
}

public void paint(Graphics g)
{
    // neues Objekt farbe der Klasse Color wird durch RGB-Werte
    // (Konstruktor von Color) erzeugt
    // jedesmal wieder bei Neuaufruf von paint()
    Color farbe = new Color(150, 200, 130);
    g.setColor(farbe);
    g.drawString("neue Farbe!", 20, 40);
    ...
    // beim Erreichen des Endes der Methode paint()
    // wird das Objekt farbe automatisch geloescht
}

...
}

```

Ein Objekt besitzt *eigene* Kopien der Datenelemente der Klasse, sofern diese nicht als "static" (also *objektunabhängig*) qualifiziert sind. Jedes Stringobjekt verwaltet daher seine eigene Zeichenkette, jedes Objekt der Klasse "Polygon" hat seine eigenen (öffentlich zugänglichen, durch "public" gekennzeichneten) Datenelemente

```

public int npoints;    // Anzahl der Ecken
public int xpoints[]; // Vektor der x-Koordinaten der Ecken
public int ypoints[]; // Vektor der y-Koordinaten der Ecken

```

Ansprechen von objektabhängigen Datenelementen:

```

Polygon p = new Polygon();
p.addPoint(10, 100);
p.addPoint(50, 100);
p.addPoint(30, 10);

Polygon p2 = new Polygon();
p2.addPoint(150, 50);
p2.addPoint(125, 25);
p2.addPoint(100, 50);
p2.addPoint(125, 75);

int anz_ecken = p.npoints;    // Anzahl der Ecken von p (hier: 3)
...
anz_ecken = p2.npoints;      // Anzahl der Ecken von p2 (hier: 4)

```

Ansprechen von objektunabhängigen Datenelementen:
Die Klasse `Color` definiert die objektunabhängigen Datenelemente:

```
public static final Color white;
public static final Color black;

public static final Color red;
public static final Color blue;
public static final Color green;

...
```

Da sie als "static" qualifiziert sind, muß man sie mit dem Klassennamen ansprechen. Diese Elemente sind also nicht Bestandteil von Objekten der Klasse `Color`.

```
public void paint(Graphics g)
{
    // Color.red ist das objektunabhaengige Datenelement red
    // der Klasse Color
    g.setColor(Color.red);
    g.drawString("Warnung", 20, 40);

    // Color.blue ist das objektunabhaengige Datenelement blue
    // der Klasse Color
    g.setColor(Color.blue);
    g.drawString("vor dem Hund!", 20, 80);
}
```

Man braucht ein Objekt, um *objektabhängige* Methoden der Klasse aufzurufen oder auf *objektabhängige* Datenelemente zuzugreifen. Dagegen muß überhaupt kein Objekt der Klasse existieren, wenn man *objektunabhängige* (als `static` klassifizierte) Methoden oder Datenelemente benutzen will.

1.8.4 Methode

Eine Methode ist eine Funktion, die in einer Klasse definiert ist. Der Standardaufruf einer Methode geschieht immer über ein Objekt, es sei denn die Methode ist zusätzlich als "static" klassifiziert. In diesem Fall muß sie statt mit dem Objektname mit dem Klassennamen aufgerufen werden.

Beispiel einer objektabhängigen Methode:

```
...
String vorn = new String("Vorname");
String nachn = new String("Nachname");

// die objektunabhaengige Methode length() der Klasse String
// wird fuer das Objekt vorn aufgerufen
```

```

int laenge_1 = vorn.length();

// length() wird fuer das Objekt nachn aufgerufen
int laenge_2 = nachn.length();

System.out.println("laenge_1 = " + laenge_1);
System.out.println("laenge_2 = " + laenge_2);

...

```

Aufruf von vordefinierten objektabhängigen Methoden:

Wenn man in der Java Dokumentation die Beschreibung der Methode `length()` der Klasse `String` nachliest, sieht man dort den Funktionskopf (auch Funktionsprototyp genannt). Dieser beschreibt Zugriffsschutz der Funktion, ihren Rückgabetyt, ob sie objektabhängig oder objektunabhängig (als `static` klassifiziert), ihren Namen und alle Übergabeparameter mit vorangestelltem Datentyp.

Beispiele:

Methode ohne Übergabeparameter:

```

// Beschreibung aus der Java-Dokumentation der Klasse String

//   public int length()

// Zugriffsschutz:      public, d.h. kein Zugriffsschutz vorhanden
// objektabhaengig:    ja, da static fehlt
// Rueckgabedatentyp:  int (Funktion liefert also einen int-Wert)
// Name der Methode:   length
// Argumente der Methode:  (), d.h. keine Argumente

// Aufruf:
String s = "Hallo";
int laenge = s.length();

// oder:
System.out.println("Laenge = " + s.length());

```

weiteres Beispiel:

```

...
String vorn = new String("Vorname");
String nachn = new String("Nachname");

// die objektabhaengige Methode length() der Klasse String
// wird fuer das Objekt vorn aufgerufen
int laenge_1 = vorn.length();

```

```

// length() wird fuer das Objekt nachn aufgerufen
int laenge_2 = nachn.length();

System.out.println("laenge_1 = " + laenge_1);
System.out.println("laenge_2 = " + laenge_2);

...

```

Beispiel der Methode `drawPolygon` der Klasse `Graphics`, die als Rückgabetyt `void` deklariert:

```

// Zugriffsschutz:      public, d.h. kein Zugriffsschutz vorhanden
// objektabh angig:     ja, da static fehlt
// Rueckgabedatentyp:  void, also keine Rueckgabe!
// Name der Methode:    drawPolygon
// Argumente der Methode:  eines, Argument hat Datentyp Polygon

public void drawPolygon(Polygon p)

```

Der Funktionsaufruf liefert also keine Werte zur uck und kann also keiner Variablen als neuer Wert zugewiesen werden.

Aufruf:

```

public void paint(Graphics g)
{
    Polygon dreieck = new Polygon();
    dreieck.addPoint(10, 100);
    dreieck.addPoint(50, 100);
    dreieck.addPoint(30, 10);

    g.drawPolygon(dreieck);
}

```

Dabei ersetzt der *aktuelle* Parameter `dreieck` den *formalen* Parameter `p` in der formalen Spezifikation

```

public void drawPolygon(Polygon p)

```

der Methode `drawPolygon`. In der formalen Spezifikation wird vor den Namen `p` des formalen Parameters immer der Datentyp (hier: `Polygon`) vorangestellt. Beim Aufruf wird der Datentyp weggelassen, allerdings mu  der Datentyp des aktuellen Parameters `dreieck` mit dem des formalen Parameters  bereinstimmen.

Aufruf von vordefinierten objektunabh angigen Methoden:

Die Klasse `Math` definiert z.B. die objektunabh angigen Methoden `Sinus`, `Cosinus`, `Maximum`, `Minimum`, ..., (vgl. die Dokumentation)

```

// Zugriffsschutz:      public, d.h. kein Zugriffsschutz vorhanden
// objektunabhaengig:   nein, da als static deklariert
// Rueckgabedatentyp:   int (Funktion liefert also einen int-Wert)
// Name der Methode:     min
// Argumente der Methode: 2, 1. Argument hat Datentyp int,
//                        2. Argument hat Datentyp int

public static int min(int a, int b) // Minimum zweier int-Zahlen

```

Im Java-Programm wird die Minimum-Funktion "min()" wie folgt aufgerufen:

```

public void paint(Graphics g)
{
    Dimension dim = getSize();
    int breite = dim.width;
    int hoehe = dim.height;

    // Aufruf der objektunabhaengigen Methode min() der Klasse Math
    int durchmesser = Math.min(breite, hoehe);

    setBackground(Color.white);
    g.setColor(Color.blue);
    g.drawOval(0, 0, durchmesser-1, durchmesser-1);
}

```

Dabei ersetzen die *aktuellen* Parameter *breite* und *hoehe* die *formalen* Parameter *a* und *b* in der formalen Spezifikation

```

public static int min(int a, int b)

```

der Methode *min* (vgl. die Java API Dokumentation). In der formalen Spezifikation wird vor den Namen *a* und *b* der formalen Parameter immer der Datentyp (hier: zweimal *int*) vorangestellt. Beim Aufruf wird der Datentyp weggelassen, allerdings müssen die Datentypen der aktuellen Parameter mit denen der formalen Parameter übereinstimmen. Vor dem Namen *min* steht der Datentyp *int* für den Funktionsrückgabewert. Bei dem Funktionsaufruf muß daher als *int*-Wert behandelt werden (hier wird es der *int*-Variable *durchmesser* zugewiesen).

Aufruf von Methoden der Klassen innerhalb der Klassendefinition:

Wird innerhalb einer selbstdefinierten Klasse in einer Methode eine andere in der Klasse definierte Methode aufgerufen, entfällt die Angabe des Objektnamens (bzw. des Klassennamens bei objektunabhängiger Methode) und des ".". Gemeint ist dann bei objektunabhängigen Methoden stets das aktuelle Objekt der Klasse. Zur Unterstreichung kann noch der in Java vordefinierte Verweis namens "**this**" auf das aktuelle Objekt verwendet werden.

Beispiel:

```

class Uhrzeit
{
    // zugriffsgeschuetzte Datenelemente
    private int stunden, minuten, sekunden;
    // zugreifbares objektunabhaengiges Datenelement
    public static int max_stunden = 24;

    public Uhrzeit()
    {
        // gemeint sind die Datenelemente des aktuellen Objekts
        stunden = minuten = sekunden = 0;
    }

    public Uhrzeit(int h, int min, int sec)
    {
        // gemeint sind die Datenelemente des aktuellen Objekts
        stunden = h;
        minuten = min;
        sekunden = sec;
    }

    ...

    public String zeigeUhrzeit()
    {
        // gemeint sind die Datenelemente des aktuellen Objekts
        String ausgabe = new String(stunden + ":" + minuten + "." + sekunden);
        return ausgabe;
    }
}

```

Im Aufruf im Applet wird die Rolle des aktuellen Objekts klarer: Beispiel:

```

// im Konstruktor ohne Argumente ist jetzt standard das aktuelle Objekt
Uhrzeit standard = new Uhrzeit();

String ausgabe;
// in der Methode zeigeUhrzeit() ist jetzt standard das aktuelle Objekt,
// d.h. stunden steht jetzt fuer standard.stunden,
//      minuten steht jetzt fuer standard.minuten,
//      sekunden steht jetzt fuer standard.sekunden
ausgabe = standard.zeigeUhrzeit();

// im Konstruktor ohne Argumente ist jetzt vorlesungsbeginn
// das aktuelle Objekt
Uhrzeit vorlesungsbeginn = new Uhrzeit(16, 15, 0);

```

```

// in der Methode zeigeUhrzeit() ist jetzt ausgabe das aktuelle Objekt
// d.h. stunden steht jetzt fuer vorlesungsbeginn.stunden,
//      minuten steht jetzt fuer vorlesungsbeginn.minuten,
//      sekunden steht jetzt fuer vorlesungsbeginn.sekunden
ausgabe = vorlesungsbeginn.zeigeUhrzeit();

```

Beispiel:

```

// Datenelemente
int breite, hoehe, min_breite, min_hoehe;

public void init()
{
    String min_br = getParameter("minim_breite");
    if (min_br == null)
    {
        System.out.println("Parameter \"minim_breite\" in "
            + "HTML-File nicht definiert!");
        // Ansprechen des Datenelements min_breite
        // des aktuellen Objekts
        min_breite = 100;
    }

    // Aufruf von eigener Methode umwandel()
    // fuer aktuelles Applet-Objekt
    min_breite = umwandel(min_br);

    String min_ho = getParameter("minim_hoehe");

    ...

    // this ist Verweis auf aktuelles Appletobjekt
    this.min_hoehe = this.umwandel(min_ho);
}

public int umwandel(String zahl)
{
    try
    {
        Integer int_obj = new Integer(zahl);
        return int_obj.intValue();
    }
    catch (NumberFormatException exc)
    {
        System.out.println("String \"" + zahl + "\" enthaelt "
            + "keinen korrekten int-Wert!");
    }
}

```

```

        return 0;
    }
}

```

1.9 * Umgang mit in Java vordefinierten Klassen

1.9.1 * Datenelemente

Von Java vordefinierte Klassen verbergen meist all ihre/die meisten ihrer Datenelemente, d.h. diese sind zugriffsgeschützt. Allerdings braucht man nur die öffentlich zugänglichen (nicht zugriffsgeschützten) Datenelemente kennen und ihre öffentlichen Methoden, um die Klasse zu verwenden.

Die Datenelemente findet man in der Klassenbeschreibung der Online-Dokumentation unter dem Abschnitt "Field Summary" (JDK 1.2/1.3). Dort sind Name des Datenelements und der zugehörige Datentyp aufgelistet, ggf. sogar eine Kurzbeschreibung über den Sinn dieses Datenelements.

Bsp.: java.awt.Dimension

```

// in Klasse java.awt.Dimension findet man:
public int height
public int width

```

d.h. jedes Objekt der Klasse Dimension hat diese öffentlichen Datenelemente (und evtl. noch mehr, uns unbekannt, nicht zugreifbare Datenelemente).

```

Dimension groesse = new Dimension(100, 200);
int breite, hoehe;

breite = groesse.width;
hoehe = groesse.height;

// aendert nichts an groesse.height
hoehe = hoehe + 10;

// aendert Hoehe im Objekt groesse
groesse.height = groesse.height + 10;

```

Bsp.: java.awt.Polygon

```

// in Klasse java.awt.Polygon findet man:
public int npoints
public int[] xpoints
public int[] ypoints
protected Rectangle bounds

```

d.h. jedes Objekt der Klasse Polygon hat diese Datenelemente, jedoch kann man nicht auf bounds zugreifen!

```
int [] x_koord = { 0, 199, 99 };
int [] y_koord = { 299, 299, 149 };
Polygon dreieck = new Polygon(x_koord, y_koord, x_koord.length);

int anzahlEcken = dreieck.npoints;

// Fehler, da zugriffsgeschuetzt durch "protected"
// Rectangle grenzen = dreieck.bounds;

for (int i = 0; i < anzahlEcken; i++)
    System.out.println("Ecke " + i + " hat Koordinaten ("
        + dreieck.xpoints[i] + ","
        + dreieck.ypoints[i] + ")");
```

Bsp.: java.awt.Font

```
// in Klasse java.awt.Font findet man:

// objektunabhaengig, da static
public static final int BOLD
public static final int ITALIC
...
public static final int HANGING_BASELINE

// objektabhaengig
protected String name
protected int style
protected int size
```

d.h. jedes Objekt der Klasse Font hat diese Datenelemente, jedoch kann man nicht auf die drei angegebenen objektunabhängigen Datenelemente zugreifen!

```
int stil = Font.BOLD;
...
stil = Font.ITALIC;

// Erzeugen eines Fonts durch Name, Stil, Pixelgroesse
Font boldfaced = new Font("SansSerif", Font.BOLD, 12);
```

Kennt man keine Datenelemente, muss man mit get- und set-Methoden die zugriffsgeschützten Datenelemente abfragen/ändern.

```

Font boldfaced = new Font("SansSerif", Font.BOLD, 12);

String name;
name = boldfaced.getFontName();
System.out.println("Fontname: " + name);

int stil = boldfaced.getStyle();
if (stil == Font.BOLD)
    System.out.println("## Font ist boldfaced!");

System.out.println("## Pixelgr\u00F6\u00DFe: "
    + boldfaced.getSize());

```

Bsp.: java.util.Date

```

Date heute = new Date();

// die get- und set-Methoden von Date sind veraltet,
// die neueren get- und set-Methoden von Calendar sind jedoch komplizierter

// frage Tag ab
System.out.println("Tag: " + heute.getDate());

// frage Monat ab
int monat = heute.getMonth() + 1;
System.out.println("Monat: " + monat);

// frage Jahr ab
int jahr = heute.getYear() + 1900;
System.out.println("Jahr: " + jahr);

// setze Tag
heute.setDate(24);
// setze Monat
heute.setMonth(12-1);
// setze Jahr
heute.setYear(2000 - 1900);

```

1.9.2 * Konstruktoren

Konstruktoren einer Klasse erkennt man daran, dass ihr Methodename gleich dem Klassennamen ist und dass kein Rückgabewert der Methode angegeben ist. Die Konstruktoren findet man in der Klassenbeschreibung der Online-Dokumentation unter dem Abschnitt "Constructor Summary" (JDK 1.2/1.3). Dort sind die verschiedenen Konstruktoren erläutert (Angabe der Datentypen der Argumente).

Bsp.: java.awt.Dimension

```
// in Klasse java.awt.Dimension findet man:  
public Dimension()  
public Dimension(Dimension d)  
public Dimension(int width, int height)
```

d.h. ein neues Objekt der Klasse `Dimension` kann man ohne Vorgaben, aus einem bestehenden Objekt dieser Klasse und aus expliziter Größenangabe (Breite und Höhe) erzeugen.

```
Dimension standard = new Dimension();  
// standard.width = 0;  
// standard.height = 0;  
  
Dimension groesse = new Dimension(100, 200);  
// groesse.width = 100;  
// groesse.height = 200;  
  
Dimension kopie = new Dimension(groesse);  
// kopie.width = 100;  
// kopie.height = 200;
```

Bsp.: java.awt.Polygon

```
// in Klasse java.awt.Polygon findet man:  
public Polygon()  
public Polygon(int[] xpoints, int[] ypoints, int npoints)
```

d.h. ein neues Objekt der Klasse `Polygon` kann man ohne Vorgaben oder mit expliziter Angabe der x- und y-Koordinaten der Ecken und deren Anzahl erzeugen.

```
int [] x_koord = { 0, 199, 99 };  
int [] y_koord = { 299, 299, 149 };  
Polygon dreieck = new Polygon(x_koord, y_koord, x_koord.length);  
// dreieck.xpoints hat selben Komponenten wie x_koord  
// dreieck.ypoints hat selben Komponenten wie y_koord  
// dreieck.npoints stimmt mit x_koord.length (also 3) ueberein  
  
Polygon dreieck_2 = new Polygon();  
// dreieck.npoints ist auf 0 gesetzt  
// dreieck.xpoints ist irgendwie gesetzt  
// dreieck.ypoints ist irgendwie gesetzt
```

Bsp.: java.awt.Font

```
// in Klasse java.awt.Font findet man:
```

```
public Font(Map attributes)
public Font(String name, int style, int size)
```

d.h. ein neues Objekt der Klasse Polygon kann man durch ein vorhandenes Objekt der Klasse, die das Interface Map realisiert (sehr kompliziert) erzeugen oder indem man explizit den Fontnamen, den Fontstil (boldfaced, italic, normal oder eine Kombination) und die Pixelgröße vorschreibt,

```
// Erzeugen eines Fonts durch Name, Stil, Pixelgroesse
// (serifenlos, fette Schrift, 12 Pixel gross)
Font boldfaced = new Font("SansSerif", Font.BOLD, 12);
```

```
// Erzeugen eines Fonts durch Name, Stil, Pixelgroesse
// (serifenlos, fette, kursive Schrift, 15 Pixel gross)
Font bold_italic = new Font("SansSerif", Font.BOLD | Font.ITALIC, 15);
```

```
// Beispiel mit Map-Objekt sehr kompliziert ...
Map attribute = boldfaced.getAttributes();
```

```
Font kopie_von_boldfaced = new Font(attribute);
```

Bsp.: java.util.Date

```
// in Klasse java.util.Date findet man:
```

```
public Date()
public Date(long date)
public Date(int year, int month, int date) // veraltet seit JDK 1.1
public Date(int year, int month, int date,
            int hrs, int min) // veraltet seit JDK 1.1
public Date(int year, int month, int date,
            int hrs, int min, int sec) // veraltet seit JDK 1.1
public Date(String s) // veraltet seit JDK 1.1
```

Anwendung:

```
// heutiges Datum wird erzeugt
Date heute = new Date();
```

```
// Anzahl der Sekunden nach 1.1.1970 ermitteln
long anz_milli_sekunden_nach_1970 = heute.getTime();
```

```

// heutiges Datum wird erzeugt
long milli_sekunden_gestern = anz_milli_sekunden_nach_1970
    - 24L*60L*60L*1000L;

// gestriges Datum wird erzeugt
Date gestern = new Date(milli_sekunden_gestern);

```

1.9.3 * Methoden

Die Methoden findet man in der Klassenbeschreibung der Online-Dokumentation unter dem Abschnitt "Field Summary" (JDK 1.2/1.3). Dort sind der Name der Methode, die Anzahl und die jeweiligen Datentypen der Argumente sowie der Datentyp des Ergebnisses der Methode (Rückgabe) aufgeführt, ggf. sogar eine Kurzbeschreibung über den Sinn dieser Methode. Bei den Methoden unterscheidet man zwischen objektunabhängigen Methoden (in der Dokumentation als `static` gekennzeichnet) und objektabhängigen (`static` fehlt in der Dokumentation). Danach richtet sich dann auch der Aufruf (bei objektunabhängigen Methoden braucht man kein Objekt, es muss keines vorhanden sein, es genügt der Klassenname; bei objektabhängigen braucht man ein vorhandenes Objekt, mit dem man die Methode aufruft).

Bsp.: `java.awt.Polygon`

```

// in Klasse java.awt.Polygon findet man:

public void addPoint(int x, int y)
public boolean contains(double x, double y)
...
public void translate(int deltaX, int deltaY)

```

d.h. für jedes Objekt der Klasse `Polygon` kann man diese Methoden aufrufen!

```

int [] x_koord = { 0, 199, 99 };
int [] y_koord = { 299, 299, 149 };
Polygon dreieck = new Polygon(x_koord, y_koord, x_koord.length);

// verschiebe Ecken von dreieck
dreieck.translate(10, 20);

Polygon strecke = new Polygon();
// addiere neue Ecke zu strecke
strecke.addPoint(-10, -10);

// verschiebe Ecken von strecke
strecke.translate(10, 20);

```

1.10 * Umgang mit Strings in Java

Als Motivation zum Umgang mit Strings dient das folgende Beispiel:

Beispiel 1.1 *In einem Applet sollen die Fluggesellschaft und die Nummer des Fluges in Texteingabefeldern eingegeben werden. Nach Drücken eines Buttons soll aus den Großbuchstaben der Fluggesellschaft und der eingegebenen Nummer die Flugnummer berechnet und in einem Textausgabefeld berechnet werden.*

Dazu muss man den eingegebenen Text aus den Texteingabefeldern ermitteln. Danach durchläuft man den String für die Fluggesellschaft Zeichen für Zeichen (Länge des Strings muss ermittelt werden!) und prüft für jedes Zeichen, ob es ein Großbuchstabe ist. Jeden gefundenen Großbuchstaben hängt man hinten an einen Ergebnisstring. Danach hängt man ein Leerzeichen an zur Trennung von den Großbuchstaben und der Nummer und danach die eingegebene Nummer.

Man beachte, dass die eingegebene Nummer nicht als ganze Zahl abgelegt wird, sondern wieder nur als String zur Verfügung steht.

Zudem sollen fehlende Eingaben bzw. fehlende Großbuchstaben in dem Namen der Fluggesellschaft entdeckt und bemängelt werden.

Einige Stringobjekte muss man also anlegen, deren Länge ermitteln, Zeichen für Zeichen durchlaufen und sie mit anderen Strings vergleichen.

Einige relevante Code-Ausschnitte:

```
import java.awt.*;
import java.awt.event.*; // noetig fuer das Interface ActionListener
import java.applet.*;

public class FlugNummer extends Applet
    implements ActionListener // noetig, um Button-Druck
    // zu erkennen
{
    // wichtige Datenelemente, werden in init() und in actionPerformed() gebraucht
    TextField tf_nummer;
    TextArea ta_flug_nummer;

    public void init()
    {
        // Anlegen der TextField-Objekte fuer die Texteingabefelder (1 Zeile)
        tf_flug_gesellschaft = new TextField(30);
        tf_nummer = new TextField(30);

        // Anlegen des TextArea-Objekts fuer das Textausgabefeld (mehrere Zeilen
        // moeglich)
        ta_flug_nummer = new TextArea(30,

        // Gitteranordnung der Labels und Textein- und ausgabefelder
        // durch GridLayout garantieren
        ...
    }
}
```

```

}

// wird automatisch aufgerufen, wenn der Button gedrueckt wird
// Methode gehoert zum Interface ActionListener

public void actionPerformed(ActionEvent e)
{
    // leeren String als neues Objekt anlegen
    String flug_nummer = new String();

    // Ermitteln des eingegebenen Textes bei Fluggesellschaft
    String s_gesellschaft = tf_flug_gesellschaft.getText();

    char c;
    for (int i = 0; i < s_gesellschaft.length(); i++)
    {
        c = s_gesellschaft.charAt(i);    // i-tes Zeichen des Strings
        if (c >= 'A' && c <= 'Z')      // Erkennen von Grossbuchstaben
            flug_nummer = flug_nummer + c; // Anhaengen des Grossbuchstabens
    }

    // "" ist neues (leeres) Stringobjekt, s_fehler zeigt auf dieses
    String s_fehler = "";

    // Wurden Grossbuchstaben erkannt oder ist der String leer?
    if (flug_nummer.equals(""))
    {
        ...
    }
    else
    {
        // Ermitteln des eingegebenen Textes bei Nummer,
        // getText() liefert Stringobjekt
        String s_nummer = tf_nummer.getText();

        // Ist eingegebene Nummer nicht leer? "" ist ein Stringobjekt!
        if (!"".equals(s_nummer))
        {
            flug_nummer = flug_nummer + ' '; // haenge Leerzeichen an
            flug_nummer = flug_nummer + s_nummer; // haenge Nummer als String an

            // setze berechnete Flugnummer als neuen Text im Textausgabefeld
            ta_flug_nummer.setText(flug_nummer);
        }
    }
}
}

```

}

1.10.1 * Anlegen von Strings

Strings oder auch Zeichenketten genannt sind in Java einfach eine Folge von Zeichen beliebiger Länge (also z.B. ein Text mit Leerzeichen, ein Wort, ein Buchstabe, ...).

In Java sind Strings

Objekte der Klasse String.

Strings können dabei *variable* oder *konstante* Objekte sein, letztere erkennt man daran, daß sie in Doppelanführungszeichen

"

eingeschlossen sind.

Beispiele für konstante Strings:

""	leerer String, enthält keine Zeichen, Länge: 0
"A"	String aus einem Buchstaben, enthält das Zeichen A, Länge: 1
" "	String aus einem Buchstaben, enthält das Leerzeichen, Länge: 1
"\""	String aus einem Buchstaben, enthält das Doppelanführungszeichen, Länge: 1
"\n"	String aus einem Buchstaben, enthält das Newline-Zeichen, Länge: 1
"Java"	String aus 4 Buchstaben, Länge: 4
"Java 1.1"	String aus 8 Buchstaben mit eingeschlossenem Leerzeichen, Länge: 8
"Er sagte: \"Hallo!\""	String aus 18 Buchstaben mit zwei enthaltenen Doppelanführungszeichen, Länge: 18

Beachte: Bestimmte Sonderzeichen wie Doppelanführungs- oder Newline-Zeichen müssen im String durch das beginnende Erkennungszeichen '\' markiert werden (auch Maskierung genannt).

Beispiele für variable Strings:

String s = new String();	leerer String s
String s = new String("");	leerer String s
String s;	String implizit mit Referenz null vorbesetzt
String s = null;	String explizit mit Referenz null vorbesetzt
String s = new String("A");	String mit einem Zeichen
String s = new String("Java 1.1");	String mit zwei Wörtern

abkürzende Schreibweise (nur für Strings erlaubt):

```
String s = "";           leerer String s
String s = "A";         String mit einem Zeichen
String s = "Java 1.1";  String mit zwei Wörtern
```

Anlegen von Stringobjekten:

Stringobjekte werden in Funktionen oder Klassen definiert.

a) Definition innerhalb von Funktionen:

Dazu werden die Definitionen der Stringobjekte nach dem Funktionsnamen und zwischen '{' und '}' eingefügt.

```
public class Test
{
    public static void main(String [] argv)
    {
        String s1 = "Java";           // s1 erhaelt den Text 'Java'
        String s2 = "";              // s2 ist der leere String

        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
    }
}
```

Abkürzend kann man schreiben:

```
public class Test
{
    public static void main(String [] argv)
    {
        // s1 UND s2 sind Strings
        String s1 = "Java", s2 = ""; // s1 erhaelt den Text "Java"
                                    // s2 ist der leere String

        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
    }
}
```

In Applets fügt man die Stringdefinition z.B. in die Methode "paint()" ein.

```
import java.applet.Applet;
import java.awt.Graphics;

public class StringApplet_1 extends Applet
{
    public void paint(Graphics g)
    {
```

```

        String titel = "Kurzer Titel";
        g.drawString(titel, 20, 20);
    }
}

```

Da die Methode "drawString()" als 1. Argument ein Stringobjekt erwartet, kann man statt dem konstanten String "Kurzer Titel" auch den variablen String "titel" an die Methode übergeben.

b) Definition außerhalb von Funktionen:

Oftmals bietet es sich an, Stringobjekte außerhalb den Methoden der Klasse, aber innerhalb der Klasse zu definieren. Damit werden diese Stringobjekte zu Datenelementen der Klasse, die in allen (objektabhängigen) Methoden der Klasse verfügbar sind. Das hat den Vorteil, daß man jetzt in allen Methoden auf die Stringobjekte zugreifen kann.

```

import java.applet.Applet;
import java.awt.*;          // fuer Color und Graphics

public class StringApplet_2 extends Applet
{
    // text ist Datenelement der Klasse StringApplet_2
    String text = "wenig Neues";

    public void init()
    {
        // richtig: text ist in allen Methoden vorhanden
        System.out.println("Es gibt " + text + "!");

        // Fehler: titel nur in paint() definiert
        // System.out.println("Titel: " + titel);
    }

    public void paint(Graphics g)
    {
        // titel ist kein Datenelement
        String titel = "Kurzer Titel";
        g.drawString(titel, 20, 20);

        // setzt die Farbe auf Blau dauerhaft um
        // String wird in blau ausgegeben
        g.setColor(Color.blue);

        // richtig: text ist in allen Methoden vorhanden
        g.drawString(text, 20, 100);
    }
}

```

einzelne Zeichen von Strings:

Einzelne Zeichen des Strings haben den Datentyp `char` (bedeutet: character). Strings erlauben zwar das Auslesen einzelner Zeichen mit der Methode `charAt(int index)`, nicht aber das Ändern einzelner Zeichen. Dazu muß man `StringBuffer`-Objekte verwenden, die eine Methode `setCharAt(int index, char neues_zeichen)` anbieten.

```
String gruesse = "Hallo";
System.out.println("1. Zeichen: " + gruesse.charAt(0));
System.out.println("2. Zeichen: " + gruesse.charAt(1));
int index = gruesse.length() - 1;
System.out.println("letztes Zeichen: " + gruesse.charAt(index));

// Durchlauf aller Zeichen
for (int i = 0; i < gruesse.length(); i++)
    System.out.println("Zeichen " + (i+1) + ": " + gruesse.charAt(i));
```

Obwohl sich *einzelne* Zeichen nicht ändern lassen, kann man den gesamten String ändern (genauer: man ändert die Referenz auf das Stringobjekt).

```
String prog = "applet";
System.out.println("prog vor der \u00C4nderung: " + prog);
prog = "Applet";
System.out.println("prog nach der \u00C4nderung: " + prog);
```

'+' bei Strings

Die Klasse `String` erlaubt mit einer leichten Notation, an einen vorhandenen String etwas anzuhängen. Man kann durchaus mehrfach anhängen.

```
String prog = "Java";

System.out.print("prog = ");
// Wert "Java" wird ohne Doppelanführungszeichen ausgegeben
System.out.println(prog);

// String "Java-Programm" steht in prog_2
String prog_2 = prog + "-Programm";
System.out.print("prog_2 = ");
System.out.println(prog_2);

// String "prog = Java" steht in s
String s = "prog = " + prog;
System.out.println(s);
// kuerzer:
System.out.println("prog = " + prog);

String s2 = "", typ = "Applet";
```

```
// String "Java-Applet" steht in s2
s2 = prog + "-" + typ;
System.out.println("s2 = " + s2);
```

Bei der Verwendung von "+" müssen nicht beide Operanden Strings sein. Alle Standard-datentypen wie "int", "double", ... und die meisten vordefinierten Klassen sind in Strings umwandelbar.

Beispiele:

```
public void paint(Graphics g)
{
    Dimension dim = getSize();
    int breite = dim.width;
    int hoehe = dim.height;

    // Kurzform fuer:
    // System.out.print("Breite des Appletfensters: ");
    // System.out.println(breite); // Ausgabe des int-Wertes (als String)
    System.out.println("Breite des Appletfensters: " + breite);

    System.out.println("Hoehe des Appletfensters: " + hoehe);
    System.out.println("Groesse des Appletfensters: " + dim);
}
```

Bei dem obigen Programm wird mit der Ausgabemethode "println()" des Objekts "System.out" folgendes durch Umwandlung in Strings ausgegeben:

- "Breite des Appletfensters: " + breite: wandelt `int`-Wert `breite` in einen String um (Wert als Zeichenkette) und hängt diesen an den String "Breite des Appletfensters: "
 " Ist die aktuelle Breite 400 Pixel, wird also der String "Breite des Appletfensters: 400" ausgegeben
- "Hoehe des Appletfensters: " + hoehe: wandelt `int`-Wert `hoehe` in einen String um (Wert als Zeichenkette) und hängt diesen an den String "Hoehe des Appletfensters: "
 " Ist die aktuelle Höhe 300 Pixel, wird also der String "Hoehe des Appletfensters: 300" ausgegeben
- "Groesse des Appletfensters: " + dim: wandelt das Objekt `dim` der Klasse `Dimension` in einen String um (Bezeichnung + Datenelemente als Zeichenkette) und hängt diesen an den String "Groesse des Appletfensters: "
 " Ist die aktuelle Größe 400 x 300 Pixel, wird also ein String der Art "Groesse des Appletfensters: `java.awt.Dimension[width=400,height=300]`" ausgegeben

einige wichtige Methoden für Strings:

Eine Klasse wie z.B. `String` bietet Funktionen an, die über Objekte und dem `'.'` aufgerufen werden.

Method	Beispielaufruf	Bedeutung
length(..)	<pre>String user = getParameter("name"); if (user.length() > 8) { System.out.println("Zu lang!"); }</pre>	Stringobjekt <code>user</code> wird aus dem PARAM-Tag des HTML-Files ausgelesen und dessen Länge ermittelt.
equals(..)	<pre>String user = getParameter("name"); if (user.equals("administrator")) { System.out.println("Superuser!"); }</pre>	Stringobjekt <code>user</code> wird aus dem PARAM-Tag des HTML-Files ausgelesen und dann mit dem String <code>"administrator"</code> auf Gleichheit geprüft.
substring(..)	<pre>String flugnr = "LH 1234"; String kuerzel = flugnr.substring(0, 2); String nr = flugnr.substring(3);</pre>	Stringobjekt <code>flugnr</code> wird besetzt mit <code>"flugnr"</code> , das Stringobjekt <code>kuerzel</code> enthält die ersten beiden Buchstaben <code>"LH"</code> (0 ist Startindex, 2-1 = 1 ist Endindex) <code>nr</code> enthält ab dem vierten Buchstaben alle Zeichen des Strings, also <code>"1234"</code> (3 ist Startindex)

Beispiel:

```
String s1 = "Applet";
String s2 = "Applikation";

System.out.print("String s1: ");
System.out.println(s1);

// kuerzer, aber aequivalent fuer s2:
System.out.println("String s2: " + s2);

if (s1.equals(s2))
    System.out.println("Nanu? Strings sind gleich!");
else
    System.out.println("Strings sind verschieden!");

if (s1.equals("Applet"))
{
    System.out.println("Habe ich mir gedacht!");
    System.out.println("In s1 steht Applet!");
}
else
```

```

{
    System.out.println("Katastrophe!");
    // Abbruch mit Fehlercode 1
    System.exit(1);
}

System.out.println("Laenge von String s1: " + s1.length());
System.out.println("Laenge von String s2: " + s2.length());

// n erhaelt Laenge vom konstanten String "Hallo" (also 5)
int n = "Hallo".length();
System.out.print("Laenge von \"Hallo\": ");
System.out.println(n);

s1 = "Java ist meine Nummer 1!";
System.out.println("s1 = " + s1);
System.out.println("s1 (kleingeschrieben) = " + s1.toLowerCase());

```

Die Klasse `String` bietet ca. 50 verschiedene Methoden an, davon haben 26 Methoden verschiedenen Namen. Es gibt also Methoden mit gleichem Funktionsnamen, aber *unterschiedlichen* Übergabeparametern.

In der Dokumentation zu den APIs (Application Programming Interface = API) des JDK finden Sie eine detaillierte Beschreibung der Klasse `String` im Paket `java.lang`.

1.10.2 * Konvertierung von/in Strings

Beispiel 1.2 *Es soll ein Applet geschrieben werden, das im HTML-File angibt, welche Pizza mit welchem Preis und welches Getränk mit welchem Preis in einem Restaurant bestellt wurde.*

Den Preis der Pizza und des Getränks soll auch nach dem Start des Applets noch änderbar sein.

Das Applet soll den (nicht veränderbaren) Gesamtpreis berechnen und ausgeben (alle Preise sollen in Euro sein).

Vorgehen: *Das HTML-File muss einen Parameter mit einem Namen, z.B. "NamePizza", in einem PARAM-Tag innerhalb des APPLET-Tags spezifizieren und ihm einen Wert (immer ein String) zuweisen.*

HTML-File:

```

<APPLET CODE="Restaurant.class" WIDTH="850" HEIGHT="250">
  <PARAM NAME="NamePizza" VALUE="Calzone">

```

...

Entweder ist der WWW-Browser nicht `javafähig` oder die `Ausführung` von Java-Programmen ist bei den Optionen untersagt worden.

```

</APPLET>

```

Im Applet wird ein Datenelement vom Datentyp *"String"* eingeführt, z.B. mit Namen *"name_pizza"*. Mit der Applet-Methode *"getParameter(..)"* (einziges Argument ist der Name des Parameters, hier *"NamePizza"*), kann dieser Wert ermittelt werden. Da damit ein Datenelement initialisiert wird, sollte man dies in der Methode *"init()"* durchführen. Ist dieser Parameter im *NAME*-Attribut eines *PARAM*-Tag in dem *HTML*-File vorhanden, wird der Wert des zugehörigen *VALUE*-Attributs (immer) als *String* zurückgegeben. Fehlt diese Parameterangabe, wird stattdessen die (ungültige) Referenz *"null"* zurückgegeben. Dieser Fall sollte berücksichtigt werden und ggf. das Datenelement einen Standardwert erhalten (oder ein Abbruch/Ende des Applets eingeleitet werden).
Ausschnitt aus dem Java-Applet:

```
public class Restaurant extends Applet ...
{
    // Datenelemente definieren
    // vom HTML-File uebergebener Wert des Namens
    String name_pizza;
    // Defaultwert fuer Datenelement name_pizza
    String def_name_pizza;
    ...

    public void init()
    {
        // Defaultwert setzen
        def_name_pizza = "Maestro";

        name_pizza = getParameter("NamePizza");
        if (name_pizza == null)
        {
            // z.B. Fehler ausgeben
            System.out.println("Fehler! Parameterwert \"NamePizza\" fehlt");

            // Wert mit Defaultwert vorbesetzen
            name_pizza = def_name_pizza;
        }

        ...
    }
}
```

Mit der Preisangabe geht man mit einem *double*-Datenelement *preis_pizza* analog vor (auch die Zahl wird - leider - als *String* übermittelt). Man muss daher zunächst den *String* ermitteln und dann diesen in einen *double*-Wert umwandeln. Dabei kann jedoch eine Umwandlung unmöglich sein, wenn der *String* gar kein gültiges Zahlenformat hat. Diese Ausnahmesituation (in Java ein automatisch erzeugtes Objekt der Klasse *NumberFormatException*) muss in einer *try-catch*-Anweisung abgefangen werden. Der *try*-Block wird sofort abgebrochen, wenn ein Fehlerfall eintritt, und es wird in den *catch*-Block verzweigt, wenn der

*Fehlerfall mit dem Datentyp der Ausnahmesituation (hier: `NumberFormatException`) übereinstimmt. In dem `catch`-Block sollte eine Fehlerwarnung ausgegeben werden und ggf. ein Standardwert für den Preis gesetzt werden (oder das Applet beendet werden).
Ausschnitt aus dem Java-Applet:*

```
public class Restaurant extends Applet ...
{
    // Datenelemente definieren
    //  Preis der Pizza als Datenelement
    double preis_pizza;
    //  Default-Preis der Default-Pizza
    double def_preis_pizza;
    ...

    public void init()
    {
        // Defaultwert setzen
        def_preis_pizza = 7.10;

        String param_preis_pizza = getParameter("PreisPizza");
        if (param_preis_pizza == null)
        {
            // z.B. Fehler ausgeben
            System.out.println("Fehler! Parameterwert \"PreisPizza\" fehlt");

            // Wert mit Defaultwert vorbesetzen
            preis_pizza = def_preis_pizza;
        }
        else
        {
            try
            {
                // versuchte Umwandlung von String in double-Wert
                //  ueber den Umweg Konstruktor von Wrapperklasse "Double"
                Double d_obj = new Double(param_preis_pizza);

                // im Fehlerfall wird autom. ein Objekt der Klasse
                //  NumberFormatException erzeugt und diese Anweisung
                //  nicht mehr ausgefuehrt -> Sprung in den catch-Block
                preis_pizza = d_obj.doubleValue();
            }
            // wird nur im passenden Fehlerfall ausgefuehrt
            catch (NumberFormatException exc)
            {
                System.out.println("Fehler! Umwandlung des Strings \""
                    + param_preis_pizza + "\"");
                System.out.println("          f\u00fcr Preis gescheitert!");
            }
        }
    }
}
```

```

        // Default-Werte setzen
        name_pizza = def_name_pizza;
        preis_pizza = def_preis_pizza;
    }
}

...
}
}

```

Die so erhaltenen Variablen werden benutzt, um die Vorgabewerte für die Texteingabefelder und die Labels zu setzen sowie den Gesamtpreis zu berechnen. Allerdings muss der Preis in der `double`-Variable zur Anzeige im Texteingabefeld wieder in einen String umgewandelt werden.

Ausschnitt aus dem Java-Applet:

```

public class Restaurant extends Applet ...
{
    // Datenelemente definieren
    Label lab_name_pizza;
    TextField tf_preis_pizza;
    ...

    public void init()
    {
        ...

        // uebergebenen Name der Pizza als Label setzen
        // (Verkettung von Strings)
        lab_name_pizza = new Label("Pizza " + name_pizza + ":");

        // uebergebenen Preis der Pizza (double-Variable) in
        // String umwandeln und damit Texteingabefeld initialisieren
        // (30 = Anzahl der zu zeigenden Zeichen des Texteingabefelds)
        tf_preis_pizza = new TextField("" + preis_pizza, 30);

        ...
    }
}

```

Es gibt zahlreiche Möglichkeiten, einen beliebigen Datentyp in einen String umzuwandeln (oder in der umgekehrten Richtung). Ein gängiger Weg ist jeweils als erster Punkt beschrieben.

- (i) beliebigen Datentyp in String umwandeln einfachste Möglichkeit (geht mit allen Datentypen): Anhängen des Wertes/der Variable von einem Datentyp an den leeren String:

```

int n = 123;
String n_als_string = "" + n;          // enthaelt String "123"

String zahl_als_string = "" + 234;    // enthaelt String "234"

double x = 6.78;
String x_als_string = "" + x;         // enthaelt String "6.78"

Dimension dim = new Dimension(100, 200);
String dim_als_string = "" + dim;
// enthaelt String "java.awt.Dimension[width=100,height=200]"

// oder:
// String dim_als_string = dim.toString();

```

Für Objekte kann man die objektabhängige Methode `toString()` verwenden, die es für alle Klassen gibt.

weitere Möglichkeiten:

a) objektunabh. Methode `valueOf()` von `String`

```

static String valueOf(boolean b)
static String valueOf(char c)
...
static String valueOf(float f)
static String valueOf(double d)
static String valueOf(char [] values)
static String valueOf(Object obj)

```

Bsp.:

```

double x = 0.5;
String s = String.valueOf(x);

```

b) Konstruktoren von `String`

```

public String (String s)
public String (StringBuffer sb)
public String (byte [] bytes)
public String (char [] values)

```

Bsp.:

```

char [] wort = { 'g', 'u', 't' };
String s = new String(wort);

```

oder:

```

s = String.copyValueOf(wort);

```

c) objektunabh. Methode `toString()` der Wrapperklassen

```

static String toString(<zugeh. Standarddatentyp> value)

```

für die Wrapperklassen

Byte, Double, Float, Integer, Long, Short,
nicht für Boolean und Character.

Bsp.:

```
double x = 0.5;
String s = Double.toString(x);
```

d) objektabh. Methode toString() der Wrapperklassen

```
String toString()
```

(vgl. c), aber auch für Boolean und Character).

Bsp.:

```
Double x_obj = new Double(0.5);
String s = x_obj.toString();
```

e) weitere klassenspezifische Umwandlungen

```
static String toBinaryString(int i)
static String toOctalString(int i)
static String toHexString(int i)
static String toBinaryString(long l)
static String toOctalString(long l)
static String toHexString(long l)
static String copyValueOf(char [] values)
```

Bsp.:

```
int i = 31;
String s_bin = String.toBinaryString(i);
String s_oct = String.toOctalString(i);
String s_hex = String.toHexString(i);
```

f) Stringkonkatenation

```
"" + <Ausdruck eines Standarddatentyps>
<Ausdruck eines Standarddatentyps> + ""
<Ausdruck vom Typ String> + <Ausdruck eines Standarddatentyps>
<Ausdruck eines Standarddatentyps> + <Ausdruck vom Typ String>
```

Bsp.:

```
double x = 0.5;
int i = 31;
String s = "" + i;
s = "i = " + i;
s = x + "";
s = x + "(Wert von x)";
```

(ii) String in beliebigen Datentyp umwandeln eine Möglichkeit (geht mit allen Standarddatentypen außer mit char):

Finde die passende Wrapperklasse zum Standarddatentyp (im Beispiel unten die Klasse Integer zum Standarddatentyp int), erzeuge ein Objekt durch den Konstruktor

mit einem Stringargument (hier: `Integer n_obj = new Integer(s)`), rufe deren objektabhängige Methode `xyzValue()` mit diesem Objekt auf (hier: `n_obj.intValue()`), die Rückgabe ist dann ein Wert des Standarddatentyps (hier: `int`).

Allerdings muss man den Aufruf in eine `try-catch`-Umgebung einbauen, damit man Fehler beim Umwandeln (in Java sogenannte `Exceptions` = Ausnahmesituationen) abfangen kann. Im Fehlerfall sollte man entweder abbrechen oder einen sinnvollen Defaultwert setzen.

```
String int_als_string = "123";
int n;

try
{
    Integer n_obj = new Integer(int_als_string);
    n = n_obj.intValue();
}
catch (NumberFormatException exc)
{
    System.out.println("Fehler bei der Umwandlung int -> String!");
    System.out.println("Systemmeldung:");
    System.out.println(exc);
    n = 0;
}

String double_als_string = "123";
double x;

try
{
    Double x_obj = new Double(double_als_string);
    x = x_obj.doubleValue();
}
catch (NumberFormatException exc)
{
    System.out.println("Fehler bei der Umwandlung double -> String!");
    System.out.println("Systemmeldung:");
    System.out.println(exc);
    x = 0.0;
}
```

Dies geht für alle Standarddatentypen bis auf `char`, `boolean` spielt eine Sonderrolle, da beim Anlegen des Objekts keine Exception erzeugt wird.

Standarddatentyp	Wrapperklasse	Umwandlungsmethode
bool	Boolean	boolean booleanValue()
byte	Byte	byte byteValue()
double	Double	double doubleValue()
float	Float	float floatValue()
int	Integer	int intValue()
long	Long	long longValue()
short	Short	short shortValue()

Eine andere Möglichkeit für alle Standarddatentypen außer boolean und char ist folgende:

Finde die passende Wrapperklasse zum Standarddatentyp (im Beispiel unten die Klasse `Integer` zum Standarddatentyp `int`), rufe deren objektunabhängige Methode `parseXYZ(...)` mit einem String auf (hier: `Integer.parseInt(...)`), die Rückgabe ist dann ein Wert des Standarddatentyps (hier: `int`).

Allerdings muss man den Aufruf in eine `try-catch`-Umgebung einbauen, damit man Fehler beim Umwandeln (in Java sogenannte `Exceptions` = Ausnahmesituationen) abfangen kann. Im Fehlerfall sollte man entweder abbrechen oder einen sinnvollen Defaultwert setzen.

```
String int_als_string = "123";
int n;

try
{
    n = Integer.parseInt(int_als_string);
}
catch (NumberFormatException exc)
{
    System.out.println("Fehler bei der Umwandlung int -> String!");
    System.out.println("Systemmeldung:");
    System.out.println(exc);
    n = 0;
}

String double_als_string = "123";
double x;

try
{
    x = Double.parseDouble(double_als_string);
}
catch (NumberFormatException exc)
{
    System.out.println("Fehler bei der Umwandlung double -> String!");
    System.out.println("Systemmeldung:");
}
```

```

    System.out.println(exc);
    x = 0.0;
}

```

Wieder muss man wie oben eine Exception abfangen.
weitere Möglichkeiten (meistens ist auch hier eine Ausnahmebehandlung nötig):

- a) objektabh. Methoden von String

```

char charAt(int index)
byte [] getBytes() // höherwertige Bytes gehen verloren
void getChars(int posStart, int posEnd,
               char [] dest, int posDestStart)
char [] toCharArray()

```

Bsp.:

```

String s = "k";
char c = s.charAt(0);
String s = "gut";
char [] values = s.toCharArray();

```

- b) Konstruktoren der Wrapperklassen

```

public <Wrapperklasse>(String s)

```

für die Wrapperklassen

Boolean, Byte, Double, Float, Integer, Long, Short,

nicht für Character.

Bsp.:

```

String s = "0.5";
Double x_obj = new Double(s);
double x = x_obj.doubleValue();

```

kürzer:

```

String s = "0.5";
double x = new Double(s).doubleValue();

```

- c) objektunabh. Methode valueOf() der Wrapperklassen

```

static <Wrapperklasse> valueOf(String s)

```

für alle Wrapperklassen außer Character.

Bsp.:

```

String s = "31";
Integer i_obj = Integer.valueOf(s);
int i = i_obj.intValue();

```

kürzer:

```

String s = "31";
int i = Integer.valueOf(s).intValue();

```

d) weitere klassenspezifische Umwandlungen

Klasse Byte:

```
static Byte decode(String s)
static byte parseByte(String s)
static byte parseByte(String s, int radix)
```

Klasse Double:

```
static double parseDouble(String s)
```

Klasse Float:

```
static float parseFloat(String s)
```

Klasse Integer:

```
static Integer decode(String s)
static int parseInt(String s)
static int parseInt(String s, int radix)
```

Klasse Long:

```
static Long decode(String s)
static long parseLong(String s)
static long parseLong(String s, int radix)
```

Klasse Short:

```
static Short decode(String s)
static short parseShort(String s)
static short parseShort(String s, int radix)
```

Bsp.:

```
String s = "31";
Integer i_obj = Integer.decode(s);
int i = i_obj.intValue();
i = Integer.parseInt(s);
i = Integer.parseInt(s, 10);
i = Integer.parseInt(s, 16);
```

1.10.3 * Hauptanwendung von Strings

Strings werden in Java vielfältig eingesetzt:

- (i) Übergabe von Kommandozeilenargumenten an Applikationen
Dies erfolgt über den Funktionsparameter `argv` in der Funktion `main()`. `argv` ist dabei ein Array von Strings (d.h. `argv` besteht aus einzelnen Komponenten, die alle Strings sind; die Komponenten werden über die Indizes 0 bis Länge-1 angesprochen).

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

Dann ist `argv.length` die Anzahl der übergebenen Argumente und

```

argv[0]:          1. Argument als String
argv[1]:          2. Argument als String
                :
                :
argv[argv.length-1]: letztes Argument als String

```

Der Aufruf "java Zins Robert.Baier 5 3.5" übergibt also die drei Strings Robert.Baier" in argv[0], "5" in argv[1] und "3.5" in argv[2] (und nicht etwa den int-Wert 5 und den double-Wert 3.5). Hier ist argv.length gleich 3. Wird nichts übergeben, ist argv.length gleich 0.

(ii) Übergabe von Parametern an Applets

Dies erfolgt über die HTML-Seite innerhalb des APPLET-Befehls im PARAM-Befehl. Dabei gibt NAME den Namen des Parameters für das Java-Programm an und VALUE den Wert des Parameters als String.

```

<APPLET CODE="ZinsApplet.class" HEIGHT=300 WIDTH=400>
  <PARAM NAME="namen" VALUE="Robert.Baier">
  <PARAM NAME="jahre" VALUE="5">
  <PARAM NAME="zinssatz" VALUE="3.5">
</APPLET>

```

Im Java-Applet-Programm kann man dann durch Aufruf von "getParameter()" mit Angabe der Namensbezeichnung des Parameters den Wert als String erhalten:

```

String person = getParameter("namen");
String zinsjahre = getParameter("jahre");
String zinsprozente = getParameter("zinssatz");

```

Wird nichts übergeben (d.h. ist kein PARAM-Tag im HTML-File mit dem gewünschten Namen vorhanden), ist der String (z.B. person) gleich der Nullreferenz null.

```

String person = getParameter("namen");
if (person == null)
{
  System.out.println("Fehler!");
  System.out.println("Der Parameter \"namen\" wurde nicht im HTML-File vorbes
  // mit Defaultwert besetzen (oder abbrechen)
  person = "Unbekannt";
}

```

(iii) Umwandlung von Parameterstrings in Standarddatentypen

Die übergebenen Strings aus (i) und (ii) müssen dann in einen int- oder einen double-Wert umgewandelt werden (→ die Exception NumberFormatException muß mit try- und catch-Block abgefangen werden).

vgl. (i), Java-Applikation:

```

int anzahlJahre;
double prozente;
try
{
    anzahlJahre = Integer.parseInt(argv[1]);
    prozente = Double.valueOf(argv[2]).doubleValue();
}
catch (NumberFormatException exc)
{
    System.err.println("Fehler bei der Umwandlung aufgetreten!");
    anzahlJahre = 0;
    prozente = 0.0;
}

```

vgl. (ii), Java-Applet:

```

int anzahlJahre;
double prozente;
try
{
    anzahlJahre = Integer.parseInt(zinsjahre);
    preisEinheit = Double.valueOf(zinsprozente).doubleValue();
}
catch (NumberFormatException exc)
{
    System.err.println("Fehler bei der Umwandlung aufgetreten!");
    anzahlJahre = 0;
    prozente = 0.0;
}

```

- (iv) Umwandlung von Standarddatentypen in Parameterstrings
 Bei Aufruf vieler Methoden von GUI-Komponenten in Applets muß der Parameter ein String sein, d.h. es muß zunächst eine Umwandlung in einen String erfolgen.

```

int anz = 10;
g.drawString("Anzahl: " + n, 30, 50);

```

- (v) Abfrage von Texteingaben in Applets
 Die Abfrage von Texteingaben in `TextField`- bzw. `TextArea`-Objekten erfolgt durch Einlesen des Strings mit der Methode `getText()` der gemeinsamen Oberklasse `TextComponent`. Dieser muß ggf. wieder in einen Standarddatentyp umgewandelt werden.

```

// fieldAnzahl hat Textvorgabe "0" und 4 als max. Anzahl der Zeichen
TextField fieldAnzahl = new TextField("0", 4);

```

```

TextField fieldPreis = new TextField("1.99", 10);
...
String eingabeAnzahl = fieldAnzahl.getText();
String eingabePreis = fieldPreis.getText();

int anz;
double preisEinheit;
try
{
    anz = Integer.parseInt(eingabeAnzahl);
    preisEinheit = Double.valueOf(eingabePreis).doubleValue();
}
catch (NumberFormatException exc)
{
    System.err.println("Fehler bei der Umwandlung aufgetreten!");
    anz = 0;
    preisEinheit = 0.0;
}

```

(vi) Änderung von Text in GUI-Komponenten des Applets

Auch die Änderung von Text in `TextField`- bzw. `TextArea`-Objekten erfolgt durch Übergabe eines Strings mit der Methode `setText()` der gemeinsamen Oberklasse `TextComponent`.

```

TextField fieldSumme = new TextField("0.0", 4);

double summe = anz*preisEinheit;
fieldSumme.setText(Double.toString(summe));

```

(vii) Angabe von Labels in `Frame`, `Button`, ...

viele GUI-Komponenten besitzen Labels oder Namen, die meistens als Strings realisiert sind

```

Button button_ok = new Button("OK");
Frame newWindow = new Frame("Neues Fenster");
..

```

(viii) Eingabe von Zeilen/Werten in Applikationen

vgl. Vorlesungsfile "`SwitchCountdown.java`"

Die Eingabe von Zeilen/Werten in Applikationen erfolgt über Eingabeströme (input streams). Komfortablere Eingabeströme können nicht nur byteweise einlesen, sondern auch ganze Zeilen. Bei Objekten der Klasse "`BufferedReader`" gibt es eine Methode "`readLine()`", die eine eingelesene Zeile als String zurückgibt.

```

BufferedReader is = new BufferedReader(
    new InputStreamReader(System.in));

String antw = "";
int sekunden = 0;
try
{
    System.out.print("Anzahl der Sekunden vor dem Countdown: ");
    antw = is.readLine();
    is.close();
    sekunden = Integer.valueOf(antw).intValue();
}
catch (IOException exc)
{
    System.err.println("Exception abgefangen: " + exc);
}
}

```

1.11 * Unterschied: Objekte anlegen mit Konstruktoren und Abfrage/Neubesetzung der Verweise

Das Erzeugen eines neuen Objekts geschieht in Java stets mit dem Operator "new", dem ein Konstruktoraufruf folgt.

Konstruktoren sind Klassenmethoden, die *keinen* Rückgabetyt vereinbaren und deren Methodennamen der Klassenname selbst ist.

Nicht verwechseln mit der Erzeugung neuer Objekte durch einen Konstruktor sollte man das Ermitteln bereits (irgendwo/irgendwann vorher) erzeugter Objekte. get-Methoden liefern bereits erzeugte Objekte.

...

```

public class MeinApplet extends Applet
{
    public void init()
    {
        // dim wird nicht neu erzeugt, Groessenangabe des Applets
        // (genauer des Containers) wird ermittelt

        Dimension dim = getSize();
        ...
    }

    // der Graphikkontext g wird auch nicht neu erzeugt
    public void paint(Graphics g)
    {
        // noch kein Objekt neu erzeugt, nur eine Referenz angelegt
    }
}

```

```

    Color hintergr;

    // aktuelle Zeichenfarbe mit abgefragt
    hintergr = g.getColor();

    ...

    // kein neues Objekt erzeugt, Color.red ist bereits vordef. Farbe
    // Verweise werden gleichgesetzt
    Color rot = Color.red;
}
}

```

In der Dokumentation zur Klasse `Color` im Paket `java.awt` finden wir 3 verschiedene Konstruktoren (Methodenname = Klassenname = `Color`):

```

public Color(int r, int g, int b) // aus Rot-, Gruen- und Blauanteilen erzeugte
                                // (int-Werte zwischen 0 und 255)

public Color(int rgb)           // aus einem RGB-Wert (int) erzeugte Farbe

                                // wie 1. Konstruktor, nur sind die Werte hier
                                // zwischen 0.0 und 1.0 (also r/255.0, g/255
public Color(float r, float g, float b)

```

Allen Konstruktoren fehlt ein Datentyp für die Rückgabe, alle Namen der Konstruktor-
methoden gleichen dem Klassennamen `Color`.

Erzeugen von Objekten der Klasse `Color`:

```

// kein Aufruf eines Konstruktors
// die Hintergrundfarbe (z.B. des Applets) wird ermittelt
// die Methode getBackground() (der Klasse Component)
// liefert eine Referenz auf ein Color-Objekt, das bereits
// erzeugt wurde und dessen Referenz in farbe gespeichert wird
// farbe ist daher KEIN neues Color-Objekt
Color farbe = getBackground();

// kein Aufruf eines Konstruktors, rot_1 ist KEIN neues Objekt
// Objektvariable rot_1 erhaelt Verweis auf in Klasse Color
// bereits definierte Farbe Color.red
Color rot_1 = Color.red;

// neues Objekt rot_2 wird erzeugt
// verwendeter Konstruktor: public Color(int r, int g, int b)
Color rot_2 = new Color(255, 0, 0);

```

```

// neues Objekt rot_3 wird erzeugt
// verwendeter Konstruktor: public Color(int rgb)
Color rot_3 = new Color(-65536);

// neues Objekt rot_4 wird erzeugt
// verwendeter Konstruktor: public Color(float r, float g, float b)
Color rot_4 = new Color(1.0, 0.0, 0.0);

```

Alle Farbobjekte repräsentieren übrigens dieselbe Farbe `Color.red`.
Anwendung in einem Applet:

```

public void paint(Graphics g)
{
    Dimension dim = getSize();
    int breite = dim.width;
    int hoehe = dim.height;

    Color farbe = getBackground();

    Color rot_1 = Color.red;
    g.setColor(rot_1);
    g.fillRect(0, 0, breite/4 - 1, hoehe - 1);

    int red_value = rot_1.getRed();
    int green_value = rot_1.getGreen();
    int blue_value = rot_1.getBlue();

    Color rot_2 = new Color(red_value, green_value, blue_value);
    g.setColor(rot_2);
    g.fillRect(breite/4, 0, breite/4 - 1, hoehe - 1);

    int rgb_value = rot_1.getRGB();

    Color rot_3 = new Color(rgb_value);
    g.setColor(rot_3);
    g.fillRect(breite/2, 0, breite/4 - 1, hoehe - 1);

    float red_percent = rot_1.getRed()/255.0f;
    float green_percent = rot_1.getGreen()/255.0f;
    float blue_percent = rot_1.getBlue()/255.0f;

    Color rot_4 = new Color(red_percent, green_percent, blue_percent);
    g.setColor(rot_4);
    g.fillRect(3*breite/4, 0, breite/4 - 1, hoehe - 1);
}

```

1.12 * Wahl des Ortes für Objektdefinitionen

Jedes Objekt besitzt einen Datentyp. Will man ein Objekt neu erzeugen oder eine bereits vorhandene Referenz in der Objektvariable erzeugen, muß man zuerst eine Objektvariable definieren. Dies erfolgt immer nach dem Schema

```
Datentyp Objektvariablenname;
Datentyp Objektvariablenname = new Konstruktoraufruf;
Datentyp Objektvariablenname = Referenz_auf_Objekt;
```

Stringobjekte (oder andere Objekte und Variablen von Standarddatentypen) werden entweder in Funktionen oder in Klassen definiert.

a) Definition innerhalb von Funktionen:

Dazu werden die Definitionen der Stringobjekte nach dem Funktionsnamen und zwischen '{' und '}' eingefügt.

```
public class Test
{
    public static void main(String [] argv)
    {
        String s1 = new String("Java");    // s1 erhaelt den Text 'Java'
        String s2 = new String("");        // s2 ist der leere String

        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
    }
}
```

Abkürzend kann man schreiben:

```
public class Test
{
    public static void main(String [] argv)
    {
        // s1 UND s2 sind Strings
        String s1 = new String("Java"),
            s2 = new String("");    // s1 erhaelt den Text "Java"
                                    // s2 ist der leere String

        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
    }
}
```

In Applets fügt man die Stringdefinition z.B. in die Methode "paint()" ein.

```
import java.applet.Applet;
```

```

import java.awt.Graphics;

public class StringApplet_1 extends Applet
{
    public void paint(Graphics g)
    {
        String titel = "Kurzer Titel";
        g.drawString(titel, 20, 20);
    }
}

```

Da die Methode "drawString()" als 1. Argument ein Stringobjekt erwartet, kann man statt dem konstanten String "Kurzer Titel" auch den variablen String "titel" an die Methode übergeben.

b) Definition außerhalb von Funktionen als Datenelemente:

Oftmals bietet es sich an, Stringobjekte außerhalb den Methoden der Klasse, aber innerhalb der Klasse zu definieren. Damit werden diese Stringobjekte zu Datenelementen der Klasse, die in allen (objektabhängigen) Methoden der Klasse verfügbar sind. Das hat den Vorteil, daß man jetzt in allen Methoden auf die Stringobjekte zugreifen kann.

1. Beispiel: kein Datenelement

```

import java.applet.Applet;
import java.awt.Graphics;

public class StringApplet_1 extends Applet
{
    public void paint(Graphics g)
    {
        // titel ist KEIN Datenelement der Klasse StringApplet_1
        String titel = new String("Kurzer Titel");
        g.drawString(titel, 20, 20);
    }
}

```

2. Beispiel: ein Datenelement text

```

import java.applet.Applet;
import java.awt.*;          // fuer Color und Graphics

public class StringApplet_2 extends Applet
{
    // text ist ein Datenelement der Klasse StringApplet_2
    String text = new String("wenig Neues");

    public void init()
    {

```

```

// richtig: text ist in allen Methoden vorhanden
System.out.println("Es gibt " + text + "!");

// Fehler: titel nur in paint() definiert
// System.out.println("Titel: " + titel);
}

public void paint(Graphics g)
{
    // titel ist KEIN Datenelement der Klasse StringApplet_2
    String titel = new String("Kurzer Titel");
    g.drawString(titel, 20, 20);

    // setzt die Farbe auf Blau dauerhaft um
    // String wird in blau ausgegeben
    g.setColor(Color.blue);

    // richtig: text ist in allen Methoden vorhanden
    g.drawString(text, 20, 100);
}
}

```

3. Beispiel: ein Datenelement text, andere Reihenfolge der Methoden ändert nichts

```

import java.applet.Applet;
import java.awt.*;          // fuer Color und Graphics

public class StringApplet_3 extends Applet
{
    String text = new String("wenig Neues");

    public void paint(Graphics g)
    {
        String titel = new String("Kurzer Titel");
        g.drawString(titel, 20, 20);

        // setzt die Farbe auf Blau dauerhaft um
        // String wird in blau ausgegeben
        g.setColor(Color.blue);

        // richtig: text ist in allen Methoden vorhanden
        g.drawString(text, 20, 100);
    }

    public void init()
    {

```

```

    // richtig: text ist in allen Methoden vorhanden
    System.out.println("Es gibt " + text + "!");

    // Fehler: titel nur in paint() definiert
    // System.out.println("Titel: " + titel);
}
}

```

4. Beispiel: eine Objektvariable `text` wird zunächst erzeugt, die eine Referenz auf ein Objekt speichern kann

```

import java.applet.Applet;
import java.awt.*;          // fuer Color und Graphics

public class StringApplet_4 extends Applet
{
    // text ist Objektvariable, die eine Referenz auf ein
    // String-Objekt speichern kann
    // Dadurch wird noch KEIN Objekt erzeugt!
    String text;

    public void init()
    {
        // an dieser Stelle wird ein neues Objekt erzeugt
        // und die Referenz darauf in text abgelegt
        text = new String("wenig Neues");

        // richtig: text ist in allen Methoden vorhanden
        System.out.println("Es gibt " + text + "!");

        // Fehler: titel nur in paint() definiert
        // System.out.println("Titel: " + titel);
    }

    public void paint(Graphics g)
    {
        String titel = new String("Kurzer Titel");
        g.drawString(titel, 20, 20);

        // setzt die Farbe auf Blau dauerhaft um
        // String wird in blau ausgegeben
        g.setColor(Color.blue);

        // richtig: text ist in allen Methoden vorhanden
        g.drawString(text, 20, 100);
    }
}

```


2 Grafik mit Applets

2.1 * Ändern der Zeichenfarbe

Einige Farben sind in der von Sun vordefinierten Klasse `Color` im Paket `java.awt` als klassenabhängige, konstante Datenelemente abgespeichert (siehe Dokumentation).

Beispiele:

- Grundfarben: `white`, `black`, `blue`, `green`, `red`, `yellow`
- Mischfarben: `cyan`, `magenta`, `orange`, `pink`
- Grautöne: `gray`, `lightGray`, `darkGray`

Aufgrund ihrer Definition als Datenelement der Klasse `Color` in der Form

```
public static final Color white;
```

(öffentlich zugänglich, klassenabhängig, konstant) wird die Farbe weiß in der Form

```
Color.white
```

in eigenen Java-Programmen angesprochen.

Andere Farben können durch spezielle Konstruktoren (Erzeugungsmethoden) z.B. durch ihren RGB-Wert (Red-Green-Blue-Farbenmodell) erzeugt werden.

Wichtige Methoden zum Zeichnen in ihrer Verwendung in der Methode `public void paint(Graphics g)`:

- Umstellen der aktuellen Zeichenfarbe auf rot für alle folgenden Zeichenoperationen:

```
g.setColor(Color.red);  
g.drawString("Warnung!", 20, 40); // String wird in ROT ausgegeben  
g.drawString("Gefahr!", 20, 90); // auch dieser String wird in ROT ausgegeben
```

- Abfrage der aktuellen Zeichenfarbe:

```
Color farbe = g.getColor();
```

- Setzen der Hintergrundfarbe auf grau:

```
setBackground(Color.gray);
```

Beachte: kein führendes "g."

- Abfrage der Hintergrundfarbe:

```
Color bg_farbe = getBackground();
```

- Setzen der Vordergrundfarbe auf schwarz:

```
setForeground(Color.black);
```

- Abfrage der Vordergrundfarbe:

```
Color fg_farbe = getForeground();
```

Soll eine der Methoden `getColor()` bzw. `setColor()` außerhalb der Methode `paint()` verwendet werden, muß zunächst der Grafikkontext durch Aufruf der Methode `getGraphics()` der GUI-Komponente ermittelt werden.

```
public void called_by_paint()
{
    ...
    Graphics g_momentan = getGraphics();
    g_momentan.setColor(Color.red);
    ...
}
```

Vorsicht: Es ist nicht immer sichergestellt, daß Zeichenbefehle außerhalb der `paint()`-Routine überhaupt ausgeführt werden. Einstellungen wie Hintergrund-/Vordergrundfarben kann man dagegen auch in anderen Methoden mit `getGraphics()` verwenden.

2.2 * Zeichenobjekte

Die Klasse `Graphics` im Paket `java.awt` stellt viele Operationen zum Zeichnen von Grafikobjekten zur Verfügung. Die ersten beiden Parameter `x` und `y` sind immer die Anfangskordinaten:

- Punkt $(x, y) = (20, 40)$
Ein Punkt muß z.B. als Strecke mit gleichem Anfangs- und Endpunkt gezeichnet werden.

```
g.drawLine(20, 40, 20, 40);
```

- Strecke/Linie/Geradenstück
Eine Strecke ist immer 1 Pixel breit, dickere Striche müssen durch mehrere Strecken emuliert werden.
waagrechte Linie von $(20, 40)$ nach $(100, 40)$ mit Länge 80:

```
g.drawLine(20, 40, 100, 40);
```

senkrechte Linie von (20, 40) nach (20, 80) mit Länge 40:

```
g.drawLine(20, 40, 20, 80);
```

Diagonale durch Appletbereich:

```
Dimension dim = getSize();  
g.drawLine(0, 0, dim.width-1, dim.height-1);
```

- Kreis/Oval/Ellipse

Ein Oval wird angegeben durch das zugehörige umschriebene Rechteck, d.h. durch linke obere Ecke und die Breite und Höhe dieses Rechtecks. Es wird rechts und unten 1 Pixel mehr gezeichnet als der 3. und 4. Parameter angeben.

Kreis: linke obere Ecke (20, 40), 50 Pixel breites und hohes Oval

```
g.drawOval(20, 40, 49, 49);
```

Kreis mit Radius 50 und Mittelpunkt (100, 70):

```
g.drawOval(100-50, 70-50, 2*50-1, 2*50-1);
```

abgeflachtes Oval (Breite 200, Höhe 10):

```
g.drawOval(20, 40, 199, 9);
```

hohes, schmales Oval (Breite 10, Höhe 200):

```
g.drawOval(20, 40, 9, 199);
```

Varianten sind ein ausgefülltes Oval. Bei der Ausfüllung werden im Unterschied zu oben genau die angegebene Pixelbreite und -höhe verwendet (da die `fill`-Befehle rechts und unten automatisch ein Pixel weniger zeichnen!).

```
// ausgefuelltes Oval (Breite 10, Hoehe 200)  
g.fillOval(20, 40, 10, 200);
```

Durch die verschiedenen Bedeutungen der Höhe und Breite bei Zeichnung des Randes und des ausgefüllten Objekts ergibt sich folgendes:

```

// roter Kreisrand
g.setColor(Color.red);
g.drawOval(20, 40, 49, 49);
// gruenes Kreisinneres (rechts und unten bleibt Rand bestehen)
g.setColor(Color.green);
g.fillOval(20, 40, 49, 49);

// blauer Kreisrand
g.setColor(Color.blue);
g.drawOval(100, 120, 49, 49);
// gelbes Kreisinneres (ueberall bleibt Rand stehen)
g.setColor(Color.yellow);
g.fillOval(100+1, 120+1, 48, 48);

```

- Teil eines Kreisrandes/Ovalrandes/Kreissegment/Ovalbereich
Die ersten vier Parameter sind wie bei `drawOval(...)`, der fünfte gibt den Startwinkel in Grad an, der sechste nicht den Endwinkel, sondern den zu zeichnenden Winkelbereich (auch in Grad).

```

// Rand des oberen Halbkreises
// gezeichnet: ab 0 Grad 180 Grad zeichnen, also von 0 bis 180 Grad
g.drawArc(20, 40, 49, 49, 0, 180);

// Rand des unteren Halbkreises
// gezeichnet: ab 180 Grad 180 Grad zeichnen, also von 180 bis 360 Grad
g.drawArc(20, 40, 49, 49, 180, 180);

// Teil des Randes des Ovals (Breite 10, Hoehe 200)
// gezeichnet: ab 45 Grad bis 180 = 135 + 45 Grad
g.drawArc(20, 40, 9, 199, 45, 135);

// oberer Halbkreis (ausgefuehlt)
g.fillArc(20, 40, 50, 50, 0, 180);

// unterer Halbkreis (ausgefuehlt)
g.fillArc(20, 40, 50, 50, 180, 180);

// oder:
g.fillArc(20, 40, 50, 50, 0, -180);

// ausgefuehlter Ovalbereich (Breite 10, Hoehe 200)
// gezeichnet: ab 45 Grad bis 180 = 135 + 45 Grad
g.fillArc(20, 40, 10, 200, 45, 135);

```

- Rechteck/Viereck
Ein Rechteck wird angegeben durch die linke obere Ecke und die Breite und Höhe

dieses Rechtecks. Es wird rechts und unten 1 Pixel mehr gezeichnet als der 3. und 4. Parameter angeben.

Quadrat: linke obere Ecke (20, 40), 50 Pixel breit (und auch hoch)

```
g.drawRect(20, 40, 49, 49);
```

Quadrat: Mittelpunkt (70, 90), Breite (= Höhe): 100

```
g.drawRect(70 - 50, 90 - 50, 99, 99);
```

abgeflachtes Rechteck (Breite 200, Höhe 10):

```
g.drawRect(20, 40, 199, 9);
```

hohes, schmales Rechteck (Breite 10, Höhe 200):

```
g.drawRect(20, 40, 9, 199);
```

Varianten sind ein ausgefülltes Rechteck, ein Rechteck mit abgerundeten Ecken (Rand/ausgefüllt) und ein Rechteck mit 3D-Effekt (Rand/ausgefüllt), das entweder hervortritt oder eingesunken ist. Bei der Ausfüllung werden im Unterschied zu oben genau die angegebene Pixelbreite und -höhe verwendet.

```
// ausgefülltes Rechteck (Breite: 10, Höhe: 200)
g.fillRect(20, 40, 10, 200);
```

```
// abgerundeter Rand eines Rechtecks (Breite: 50, Höhe: 200)
// Abrundung: 2*5 = 10 Pixel breit, 2*10 = 20 Pixel hoch,
g.roundRect(20, 40, 49, 199, 5, 10);
```

```
// abgerundetes ausgefülltes Rechteck (Breite: 50, Höhe: 200)
// Abrundung: 2*5 = 10 Pixel breit, 2*10 = 20 Pixel hoch,
g.fillRoundRect(20, 40, 50, 200, 5, 10);
```

```
// Rand eines hervorgehobenen Rechtecks (Breite: 50, Höhe: 200)
g.draw3DRect(20, 40, 49, 199, true);
```

```
// Rand eines eingesunkenen Rechtecks (Breite: 50, Höhe: 200)
g.draw3DRect(20, 40, 49, 199, false);
```

```
// ausgefülltes hervorgehobenes Rechtecks (Breite: 50, Höhe: 200)
g.fill3DRect(20, 40, 50, 200, true);
```

```
// ausgefuelltes eingesunkenes Rechtecks (Breite: 50, Hoehe: 200)
g.fill3DRect(20, 40, 50, 200, false);
```

- Polygon (Vieleck)

Ein Polygon wird durch Angabe aller seiner Ecken gezeichnet, es werden alle Kanten gekennzeichnet, insbesondere wird auch die Kante vom letzten zum ersten Punkt automatisch gezeichnet. Zum Abspeichern aller Ecken kann die Klasse Polygon im Paket `java.awt` genutzt werden. Damit wird das Polygon aber noch nicht gezeichnet, dies geschieht erst durch Aufruf von `"g.drawPolygon()"`.

```
// leeres Polygon, d.h. Polygon hat keine Ecken
Polygon p = new Polygon();

// 1. Ecke wird hinzugefuegt
p.addPoint(20, 100);

// 2. Ecke wird hinzugefuegt
p.addPoint(40, 100);

// 3. Ecke wird hinzugefuegt
p.addPoint(40, 30);

// Polygon mit 3 Ecken (= Dreieck) wird gezeichnet
g.drawPolygon(p);
```

Ein Polygon kann auch durch Angabe aller x- und y-Koordinaten und der Anzahl der Ecken gezeichnet werden.

```
// Array der x-Koordinaten fuer Polygon
int [] x_koords = { 20, 40, 40};
// Array der y-Koordinaten fuer Polygon
int [] y_koords = { 100, 100, 30};

g.drawPolygon(x_koords, y_koords, 3);
```

Varianten: ausgefülltes Polygon mit `"g.fillPolygon()"` und Linienzug (Kante zwischen letzter und erster Ecke entfällt) durch `"g.drawPolyLine()"`

2.3 * Flacker-/redraw-Problematik

Wie im Lebenszyklus eines Applets (Abschnitt 0.8) beschrieben, wird die `paint()`-Methode u.U. sehr häufig aufgerufen. Wie oft, entscheidet die Java Virtual Machine (also die Java-Ausführungsumgebung) selbst. Man sollte grundsätzlich vermeiden, die Appletmethode

`paint()` selbst aufzurufen (bis auf eine Ausnahme, siehe unten) und keine langen Berechnungen und Programmteile in die `paint()`-Methode aufnehmen.

Statt die Appletmethode `paint()` selbst aufzurufen, sollte man immer die Appletmethode

```
public void repaint()
```

bzw.

```
public void repaint(long pause_in_milli_sek)
```

aufzurufen, die nur den *Wunsch* an die JVM übermittelt, doch baldmöglichst die `paint()`-Methode aufzurufen. Standardmäßig macht die Appletmethode `repaint()` folgendes: Sie ruft die Appletmethode

```
public void update(Graphics g)
```

auf, diese löscht den gesamten Bildschirm und ruft die `paint()`-Methode selbst auf. Werden zu schnell hintereinander `repaint`-Anweisungen gegeben, kann es u.U. zum Bildschirmflackern kommen. In diesem Fall sollte man selbst die `update()`-Methode schreiben und z.B. nur einen Teil des Bildschirms löschen, das Bildschirmlöschen intelligenter lösen (z.B. alle bisher gezeichneten Sachen in der Hintergrundfarbe zeichnen, so dass diese auch gelöscht werden) oder das Bildschirmlöschen ganz unterlassen. Allerdings muss man darauf achten, dass beim Überdecken des Appletfensters mit einem anderen Fenster und dem Wiedersichtbarmachen des Appletfensters auch automatisch `paint()` über `update()` aufgerufen wird. Die eigene `update()`-Methode muss sicherstellen, dass auch wirklich alles wieder gezeichnet wird, was kurzzeitig verdeckt wurde.

Beispiel 2.1 *In der Methode `run()` werden ständig Koordinaten `x1`, `y1`, `x2` und `y2` sowie eine Farbe `farbe` zufällig berechnet und dann soll ein ausgefülltes Rechteck mit diesen Koordinaten/Größen sowie in der Farbe gezeichnet werden.*

Sourcecode:

```
...

public class Flackern extends Applet ...
{
    // Datenelemente
    int breite, hoehe, x1, x2, y1, y2;
    Color farbe;
    ...

    public void run()
    {
        int rot, gruen, blau;

        // Endlosschleife
```

```

while (true)
{
    // zufaellige Koordinaten/Groessen bestimmen
    x1 = (int) (Math.random()*(breite + 1)); // breite: Breite des Applets
    y1 = (int) (Math.random()*(hoehe + 1)); // breite: Hoehe des Applets
    x2 = (int) (Math.random()*(breite + 1));
    y2 = (int) (Math.random()*(hoehe + 1));
    ...
    // zufaellige RGB-Werte bestimmen
    rot = (int) (Math.random()*256);
    gruen = (int) (Math.random()*256);
    blau = (int) (Math.random()*256);

    farbe = new Color(rot, gruen, blau);

    // neues Zeichnen sofort anfordern
    // repaint();

    // neues Zeichnen nach 3 Sekunden = 3000 Millisekunden anfordern
    repaint(3000);
}
}

// zufaelliges Rechteck in zufaelliger Farbe zeichnen
public void paint(Graphics g)
{
    g.setColor(farbe);
    g.fillRect(x1, y1, x2, y2);
}
}
}

```

Die Methode `repaint()` wird in der Methode `run()` aufgerufen (und nicht `paint()` selbst!). U.U. kommt es aber zu Bildschirmflackern.

Beispiel 2.2 Das Applet gibt an vier Stellen eines Gitters Strings aus und zeichnet ein Gitter. Dabei wechselt die Farbe ständig von grün zu rot und dann wieder zu grün. Hier kann man einfach das Bildschirmlöschen ganz unterlassen, weil einfach in der neuen Farbe über die alte Farbe das Gitter gezeichnet werden kann.

Sourcecode:

```

...

public class Flackern_3 extends Applet ...
{
    // Datenelemente
    int breite, hoehe;
    boolean gruen;

```

```

...

public void init()
{
    // beginne mit gruener Farbe
    gruen = true;
    ...
}

public void run()
{
    // Endlosschleife
    while (true)
    {
        // schalte gruen ab oder an
        gruen = !gruen;
        // neues Zeichnen sofort anfordern
        // repaint();

        // neues Zeichnen nach halber Sekunde = 500 Millisekunden anfordern
        repaint(500);
    }
}

// zufaelliges Rechteck in zufaelliger Farbe zeichnen
public void paint(Graphics g)
{
    // Strings in Gitterfeldern ausgeben
    // breite = Appletbreite, hoehe = Applethoehe

    g.setColor(Color.black);
    g.drawString("Java", breite/4, hoehe/4);
    g.drawString("JDK", 3*breite/4, hoehe/4);
    g.drawString("Sun", breite/4, 3*hoehe/4);
    g.drawString("Coffee", 3*breite/4, 3*hoehe/4);

    // zwischen gruen und rot als Zeichenfarbe umschalten
    if (gruen)
        g.setColor(Color.green);
    else
        g.setColor(Color.red);

    // horizontaler und vertikaler Strich
    g.drawLine(breite/2, 0, breite/2, hoehe-1);
    g.drawLine(0, hoehe/2, breite-1, hoehe/2);
}

```

```

// eigene Update-Methode, vermeidet das Bildschirmloeschen
// und ruft paint() auf
public void update(Graphics g)
{
    // vermeide Bildschirmloeschen
    // g.clearRect(0, 0, breite, hoehe);

    paint(g);
}
}

```

In der Standardimplementierung führt die Appletmethode `update()` das Bildschirmlöschen durch den Befehl

```
g.clearRect(0, 0, breite, hoehe);
```

aus. Durch die eigene Implementierung wird dies aber genau verhindert, so dass es zu weniger Bildschirmflackern kommt.

Beispiel 2.3 *In einem Applet soll abwechselnd ein roter Kreis und darunter die Beschriftung "Rot" bzw. ein grüner Kreis mit der Beschriftung "Grün" ausgegeben werden. Um das Bildschirmflackern zu vermeiden, kann man in der eigenen Update-Methode nicht den gesamten Bildschirm löschen, sondern einfach die alte Beschriftung in der Hintergrundfarbe nochmals ausgeben (dadurch wird sie natürlich gelöscht).*

Sourcecode:

```

...

public class Flackern_7 extends Applet ...
{
    // Datenelemente
    int breite, hoehe;
    boolean gruen;
    ...

    public void init()
    {
        // beginne mit gruener Farbe
        gruen = true;
        ...
    }

    public void run()
    {
        // Endlosschleife
        while (true)

```

```

{
    // schalte gruen ab oder an
    gruen = !gruen;
    // neues Zeichnen sofort anfordern
    // repaint();

    // neues Zeichnen nach halber Sekunde = 500 Millisekunden anfordern
    repaint(500);
}
}

// zufaelliges Rechteck in zufaelliger Farbe zeichnen
public void paint(Graphics g)
{
    // Strings in Gitterfeldern ausgeben
    // breite = Appletbreite, hoehe = Applethoehe

    g.setColor(Color.black);

    if (gruen)
    {
        // Beschriftung Gruen ausgeben
        g.drawString("Gr\u00FCn", breite/2, hoehe-1);
        // gruenen Kreis zeichnen
        g.setColor(Color.green);
        g.fillOval(breite/4, hoehe/4, breite/2, hoehe/2);
    }
    else
    {
        // Beschriftung Rot ausgeben
        g.drawString("Rot", breite/2, hoehe-1);
        // roten Kreis zeichnen
        g.setColor(Color.red);
        g.fillOval(breite/4, hoehe/4, breite/2, hoehe/2);
    }
}

// eigene Update-Methode, vermeidet das Bildschirmloeschen
// und ruft paint() auf
public void update(Graphics g)
{
    // vermeide Bildschirmloeschen
    // g.clearRect(0, 0, breite, hoehe);

    // aktuelle Hintergrundfarbe wird Zeichenfarbe -> Loeschen
    g.setColor(getBackground());
}

```

```

    if (!gruen)
    {
        // Beschriftung Gruen in weiss ausgeben
        g.drawString("Gr\u00FCn", breite/2, hoehe-1);
    }
    else
    {
        // Beschriftung Rot in weiss ausgeben
        g.drawString("Rot", breite/2, hoehe-1);
    }
    paint(g);
}
}

```

Der Kreis muss nicht gelöscht werden, weil er in der genauen Position wie vorher in einer anderen Farbe darüber gezeichnet wird.

*** Threads/parallele Prozesse**

Um zu vermeiden, dass in der paint()-Methode zulange Berechnungen stattfinden, kann man sog. Threads in Java verwenden (Threads dienen zur Parallelisierung). Dabei kann man folgenden Blackbox-Rahmen verwenden:

```

...
public class Flackern_7 extends Applet
    implements Runnable // zugew. Interface fuer Threads
{
    // Datenelement vom Typ Thread
    Thread prozess;

    // Threads werden normalerweise in der Appletmethode start() gestartet
    public void start()
    {
        // neuer Prozess wird angelegt, die Methode run() von this
        // (eigenes Appletobjekt) wird durch prozess.start() gestartet
        prozess = new Thread(this);
        // Start des Prozesses
        prozess.start();
    }

    // Threads werden normalerweise in der Appletmethode stop() gestoppt
    public void stop()
    {
        // Prozess wird gestoppt, indem man Referenz auf null setzt
        prozess = null;
    }
}

```

```

// Interface-Methode, die durch prozess.start() aufgerufen wird
public void run()
{
    // momentaner Thread muss ermittelt werden
    Thread laufender_prozess = Thread.currentThread();

    // falls momentaner Thread der gestartete Thread ist
    if (laufender_prozess == prozess)
    {
        // Endlosschleife
        while (true)
        {
            gruen = !gruen;
            repaint(pause);
        }
    }
}
}

```

Durch das Implementieren des Interfaces `Runnable` verspricht man, die Methode

```
public void run()
```

in der eigenen Appletklasse zu schreiben. Diese wird dann durch die Methode

```
public void start()
```

der Klasse `Thread` im Paket `java.lang` gestartet. Prozesse werden in der Appletmethode

```
public void start()
```

gestartet und in der Appletmethode

```
public void stop()
```

gestoppt (siehe Lebenszyklus eines Applets).

Die Interface-Methode `run()` kann durchaus Endlosschleifen beinhalten, da nicht das gesamte Applet diese Methode abarbeitet, sondern nur ein Thread ("Ausführungsfaden" des Applets). Die JVM entscheidet selbstständig, wieviel Rechenzeit es für diese Methode und wieviel für die anderen Tätigkeiten (Mouseabfrage, `repaint`, ...) verwendet. Alle obigen Beispiele wurden so implementiert und in der `run()`-Methode wurde die entsprechende Endlosschleife programmiert.

2.4 * typische Vorgehensweise

Soweit es geht, sollte man in der `init()`-Methode alle zu zeichnenden Grafikobjekte vorbereiten. Dazu definiert man diese als Datenelemente des Applets und nicht als lokale Variablen/Objekte. Falls möglich, gruppiert man gleiche Zeichenobjekte und speichert diese in Arrays. In `init()` erzeugt man diese (bzw. das Array und dann einzeln für jeden Index die Zeichenobjekte) und weist ihnen sinnvolle Startwerte zu. In der `paint(...)`-Methode werden dann diese Objekte gezeichnet.

```
...

public class Graphics_RGB_3 extends Applet
{
    // Datenelemente des Applets

    Color [] farben;
    Rectangle [] rechtecke;

    // Zeichenobjekte anlegen
    public void init()
    {
        // Array farben wird erzeugt (Laenge 3, Komponenten: Verweise auf Color)
        farben = new Color[3];

        // Arraykomponenten werden besetzt als Verweise auf vordefinierte Farben
        farben[0] = Color.red;
        farben[1] = Color.green;
        farben[2] = Color.blue;

        // Array rechtecke wird erzeugt (Laenge wie Array farben,
        //                               Komponenten: Verweise auf Rectangle)
        rechtecke = new Rectangle[farben.length];

        // Arraykomponenten werden besetzt als Verweise auf neu erzeugte
        // Objekte der Klasse Rectangle
        rechtecke[0] = new Rectangle(0, 0, 100, 100);
        rechtecke[1] = new Rectangle(100, 100, 100, 100);
        rechtecke[2] = new Rectangle(200, 200, 100, 100);
    }

    // Zeichenobjekte zeichnen
    public void paint(Graphics g)
    {
        for (int i = 0; i < rechtecke.length; i++)
        {
            g.setColor(farben[i]);
        }
    }
}
```

```

        g.fillRect(rechtecke[i].x, rechtecke[i].y,
                   rechtecke[i].width, rechtecke[i].height);
    }
}
}

```

Die Grundobjekte kann man alle in der Methode `init()` vorbereiten:

...

```

public class Graphik_Objekte_4 extends Applet
{
    Point punkt;
    Point linieAnfang, linieEnde;
    Rectangle kreis;
    Rectangle rechteck;
    Rectangle halbkreis;
    Polygon dreieck;
    int [] winkel;

    public void init()
    {
        punkt      = new Point(30, 30);
        linieAnfang = new Point(20, 40);
        linieEnde   = new Point(40, 40);

        kreis      = new Rectangle(30 - 20, 70 - 20, 2*20, 2*20);
        rechteck   = new Rectangle(30 - 20, 140 - 40, 2*20, 2*40);
        halbkreis  = new Rectangle(30 - 20, 210 - 20, 2*20, 2*20);

        dreieck = new Polygon();
        dreieck.addPoint(10, 250);
        dreieck.addPoint(50, 250);
        dreieck.addPoint(30, 280);

        winkel = new int[2];
        winkel[0] = 180;
        winkel[1] = 180;

        setBackground(Color.white);
    }

    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        // Punkt
    }
}

```

```

g.drawLine(punkt.x, punkt.y, punkt.x, punkt.y);
// Linie
g.drawLine(linieAnfang.x, linieAnfang.y, linieEnde.x, linieEnde.y);

// Kreis
g.fillOval(kreis.x, kreis.y, kreis.width, kreis.height);
// Rechteck
g.fillRect(rechteck.x, rechteck.y, rechteck.width, rechteck.height);
// Halbkreis
g.fillArc(halbkreis.x, halbkreis.y, halbkreis.width, halbkreis.height,
          winkel[0], winkel[1]);

// Dreieck
g.fillPolygon(dreieck);
}
}

```

Will man im JDK 1.2 und höheren Versionen die neuen Grafikfähigkeiten ausnutzen (also z.B. die Grafikobjekte in dem Package `java.awt.geom`), castet man den Grafikkontext der `paint`-Methode in einen Verweis auf die Klasse `Graphics2D`. Diese Klasse ist eine Erweiterung von der Klasse `Graphics` und kann wesentlich mehr an Grafikoperationen. Z.B. kann man verschiedene Zeichenobjekte in ein Array der Klasse (genauer: Interface `Shape`) stecken und diese mit einem `fill`-Methode zeichnen.

```

...
import java.awt.geom.*;

public class Ampel_2 extends Applet
{
    Rectangle rahmen;
    Ellipse2D.Double ampelRot;
    Color [] alleFarben;
    Shape [] alleObjekte;

    public void init()
    {
        rahmen = new Rectangle(100, 0, 80, 220);

        double radius = 30.0;
        double [] mp = { 140.0, radius + 10.0 };

        ampelRot = new Ellipse2D.Double(mp[0] - radius, mp[1] - radius,
                                       2*radius, 2*radius);

        alleFarben = new Color[2];
        alleFarben[0] = Color.black;
    }
}

```

```

    alleFarben[1] = Color.red;

    alleObjekte = new Shape[2];
    alleObjekte[0] = rahmen;
    alleObjekte[1] = ampelRot;
}

public void paint(Graphics g)
{
    Graphics2D g2d = (Graphics2D) g;

    for (int i = 0; i < alleObjekte.length; i++)
    {
        g2d.setColor(alleFarben[i]);
        g2d.fill(alleObjekte[i]);
    }
}
}

```

Ändert sich etwas an der Zeichnung und möchte man in einer anderen als der Methode `paint` ein Neuzeichnen erreichen, geht man wie folgt vor: Zuerst führt man ein Daten-element ein, das sich diese Änderung merken kann. Dieses besetzt man in `init()` geeignet vor und ändert es in der oben genannten anderen Methode. Dann fordert man mit `repaint()` ein Neuzeichnen an (ein tatsächliches Neuzeichnen kann die JVM erst zu einem späteren Zeitpunkt durchführen). `repaint()` ruft automatisch die Methode `public void update(Graphics g)` auf, die den Bildschirm löscht und (evtl. nach einer Wartezeit) die `paint`-Methode aufruft. `paint()` selbst wird nicht direkt aufgerufen.

...

```

public class Koordinaten_2 extends Applet implements MouseListener
{
    int x, y;
    String ausgabe;

    public void init()
    {
        // Standardwerte
        x = y = 0;
        ausgabe = "";

        ...

        setBackground(Color.white);
        setForeground(Color.blue);
    }
}

```

```

public void paint(Graphics g)
{
    // Ausgabe des Strings und x, y
    g.drawString(ausgabe, x, y);

public void mousePressed(MouseEvent evt)
{
    // ermittle Mouse-Koordinaten aus MouseEvent-Objekt
    x = evt.getX();
    y = evt.getY();
    ausgabe = "Koordinaten: (x,y) = (" + x + "," + y + ")";

    ...

    repaint();
}

...

}

```

Das automatische Bildschirmlöschen kann zu sog. Bildschirmflattern führen. Es kann durch Selbstschreiben der Methode `public void update(Graphics g)` vermieden werden. Diese (oder eine andere) Methode muss dann selbst für ein geeignetes Löschen von Teilbereichen sorgen.

1. Ansatz: Man merke sich die alten Werte in zusätzlichen Variablen.

```

...

public class Koordinaten_3 extends Applet implements MouseListener
{
    int x, y, x_alt, y_alt;
    String ausgabe, ausgabe_alt;

    public void init()
    {
        // Standardwerte
        x = y = x_alt = y_alt = 0;
        ausgabe = ausgabe_alt = "";

        ...

        setBackground(Color.white);
    }
}

```

```

public void paint(Graphics g)
{
    g.setColor(Color.blue);

    // Ausgabe des Strings und x, y
    g.drawString(ausgabe, x, y);

public void update(Graphics g)
{
    // loesche Bildschirm nicht
    // String ausgabe wird geloescht
    g.setColor(getBackground());
    g.drawString(ausgabe_alt, x_alt, y_alt);

    paint(g);
}

public void mousePressed(MouseEvent evt)
{
    x_alt = x;
    y_alt = y;
    ausgabe_alt = ausgabe;

    x = evt.getX();
    y = evt.getY();
    ausgabe = "Koordinaten: (x,y) = (" + x + "," + y + ")";

    ...

    repaint();
}
...
}

```

2. Ansatz: Nur die Zeichenfarbe wird umgestellt (Hintergrund oder Vordergrund)
Wird die Mouse losgelassen, wird die Zeichenfarbe auf die Hintergrundfarbe gestellt und die Ausgabe gelöscht.

```

...

public class Koordinaten_4 extends Applet implements MouseListener
{
    int x, y;
    String ausgabe;
    Color farbe;

```

```

public void init()
{
    // Standardwerte
    x = y = 0;
    ausgabe = "";
    farbe = Color.blue;

    ...

    setBackground(Color.white);
}

public void paint(Graphics g)
{
    g.setColor(farbe);

    // Ausgabe des Strings und x, y
    g.drawString(ausgabe, x, y);

public void mousePressed(MouseEvent evt)
{
    // ermittele Mouse-Koordinaten aus MouseEvent-Objekt
    x = evt.getX();
    y = evt.getY();
    ausgabe = "Koordinaten: (x,y) = (" + x + "," + y + ")";

    ...

    farbe = Color.blue;
    repaint();
}
public void mouseReleased(MouseEvent evt)
{
    farbe = getBackground();
    repaint();
}
...
}

```

2.5 * Beispielprogramme

Testprogramme:

Graphics_Ecken.java

Graphics_Ecken_2.java

Graphics_Ecken_3.java
Graphics_Gitter.java
Graphics_Gitter_Back.java
Graphics_RGB.java
Graphics_RGB_2.java
Graphics_Rechteck.java
Graphics_Rechteck.java
GraphikDemo_11.java
Graphik_Objekte_1.java
Graphik_Objekte_2.java
Koordinaten.java
Kreise_1.java
Kreise_2.java
Kreise_3.java
VerkehrsZeichen.java
Graphics.html
Koordinaten.html
Kreise.html
VerkehrsZeichen.html

3 * GUI-Programmierung

3.1 * Grundlagen

vgl. zusätzlich die Vorlesung

Die Oberflächenprogrammierung ist in Java im Gegensatz zu Sprachen wie C, C++, Pascal, FORTRAN, ... über Betriebssystem- und Rechnergrenzen standardisiert. Das **AWT** (AWT = Abstract Windows Toolkit) befreit die Programmiererin/den Programmierer davon, wie die Komponenten des **GUI** (GUI = Graphical User Interface) konkret auf dem Betriebssystem mit dem momentanen Rechner umzusetzen sind. Die Benutzung von Komponenten des AWT bedeutet einen internen Aufruf von den zugehörigen Methoden der entsprechenden Peer-Komponente, die die Zwischenklasse zwischen Java und dem jeweiligen Windows-System (Windows oder X-Windows/Motif) darstellt. Diese sind dann durch die Java-Portierung auf das jeweilige Betriebs- und Windowssystem anzupassen und daher betriebssystemabhängig. Die Benutzung der AWT-Komponente dagegen ist auf allen Betriebssystemen gleich.

Eine Alternative bietet die betriebssystemabhängige GUI-Programmierung an, wobei je nach Betriebssystem/Rechner andere Bibliotheken zu benutzen und zu erlernen sind.

MFC	Microsoft Foundation Classes	Microsoft	Windows 95, NT
OWL	Object Windows Library	Borland	Windows (3.1,) 95, NT
Xlib	X-Windows-Library	Open Group	diverse Unix-Derivate
Motif	Motif-Library	Open Group	diverse Unix-Derivate
Mesa	Motif-Clone unter GNU-Lizenz	Brian Paul	diverse Unix-Derivate u.a.
Lesstif	Motif-Clone unter GNU-Lizenz	Hungry Programmers	diverse Unix-Derivate
Open GL	Open GL-Library (2D-, 3D-Library)	Architecture Review Board (Digital, Evans & Sutherland, HP, IBM, Intel, Microsoft, SGI) diverse Unix-Derivate, Windows NT	

Das AWT ist wesentlich einfacher gehalten in der Auswahl der zur Verfügung stehenden Komponenten und dem Event-Handling im Vergleich zur direkten Programmierung mit Benutzung der oben genannten Libraries. Dafür bietet das AWT konsequente Objektorientierung, Standardisierung, Ports auf viele Plattformen und einen einfacheren Aufbau der Klassenhierarchien und leichtere Anwendung der AWT-Komponenten.

Den Nachteil des AWT, kleinster gemeinsamer Nenner von GUI-Elementen und Event-Handling der verschiedenen Windows-Systeme zu sein, wird im JDK 1.2 beta (bzw. für JDK 1.1 als extra API) durch die Einführung des **Swing**-Toolkits, auch **JFC** = Java Foundation Classes genannt, die JavaSoft (Tochter von Sun) zusammen mit Netscape entwickelt

hat. Swing ist eine API (Application Programming Interface), d.h. es besteht aus mehreren Packages, die zeitgemäßere Komponenten und leichteres Handling (Layout-Manager, Events, ...) anbietet.

GUIs bestehen aus

Containern und Komponenten.

Eine **Komponente** ist ein Grundelement der GUI; in Java sind das unter dem JDK 1.1 die Elemente

```
Button, Canvas, CheckBox, CheckBoxMenuItem, Choice,  
Label, List, MenuItem  
Scrollbar, TextArea, TextField
```

Komponenten, die bereits für spezielle Anwendungszwecke Containerfähigkeiten besitzen, sind z.B.

```
CheckBoxGroup, MenuBar, Menu, PopupMenu.
```

Ein **Container** faßt Komponenten zusammen, nimmt diese auf und gruppiert sie, so daß z.B. Eigenschaften des Containers auf die darin enthaltenen Komponenten übertragen werden. Container, die Java vordefiniert und die flexibler einsetzbar sind als die oben genannten, sind

```
Applet,  
Dialog, FileDialog,  
Frame, Panel, ScrollPane, Window.
```

Ein Anzeigen einer Komponente in einem Applet erfolgt auf diese Weise:

- 1) Erzeugen eines Containers für die Komponente ist überflüssig, da das Applet selbst ein Container ist.
- 2) Anlegen eines Objektes der AWT-Komponente, z.B.

```
Button button_OK = new Button("OK"); // Beschriftung des Buttons: OK  
TextField tf_passwd = new TextField(15); // TextField zeigt max. 15 Zeichen
```

und Festlegen der Eigenschaften

```
tf_passwd.setEchoChar('*'); // statt eingegebenen Zeichen  
// erscheint '*'
```

3) Ggf. Wahl des Layout-Managers

Jeder Container hat einen Layout-Manager voreingestellt, der für die Anordnung der in ihr enthaltenen Komponenten verantwortlich ist.

Ein Layout-Manager kann voreingestellte Eigenschaften (Größe, Position, Ausrichtung, ...) der Komponenten überschreiben!

```
Applet, Panel:      BorderLayout
Dialog, Frame, Window: BorderLayout
```

Das Setzen des Layout-Managers geschieht durch

```
fenster.setLayout(new GridLayout(3, 2));
```

Die Methode `setLayout()` erwartet als Argument eine Referenz auf das Interface `LayoutManager`. Normalerweise muß das Argument nicht vorher in einer Variable/Referenz abgespeichert werden, da normalerweise Methoden des Layout-Manager nicht direkt aufgerufen werden müssen.

4) Hinzufügen der Komponente mit Hilfe des Layout-Managers

Im Applet erfolgt dies durch

```
add(<Komponentenobjekt>) bzw.
add(<Komponentenobjekt>, <Positionsobjekt>),
```

z.B. durch

```
add(button_OK);           // Anordnung des Buttons durch LayoutManagern
                           // (nicht: BorderLayout)
add(tf_passwd, "North");  // noerdliche Ausrichtung
                           // des Textfields bei BorderLayout
```

In Java vordefinierte Layout-Manager sind:

<code>BorderLayout</code>	Anordnung in Anteile nördlich, südlich, westlich, östlich, zentriert
<code>CardLayout</code>	Layout für mehrere Anzeigeseiten in einem Container (verwaltet mehrere Containeranordnungen)
<code>FlowLayout</code>	Anordnung nacheinander in eine Zeile, bei Bedarf Umbruch der Komponenten in eine neue Zeile
<code>GridBagLayout</code>	Anordnung gemäß von Nebenbedingungen, die durch <code>GridBagConstraints</code> formuliert werden
<code>GridLayout</code>	Anordnung in einem $m \times n$ -Gitter, d.h. m Zeilen, n Spalten
<i>eigener Layout-Manager</i>	gemäß eigenem Layout-Manager
<code>null</code>	expliziter Verzicht auf einen Layout-Manager für absolute Positionierung mit Pixelangaben

5) Ggf. Event-Handling für die Komponente definieren

Das Delegationsmodell im JDK 1.1 funktioniert folgendermaßen: Ein Ereignis/event (z.B. eine Mausbewegung, ein Tastendruck, ...) wird von einer Quelle/source (die Komponente, die den Event ausgelöst hat) an registrierte Abnehmer/Listener weitergegeben. Events und Listener sind selbst klassenhierarchisch (fast analog) aufgebaut. Listener sind Interfaces, die die Implementierung von Methoden zur Verarbeitung der Events erfordern. Die Registrierung der Listener erfolgt durch

```
<Komponentenobjekt>.add<Eventtyp>Listener(<Listener-Objekt>),  
z.B.  
button_OK.addActionListener(this);
```

Der Button `button_OK` registriert damit einen `ActionListener`, das Listener-Objekt ist dabei `this`, d.h. das aktuelle Objekt, das eine Implementierung der Listener-Methode(n) bietet. Voraussetzung dafür ist, daß die aktuelle Klasse das Interface `ActionListener` implementiert, d.h. die Methode

```
public void actionPerformed(ActionEvent evt);
```

Das Argument `evt` ist dabei das Eventobjekt, das alle wichtigen Informationen über den Event enthält (z.B. die Mouseposition beim Mouseclick, die Quelle des Ereignisses, z.B. den gedrückten Button, ...).

Eine Alternative ist die Verwendung von Adaptern, die Defaultimplementierungen (d.h. leere Methoden) der Interfacemethoden anbieten.

Ein Anzeigen einer Komponente in einer Applikation erfolgt auf diese Weise:

1) Erzeugen eines Containers für die Komponente und Einstellen der Eigenschaften des Containers

In Applikationen wird z.B. ein `Frame`-Objekt angelegt:

```
Frame fenster = new Frame("Fenster der Applikation xxx");  
fenster.setSize(200, 300); // Breite x Hoehe in Pixel
```

2) Anlegen eines Objektes der AWT-Komponente, z.B.

```
Button button_OK = new Button("OK"); // Beschriftung des Buttons: OK  
TextField tf_passwd = new TextField(15); // TextField zeigt max. 15 Zeichen
```

und Festlegen der Eigenschaften

```
tf_passwd.setEchoChar('*'); // statt eingegebenen Zeichen  
// erscheint '*'
```

3) Ggf. Wahl des Layout-Managers

Jeder Container hat einen Layout-Manager voreingestellt, der für die Anordnung der in ihr enthaltenen Komponenten verantwortlich ist.

Ein Layout-Manager kann voreingestellte Eigenschaften (Größe, Position, Ausrichtung, ...) der Komponenten überschreiben!

```
Applet, Panel:          BorderLayout
Dialog, Frame, Window: BorderLayout
```

Das Setzen des Layout-Managers geschieht durch

```
fenster.setLayout(new GridLayout(3, 2));
```

Die Methode `setLayout()` erwartet als Argument eine Referenz auf das Interface `LayoutManager`. Normalerweise muß das Argument nicht vorher in einer Variable/Referenz abgespeichert werden, da normalerweise Methoden des Layout-Manager nicht direkt aufgerufen werden müssen.

4) Hinzufügen der Komponente mit Hilfe des Layout-Managers

In Applikationen erfolgt dies durch

```
<Containerobjekt>.add(<Komponentenobjekt>) bzw.
<Containerobjekt>.add(<Komponentenobjekt>, <Positionsobjekt>),
```

z.B. durch

```
fenster.add(button_OK);           // fuer LayoutManager != BorderLayout
fenster.add(tf_passwd, "North");  // fuer BorderLayout
```

In Java vordefinierte Layout-Manager sind:

<code>BorderLayout</code>	Anordnung in Anteile nördlich, südlich, westlich, östlich, zentriert
<code>CardLayout</code>	Layout für mehrere Anzeigeseiten in einem Container (verwaltet mehrere Containeranordnungen)
<code>FlowLayout</code>	Anordnung nacheinander in eine Zeile, bei Bedarf Umbruch der Komponenten in eine neue Zeile
<code>GridBagLayout</code>	Anordnung gemäß von Nebenbedingungen, die durch <code>GridBagConstraints</code> formuliert werden
<code>GridLayout</code>	Anordnung in einem $m \times n$ -Gitter, d.h. m Zeilen, n Spalten
<i>eigener Layout-Manager</i>	gemäß eigenem Layout-Manager
<code>null</code>	expliziter Verzicht auf einen Layout-Manager für absolute Positionierung mit Pixelangaben

5) Ggf. Event-Handling für die Komponente definieren

Das Delegationsmodell im JDK 1.1 funktioniert folgendermaßen: Ein Ereignis/event (z.B. eine Mausbewegung, ein Tastendruck, ...) wird von einer Quelle/source (die

Komponente, die den Event ausgelöst hat) an registrierte Abnehmer/Listener weitergegeben. Events und Listener sind selbst klassenhierarchisch (fast analog) aufgebaut. Listener sind Interfaces, die die Implementierung von Methoden zur Verarbeitung der Events erfordern. Die Registrierung der Listener erfolgt durch

```
<Komponentenobjekt>.add<Eventtyp>Listener(<Listener-Objekt>),  
z.B.  
button_OK.addActionListener(this);
```

Der Button `button_OK` registriert damit einen `ActionListener`, das Listener-Objekt ist dabei `this`, d.h. das aktuelle Objekt, das eine Implementierung der Listener-Methode(n) bietet. Voraussetzung dafür ist, daß die aktuelle Klasse das Interface `ActionListener` implementiert, d.h. die Methode

```
public void actionPerformed(ActionEvent evt);
```

Das Argument `evt` ist dabei das Eventobjekt, das alle wichtigen Informationen über den Event enthält (z.B. die Mouseposition beim Mouseclick, die Quelle des Ereignisses, z.B. den gedrückten Button, ...).

Eine Alternative ist die Verwendung von Adaptern, die Defaultimplementierungen (d.h. leere Methoden) der Interfacemethoden anbieten.

Beispiel 3.1 *Das Pizza-Beispiel aus Abschnitt 1.20.2 (Konvertierung von/in Strings):*

Man braucht Bezeichnungen (in Java: Objekte der Klasse `java.awt.Label`) für Pizzanamen, Getränkenamen, Gesamtsumme sowie den Hinweis auf die Euro-Preise, Textein- und ausgabefelder (in Java: `java.awt.TextField`) für den Preis der Pizza, des Getränks und die Gesamtsumme, als auch einen Button (in Java: `java.awt.Button`) zum Berechnen des Preises.

```
import java.awt.*;  
import java.applet.*;  
  
// Schritt 1: entfaellt, da Restaurant ein spez. Applet ist,  
//           daher ein Container ist  
  
public class Restaurant extends Applet  
{  
    // Datenelemente  
    String name_pizza, name_getraenk;  
    double preis_pizza, preis_getraenk, summe;  
    ...  
  
    // GUI-Datenelemente:  
    Button button_preis;  
    Label lab_name_pizza, lab_name_getraenk, lab_summe;  
    TextField tf_preis_pizza, tf_preis_getraenk, tf_summe;  
    ...  
}
```

```

public void init()
{
    // Schritt 2: Anlegen der AWT-Komponenten

    lab_name_pizza = new Label("Pizza " + name_pizza + ":");
    tf_preis_pizza = new TextField("" + preis_pizza, 30);

    lab_name_getraenk = new Label(name_getraenk + ":");
    tf_preis_getraenk = new TextField("" + preis_getraenk, 30);

    lab_summe = new Label("Gesamtsumme: ");
    tf_summe = new TextField("0.00", 10);
    tf_summe.setEditable(false);

    Label lab_euro = new Label("(alle Angaben in EUR)");

    Button button_preis = new Button("Preis berechnen");

    // Schritt 3: Wahl des Layout-Managers, hier: GridLayout

    setLayout(new GridLayout(4,2,5,5));

    // Schritt 4: Hinzufuegen der Komponenten mit Layout-Manager
    //          (Reihenfolge wichtig!)

    add(lab_name_pizza);
    add(tf_preis_pizza);

    add(lab_name_getraenk);
    add(tf_preis_getraenk);

    add(lab_summe);
    add(tf_summe);

    add(lab_euro);
    add(button_preis);

    // Schrtt 5: entfaellt, hier kein Event-Handling

}
}

```

Beispiel 3.2 *Wie obiges Beispiel, nur soll jetzt noch das Event-Handling eingebaut werden. Einziges Ereignis (in Java: Event), das behandelt werden muss, ist der Druck/Anwahl des Buttons "Preis berechnen".*

Aus der Dokumentation zur Klasse `java.awt.Button` kann man ersehen, dass es nur eine

Methode der Form `"add ... Listener(..)"` gibt:

```
public void addActionListener(ActionListener l)
```

Klickt man in der Dokumentation auf das Interface `java.awt.event.ActionListener`, erhält man Informationen zum Interface: Es besteht aus einer einzigen Methode

```
public void actionPerformed(ActionEvent e)
```

Damit das Delegationsmodell von Ereignissen ab dem JDK 1.1 funktioniert, muss man im Sourcecode im Vergleich zum vorigen Schritt folgendes ändern:

```
// Schritt 5a: in den import-Anweisungen waehlt man alle Interfaces
//             aus dem Paket "java.awt.event" aus

import java.awt.event.*;

// Schritt 5b: das eigene Applet muss versprechen, das Interface
//             ActionListener implementieren, d.h.
//             es beinhaltet ALLE Methoden des Interfaces als eigene Methoden

public class Restaurant extends Applet
implements ActionListener
{
    // Datenelemente
    ...

    public void init()
    {
        // Schritte 2-4:
        ...

        // nach Schritt 2-4:

        // Schritt 5c: aktuelles Applet-Objekt (Verweis: this zeigt darauf)
        //             registriert sich als Listener zu allen Action-Events,
        //             die beim Button-Objekt button_preis auftreten
        //             --> immer, wenn dieser gedrueckt wird, wird automatisch
        //             die Methode "actionPerformed(..)" des Applets
        //             aufgerufen

        button_preis.addActionListener(this);
    }

    // Schritt 5d: versprochene Methode von ActionListener wird implementiert
```

```

public void actionPerformed(ActionEvent e)
{
    // ActionEvent wurde von Button button_preis erzeugt
    // d.h. dieser wurde angewaehlt

    if (e.getSource() == button_preis)
    {
        // Preiseingaben aus den TextField-Objekten wird abgefragt
        // und Gesamtpreis wird berechnet

        ...

        summe = preis_pizza + preis_getraenk;

        // Gesamtsumme wird in TextField-Objekt "tf_summe" geschrieben
        tf_summe.setText("" + summe);

        // TextField-Objekt "tf_summe" wird als veraltet/ungueltig
        // (vom Aussehen her) markiert

        tf_summe.invalidate();

        // das aktuelle Applet-Objekt soll ein neues gueltiges Aussehen
        // erhalten

        validate();
    }
}
}

```

3.2 * Elementare GUI-Komponenten

vgl. zusätzlich die Vorlesung

3.2.1 * Label

vgl. zusätzlich die Vorlesung

Ein **Label** ist eine Komponente, die einen einzeiligen Text beinhaltet (seine Beschriftung). Der Text kann links-, rechtsbündig oder zentriert ausgegeben werden. **Bsp.:**

```

Label name;                // Datenelement definieren
Label ueberschrift_rechts; // Datenelement definieren

public void init()
{

```

```

name = new Label("Namen:");

ueberschrift_rechts = new Label("Titel");

Font fn = new Font("SansSerif", Font.BOLD, 16);
ueberschrift_rechts.setFont(fn);

...

add(name);

...

add(ueberschrift_rechts);
}

```

allgemeine Informationen:

Klassenname: Label

Oberklasse: Component

Konstruktoren: Defaultkonstruktor: Label hat (noch) keine Beschriftung
String-Argument: Label mit Beschriftung

wichtigste Methoden von Label:	<code>int getAlignment()</code>	gebe Ausrichtung des
	<code>void setAlignment(int align)</code>	setze Ausrichtung des
	<code>String getText()</code>	gibt Beschriftung des
	<code>void setText(String label)</code>	setzt Beschriftung des

Die Ausrichtung wird dabei über die klassenabh. Konstanten

<code>Label.LEFT</code>	(Default, linksbündige Ausgabe)
<code>Label.CENTER</code>	(zentrierte Ausgabe)
<code>Label.RIGHT</code>	(rechtsbündige Ausgabe)

spezielle Listener/Events: keine

wichtigste ererbte Methoden von Component:	<code>Font getFont()</code>	gebe Font des Lab
	<code>void setFont(Font f)</code>	setze Font des Lab

3.2.2 * Button

vgl. zusätzlich die Vorlesung Ein **Button** ist eine Komponente, die ein drückbares Eingabefeld mit Text darstellt. Der Buttontext wird zentriert dargestellt. **Bsp.:**

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;           // fuer Interface ActionListener

public class MyApplet extends Applet implements ActionListener

    Button b_ok, b_no, b_cancel;    // Datenelemente definieren

    public void init()
    {
        b_ok = new Button("OK");
        b_no = new Button("No");
        b_cancel = new Button("Cancel");

        b_cancel.setEnabled(false); // Button kann man nicht druecken

        ...

        add(b_ok);
        add(b_no);
        add(b_cancel);

        // um auf Mouseclicks zu reagieren:
        b_ok.addActionListener(this);
        b_no.addActionListener(this);
        b_cancel.addActionListener(this);
        ...
    }

    public void actionPerformed(ActionEvent e)
    {
        Object verursacher = e.getSource();

        if (verursacher == b_ok)
        {
            System.out.println("OK-Button gedr\u00FCckt!");
        }
        else if (verursacher == b_no)
        {
            System.out.println("No-Button gedr\u00FCckt!");
        }
        else if (verursacher == b_cancel)
```

```

    {
        System.out.println("Cancel-Button gedr\u00FCckt!");
    }
    else
    {
        System.err.println("Manu?? Wer war der Verursacher des Events?");
    }
}
...
}

```

allgemeine Informationen:

Klassenname: Button

Oberklasse: Component

Konstruktoren: Defaultkonstruktor: Button hat (noch) keine Beschriftung
String-Argument: Button mit Beschriftung

wichtigste Methoden von Button:

	String getLabel()	gibt Beschriftung des Buttons
	void setLabel(String label)	setzt Beschriftung des Buttons

spezielle Listener/Events: ActionListener bzw.(ActionEvent)

Ein ActionEvent (abgeleitet von AWTEvent) wird vom TextField ausgelöst, sobald der Button gedrückt wird.

Interface-Methode:	void actionPerformed(ActionEvent e)
Registrierung:	void addActionListener(ActionListener al)
Event-ID:	ActionEvent.ACTION_PERFORMED

wichtigste ererbte Methoden von Component:

	bool isEnabled()	gibt zurück, ob die Komponente aktiviert ist
	void setEnabled(boolean b)	aktiviert (b) oder deaktiviert die Komponente
	Font getFont()	gebe Font der Komponente zurück
	void setFont(Font f)	setze Font der Komponente

3.2.3 * TextField

vgl. zusätzlich die Vorlesung

Ein **TextField** ist eine Komponente, die die Eingabe (oder nur die Anzeige) eines einzelnen Textes erlaubt. Das Textfeld ist dabei umrahmt und der Eingabebereich erhält eine gesonderte Hintergrunddarstellung (normalerweise weiß).

Klassenname:

TextField

Oberklasse:

TextComponent

TextComponent ist von Component abgeleitet

Konstruktoren:

Defaultkonstruktor: TextField hat (noch) keine Beschriftung und hat 0 Spalten

Argument `String text`: TextField mit Textvorgabe; die Spaltenzahl (= Anzahl der maximal gleichzeitig lesbaren Zeichen) ergibt sich aus der Länge des Strings

Argument `int cols`: TextField mit Vorgabe der Spaltenzahl ohne Textvorgabe

Argument `String text, int cols`: TextField mit Text- und Spaltenvorgabe

wichtigste Methoden von TextField:

<code>int getColumns()</code>	gibt Anzahl der Spalten des TextFields zurück
<code>void setColumns(int cols)</code>	setze Spaltenzahl des TextFields
<code>boolean echoCharIsSet()</code>	testet, ob ein Echo-Zeichen (= Zeichen, das statt eines eingegebenen Zeichen gezeigt wird) bereits gesetzt worden ist
<code>char getEchoChar()</code>	gibt das Echo-Zeichen des TextFields zurück
<code>void setEchoChar(char c)</code>	setzt das Echo-Zeichen des TextFields

TextField definiert zusätzlich die Methoden

```
Dimension getMinimumSize(),
Dimension getMinimumSize(int cols),
Dimension getPreferredSize(),
Dimension getPreferredSize(int cols)
```

mit den Varianten für die Größenangabe von TextFields mit vorgegebener Spaltenzahl.

wichtigste ererbte Methoden von TextComponent:

<code>String getText()</code>	gibt eingegebenen Text des <code>TextField</code> s zurück
<code>void setText(String s)</code>	setze den Text im <code>TextField</code> (Vorgabe/Anzeige)
<code>boolean isEditable()</code>	testet, ob der Text im <code>TextField</code> sich ändern kann
<code>void setEditable(boolean changeable)</code>	ermöglicht (Default) oder verbietet die Eingabe In letzterem Fall wird das <code>TextField</code> grau hinterlegt und hebt sich dadurch von veränderbaren <code>TextFields</code> ab
<code>String getSelectedText()</code>	gibt mit Mouse/Tasten selektierten Textes des <code>TextFields</code> zurück
<code>int getSelectionStart()</code>	gibt Anfangsindex des mit Mouse/Tasten selektierten Textes des <code>TextFields</code> zurück (Index beginnt mit 0)
<code>int getSelectionEnd()</code>	gibt Endindex des mit Mouse/Tasten selektierten Textes des <code>TextFields</code> zurück
<code>void select(int selStart, int selEnd)</code>	markiert den durch Start- und Endindex spezifizierten Teil des selektierten Textes des <code>TextFields</code> (dieser erscheint dann normalerweise in reverse-Darstellung)
<code>void selectAll()</code>	markiert den gesamten Text des <code>TextFields</code>
<code>int getCaretPosition()</code>	gibt die aktuelle Cursorposition im <code>TextField</code> zurück (als Index bzgl. des dort gezeigten Textes, beginnend mit 0)
<code>void setCaretPosition(int pos)</code>	setzt den Cursor auf die Textposition im <code>TextField</code>

spezielle Listener/Events:

a) `ActionListener` bzw. `ActionEvent`

Ein `ActionEvent` (abgeleitet von `AWTEvent`) wird vom `TextField` ausgelöst, sobald RETURN im `TextField` eingegeben wird.

Interface-Methode: `void actionPerformed(ActionEvent e)`

Registrierung: `void addActionListener(ActionListener al)`

Event-ID: `ActionEvent.ACTION_PERFORMED`

b) `TextListener` bzw. `TextEvent`

Ein `TextEvent` (abgeleitet von `AWTEvent`) wird vom `TextField` ausgelöst, sobald sich

der Text im `TextField` ändert.

```
Interface-Methode: public void textValueChanged(TextEvent e)
Registrierung:    void addTextListener(TextListener tl)
Event-ID:        TextEvent.TEXT_VALUE_CHANGED
```

3.2.4 * `TextArea`

vgl. zusätzlich die Vorlesung

Eine `TextArea` ist wie `TextField` eine Komponente, die von `TextComponent` abgeleitet ist. Sie erlaubt jedoch die Eingabe (oder nur die Anzeige) von mehrzeiligen Text. Die `TextArea` ist dabei umrahmt und der Eingabebereich erhält eine gesonderte Hintergrunddarstellung (normalerweise weiß). Je nach Größe werden automatisch horizontale und vertikale Scrollbars hinzugefügt. Die Funktionalität der Scrollbars und die Kopplung Bewegung des Scrollbars und Textverschieben muß nicht selbst programmiert werden und steht automatisch zur Verfügung.

Klassenname:

`TextArea`

Oberklasse:

`TextComponent`

`TextComponent` ist von `Component` abgeleitet

Konstruktoren:

Defaultkonstruktor: `TextArea` hat (noch) keine Beschriftung und hat 0 Zeilen und Spalten

Argument `String text`: `TextArea` mit Textvorgabe; die Größe (= Anzahl der maximal gleichzeitig lesbaren Zeilen und Spalten) ergibt sich aus der Länge des Strings und des Auftretens von Newline-Zeichen

Argument `String text, int rows, int cols`: `TextArea` mit Vorgabe der Größe und Textvorgabe

Argument `String text, int rows, int cols, int scrollbars`: `TextArea` mit Text- und Größenvorgabe und Vorgabe des Scrollbar-Erscheinungsbildes

Dabei nimmt das letzte Argument eine der klassenabh. Konstanten an, die `TextArea` definiert:

<code>TextArea.SCROLLBARS_BOTH</code>	horizontaler und vertikaler Scrollbar erscheinen
<code>TextArea.SCROLLBARS_VERTICAL_ONLY</code>	nur der vertikale Scrollbar erscheint
<code>TextArea.SCROLLBARS_HORIZONTAL_ONLY</code>	nur der horizontale Scrollbar erscheint
<code>TextArea.SCROLLBARS_NONE</code>	keiner der beiden Scrollbars erscheint

wichtigste Methoden von `TextArea`:

<code>int getColumns()</code>	gibt Anzahl der Spalten des <code>TextAreas</code> zurück
<code>void setColumns(int cols)</code>	setzt Spaltenzahl des <code>TextAreas</code>
<code>int getRows()</code>	gibt Anzahl der Zeilen des <code>TextAreas</code> zurück
<code>void setRows(int rows)</code>	setzt Zeilenzahl des <code>TextAreas</code>
<code>int getScrollbarVisibility()</code>	gibt Scrollbar-Erscheinungsbild in Form der oben erwähnten Scrollbar-Klassenkonstanten zurück
<code>void append(String s)</code>	hängt den String an die <code>TextArea</code> (hinten) an Es erfolgt kein vor- oder nachherige Ausgabe von einem Zeilenvorschub!
<code>void insert(String s, int pos)</code>	fügt den String an die angegebene Position der <code>TextArea</code> an Alle Zeichen des in der <code>TextArea</code> angezeigten Textes werden fortlaufend numeriert, dabei zählt auch jeder Zeilenumbruch als Zeichen mit.
<code>void replaceRange(String s, int start, int end)</code>	ersetzt Text der <code>TextArea</code> im angegebenen Bereich Dabei gehört der Startindex zum überschriebenen Bereich und die Endposition nicht.

`TextArea` definiert zusätzlich die Methoden

```
Dimension getMinimumSize(),
Dimension getMinimumSize(int rows, cols),
Dimension getPreferredSize(),
Dimension getPreferredSize(int rows, cols)
```

mit den Varianten für die Größenangabe von `TextAreas` mit vorgegebener Zeilen- und Spaltenzahl.

wichtigste ererbte Methoden von `TextComponent`:

vgl. `TextField`

spezielle Listener/Events:

`TextListener` bzw. `TextEvent`

vgl. hierzu `TextField`

Beachte, daß kein `ActionEvent` im `TextArea` ausgelöst wird!

3.2.5 * List

vgl. zusätzlich die Vorlesung

Ein **List** ist eine Komponente, die eine Auswahl von Möglichkeiten (= Strings) anbietet und eine Selektion von einer oder mehrerer Alternativen ermöglicht. Diese Komponente heißt auch in anderen GUIs `Listbox`. Ist der die **List** umgebende Platz zu klein, wird automatisch von Java ein vertikaler Scrollbar mit Funktionalität hinzugefügt.

Klassenname:

`List`

Oberklasse:

`Component`, implementiert das Interface `ItemSelectable`

Konstruktoren:

Defaultkonstruktor: `List` hat (noch) keine Einträge und erlaubt keine Mehrfachselektion

Argument `int anz`: `List` mit der angegebenen Zahl von gleichzeitig sichtbaren Einträgen (das ist *nicht* die Gesamtzahl der Einträge!) und erlaubt keine Mehrfachselektion

Argument `int anz, boolean multiple`: wie oben, nur mit Mehrfachselektion

wichtigste Methoden von List:

<code>int getRows()</code>	gibt Anzahl der gleichzeitig sichtbaren Einträge der <code>List</code> zurück
<code>int getItemCount()</code>	gibt Anzahl aller Einträge der <code>List</code> zurück
<code>String getItem(int index)</code>	gibt den angegebenen Eintrag der <code>List</code> mit seiner Beschriftung zurück Die Indexnumerierung beginnt dabei mit 0.
<code>String [] getItems()</code>	gibt alle Einträge der <code>List</code> mit ihrer Beschriftung als Vektor von Strings zurück
<code>void add(String moegl)</code>	hängt die angegebene Möglichkeit hinten an <code>List</code> an
<code>void add(String moegl, int pos)</code>	hängt die angegebene Möglichkeit hinten an <code>List</code> an Die Indexnumerierung beginnt dabei mit 0, ist <code>index</code> gleich <code>-1</code> oder größer gleich der Anzahl aller Einträge, so wird der Eintrag ganz hinten angehängt an das <code>List</code> -Objekt. Ansonsten wird er an die angegebene Position eingefügt und alle folgenden Einträge nach hinten verschoben.
<code>void delItem(int index)</code>	löscht den angegebenen Eintrag im <code>List</code> -Objekt
<code>void remove(int index)</code>	löscht den angegebenen Eintrag im <code>List</code> -Objekt
<code>void remove(String moegl)</code>	löscht den angegebenen Eintrag im <code>List</code> -Objekt
<code>void removeAll()</code>	löscht alle Einträge im <code>List</code> -Objekt
<code>void replaceItem(String neu, int pos)</code>	überschreibt den Text mit dem angegebenen neuen Text im angegebenen Eintrag des <code>List</code> -Objekts
<code>int getSelectedIndex()</code>	gibt die Position des mit der Mause Taste selektierten Index zurück Das Resultat ist <code>-1</code> , falls kein Eintrag selektiert ist bzw. falls mehrere Einträge selektiert wurden (keine Eindeutigkeit gegeben). Standardmäßig (d.h. kein Aufruf einer <code>select()</code> -Variante) wird kein Eintrag vorselektiert.
<code>int [] getSelectedIndexes()</code>	gibt die Position des aller mit der Mause Taste selektierten Indizes zurück (insbes. nützlich bei Mehrfachselektierbarkeit) Das Resultat ist <code>null</code> , falls kein Eintrag selektiert ist.
<code>String getSelectedItem()</code>	gibt den mit der Mause Taste selektierten Eintrag als String zurück Das Resultat ist <code>null</code> , falls kein Eintrag selektiert ist bzw. falls mehrere Einträge selektiert wurden (keine

List definiert zusätzlich die Methoden

```
Dimension getMinimumSize(),  
Dimension getMinimumSize(int rows),  
Dimension getPreferredSize(),  
Dimension getPreferredSize(int rows)
```

mit den Varianten für die Größenangabe von Lists mit vorgegebener Zahl von sichtbaren Einträgen.

spezielle Listener/Events:

a) ActionListener bzw.(ActionEvent)

Ein ActionEvent (abgeleitet von AWTEvent) wird von List ausgelöst, sobald ein Eintrag mit Doppelklick oder mit Cursor- und RETURN-Taste ausgewählt wird. Ein Doppelklick hat aber keine Selektion/Deselektion zur Folge (löst aber bei der Mause einen ItemEvent aus mit DESELECTED- oder SELECTED-Ursache), der Zustand des Eintrags bleibt gleich bei diesem Event.

```
Interface-Methode: void actionPerformed(ActionEvent e)  
Registrierung:    void addActionListener(ActionListener al)  
Event-ID:        ActionEvent.ACTION_PERFORMED  
Methoden:        String getActionCommand() gibt den angeklickten Eintrag als String
```

b) ItemListener bzw. ItemEvent

Ein ItemEvent (abgeleitet von AWTEvent) wird von List ausgelöst, sobald ein Eintrag neu selektiert oder deselektiert wird, d.h. sich der Zustand des Eintrags ändert (dies kann mit der Mousetaste oder mit den Cursor- und der Leertaste erfolgen).

```
Interface-Methode: public void itemStateChanged(ItemEvent e)  
Registrierung:    void addItemListener(ItemListener il)  
Event-ID:        ItemEvent.ITEM_STATE_CHANGED  
Event-Methoden:  Object getItem()  
                  gibt den Eintrag als Integer-Objekt zurück (mit dem  
                  zugehörigen int-Wert der Position des Eintrags), des-  
                  sen Zustand sich geändert hat  
                  int getStateChange()  
                  gibt den Eventgrund zurück, wobei die beiden Klassen-  
                  konstanten ItemEvent.SELECTED (Eintrag wurde selek-  
                  tiert) bzw. ItemEvent.DESELECTED (Eintrag wurde de-  
                  selektiert) zurückgegeben werden.  
                  ItemSelectable getItemSelectable()  
                  gibt das Objekt zurück, in dem der Event ausgelöst  
                  wurde, d.h. das List-Objekt selbst
```

3.2.6 * Choice

vgl. zusätzlich die Vorlesung

Eine **Choice** ist eine Komponente, die eine Auswahl von Möglichkeiten (= Strings) anbietet und eine Selektion von einer Alternative ermöglicht. Nur die momentan aktuelle Auswahl

in einer **Choice** ist sichtbar. Bei der Auswahl in einer **Choice** poppt eine Auswahl der Einträge auf, wenn man mit der Maus auf den Pfeil rechts von der **Choice** klickt.

Klassenname:

Choice

Oberklasse:

Component, implementiert das Interface ItemSelectable

Konstruktoren:

Defaultkonstruktor: Choice hat (noch) keine Einträge

wichtigste Methoden von Choice:

<code>int getItemCount()</code>	gibt Anzahl aller Einträge der <code>Choice</code> zurück
<code>String getItem(int index)</code>	gibt den angegebenen Eintrag der <code>Choice</code> mit seiner Beschriftung zurück Die Indexnumerierung beginnt dabei mit 0.
<code>void add(String moegl)</code>	hängt die angegebene Möglichkeit hinten an <code>Choice</code> an
<code>void addItem(String moegl)</code>	hängt die angegebene Möglichkeit hinten an <code>Choice</code> an
<code>void insert(String moegl, int index)</code>	fügt den angegebenen Eintrag im <code>Choice</code> -Objekt an die gewünschte Position ein (<code>index</code> darf nur Werte zwischen 0 und der Länge des <code>Choice</code> -Objekts - 1 annehmen.
<code>void remove(int index)</code>	löscht den angegebenen Eintrag im <code>Choice</code> -Objekt
<code>void remove(String moegl)</code>	löscht den angegebenen Eintrag im <code>Choice</code> -Objekt (genauer dessen erstes Auftreten)
<code>void removeAll()</code>	löscht alle Einträge im <code>Choice</code> -Objekt
<code>int getSelectedIndex()</code>	gibt den Index des in dem <code>Choice</code> -Objekt selektierten Eintrags zurück Standardmäßig (d.h. kein Aufruf einer <code>select()</code> -Variante) wird der erste Eintrag vorselektiert.
<code>String getSelectedItem()</code>	gibt den in dem <code>Choice</code> -Objekt selektierten Eintrag als <code>String</code> zurück Das Resultat ist <code>null</code> , falls das <code>Choice</code> -Objekt keine Einträge besitzt.
<code>Object [] getSelectedObjects()</code>	gibt ein Array der Länge 1 vom Komponententyp <code>Object</code> zurück mit dem im <code>Choice</code> -Objekt selektierten Eintrag Die Referenz verweist dabei auf ein <code>String</code> -Objekt mit der Bezeichnung des Eintrags (vgl. Interface <code>ItemSelectable</code>) Das Resultat ist <code>null</code> , falls das <code>Choice</code> -Objekt keine Einträge besitzt.
<code>void select(int index)</code>	selektiert den angegebenen Eintrag (danach wird dieser Eintrag in dem <code>Choice</code> -Objekt sichtbar)
<code>void select(String moegl)</code>	selektiert den angegebenen Eintrag (danach wird dieser Eintrag in dem <code>Choice</code> -Objekt sichtbar)

spezielle Listener/Events:

ItemListener bzw. ItemEvent

Ein ItemEvent (abgeleitet von AWTEvent) wird vom Choice ausgelöst, sobald ein Eintrag neu selektiert wird, d.h. mit der Mouse ausgewählt wird.

vgl. List

Interface-Methode:	public void itemStateChanged(ItemEvent e)
Registrierung:	void addItemListener(ItemListener il)
Event-ID:	ItemEvent.ITEM_STATE_CHANGED
Event-Methoden:	Object getItem() gibt den Eintrag als String-Objekt zurück (mit dem zugehörigen String-Wert der Bezeichnung des Eintrags) zurück, der momentan ausgewählt ist int getStateChange() gibt den Eventgrund zurück, bei Choice-Objekten ist dies stets die Klassenkonstante ItemEvent.SELECTED (Eintrag wurde selektiert) ItemSelectable getItemSelectable() gibt das Objekt zurück, in dem der Event ausgelöst wurde, d.h. das Choice-Objekt

3.2.7 * Checkbox und CheckboxGroup

vgl. zusätzlich die Vorlesung

Eine **Checkbox** ist eine Komponente, die einen Zustand ein/aus symbolisiert. Sie trägt einen erklärenden Text und links davon ein Quadrat zum Anchecken (= Anklicken) mit der Mouse.

Klassenname:

Checkbox

Oberklasse:

Component, implementiert das Interface ItemSelectable

Konstruktoren:

(), Defaultkonstruktor: Checkbox hat (noch) keine Beschriftung und ist nicht ausgewählt

(String s): Checkbox hat angegebene Beschriftung und ist nicht ausgewählt

(String s, boolean b): Checkbox hat angegebene Beschriftung und ist je nach Wert von b ausgewählt oder nicht

(String s, boolean b, CheckboxGroup cg): wie oben, nur gehört die Checkbox jetzt zu einer Gruppe, d.h. aus dieser Gruppe von Checkboxes kann immer nur eine ausgewählt sein.

wichtigste Methoden von Checkbox:

<code>String getLabel()</code>	gibt die Beschriftung der <code>Checkbox</code> zurück
<code>void setLabel(String s)</code>	setzt die Beschriftung der <code>Checkbox</code> auf den angegebenen Text
<code>boolean getState()</code>	gibt den Zustand der <code>Checkbox</code> zurück (<code>true</code> für angewählt)
<code>void setState(boolean b)</code>	setzt die Beschriftung der <code>Checkbox</code>
<code>CheckboxGroup getCheckboxGroup()</code>	gibt die zugehörige <code>CheckboxGroup</code> zurück, zu der die <code>Checkbox</code> gehört ergibt <code>null</code> , falls die <code>Checkbox</code> nicht zu einer <code>CheckboxGroup</code> gehört
<code>void setCheckboxGroup(CheckboxGroup g)</code>	nimmt die <code>Checkbox</code> in die Gruppe von <code>Checkboxen</code> , die <code>g</code> beschreibt, auf
<code>Object [] getSelectedObjects()</code>	gibt ein Array der Länge 1 vom Komponententyp <code>Object</code> zurück mit dem im <code>Checkbox</code> -Objekt selektierten Eintrag Die Referenz verweist dabei auf ein <code>String</code> -Objekt mit der Bezeichnung der <code>Checkbox</code> (vgl. Interface <code>ItemSelectable</code>)

spezielle Listener/Events:

`ItemListener` bzw. `ItemEvent`

Ein `ItemEvent` (abgeleitet von `AWTEvent`) wird vom `Checkbox` ausgelöst, sobald ein Eintrag neu selektiert wird, d.h. mit der Mouse angewählt wird.

vgl. `List`

Interface-Methode:	<code>public void itemStateChanged(ItemEvent e)</code>
Registrierung:	<code>void addItemListener(ItemListener il)</code>
Event-ID:	<code>ItemEvent.ITEM_STATE_CHANGED</code>
Event-Methoden:	<code>Object getItem()</code> gibt den Eintrag als <code>String</code> -Objekt zurück (mit dem zugehörigen <code>String</code> -Wert der Bezeichnung der <code>Checkbox</code>) zurück, die gerade angewählt wurde <code>int getStateChange()</code> gibt den Eventgrund zurück, bei <code>Checkbox</code> -Objekten ist dies stets die Klassenkonstante <code>ItemEvent.SELECTED</code> (Eintrag wurde selektiert) <code>ItemSelectable getItemSelectable()</code> gibt das Objekt zurück, in dem der Event ausgelöst wurde, d.h. das <code>Checkbox</code> -Objekt

Eine `CheckboxGroup` ist keine GUI-Komponente, sondern steht für eine Gruppe von `Checkbox`-Objekten, die zusammengefaßt werden. Die wesentliche Eigenschaft einer `CheckboxGroup` ist die Tatsache, daß innerhalb von ihr immer nur genau eine `Checkbox` an-

gewählt werden kann. Die in ihr enthaltenen `Checkbox`en werden auch mit einer Raute anstatt eines Quadrats symbolisiert.

Klassenname:

`CheckboxGroup`

Oberklasse:

`Object`

Konstruktoren:

`()`, Defaultkonstruktor: `CheckboxGroup` beinhaltet (noch) keine `Checkbox`-Objekte

wichtigste Methoden von `CheckboxGroup`:

<code>Checkbox</code> <code>getSelectedCheckbox()</code>	gibt das aktuell ausgewählte <code>Checkbox</code> -Objekt zurück
<code>void</code> <code>setSelectedCheckbox(Checkbox c)</code>	wählt das angegebene <code>Checkbox</code> -Objekt aus in der <code>CheckboxGroup</code>

Eine `Checkbox` kann auch bereits mit dem geeigneten Konstruktoraufruf von `Checkbox` zu einer `CheckboxGroup` gehören. Dies kann auch mit der `Checkbox`-Methode `setCheckboxGroup()` nachträglich erfolgen.

3.2.8 Scrollbar

vgl. die Vorlesung

3.2.9 Canvas

vgl. die Vorlesung

3.3 * Elementare GUI-Container

vgl. zusätzlich die Vorlesung

3.3.1 Panel

vgl. die Vorlesung

3.3.2 * Window

vgl. zusätzlich die Vorlesung

Ein **Window** ist ein Fenster, das stets ein "parent"-Objekt vom Typ **Frame** benötigt, um angezeigt zu werden. Ein "parent"-Objekt ist ein **Frame**, der die Erzeugung eines **Window**-Objektes initiiert. Normalerweise werden in Programmen nur Ableitungen von der Klasse **Window** verwendet, z.B. **Dialog**, **Frame**, Ein **Window**-Objekt hat deutlich weniger Möglichkeiten als die oben erwähnten Ableitungen (z.B. keinen Titel, kein eigenes Icon, ...).

Klassenname:

Window

Oberklasse:

Container

Konstruktoren:

Window(Frame parent): Die Erzeugung von Window wird vom angegebenen Frame initiiert.

wichtigste Methoden von Window:

<code>void show()</code>	zeigt das Window-Objekt an (Fenster im Betriebssystem sichtbar bzw. wird vollständig in Vordergrund gebracht) vgl. auch <code>setVisible(true)</code>
<code>void hide()</code>	verbirgt das Window-Objekt (Fenster im Betriebssystem nicht sichtbar) vgl. auch <code>setVisible(false)</code>
<code>void dispose()</code>	die vom Window-Objekt reservierten Ressourcen werden freigegeben
<code>boolean isShowing()</code>	gibt zurück, ob das Window-Objekt angezeigt wird (muß nicht notwendigerweise im Vordergrund dargestellt sein)
<code>void toFront()</code>	legt das Window-Objekt in der Darstellungsreihenfolge der Windows nach vorne und bringt es vollständig in den Vordergrund
<code>void toBack()</code>	legt das Window-Objekt in der Darstellungsreihenfolge der Windows nach hinten und macht es nicht sichtbar
<code>void pack()</code>	packt "optimal" alle im Window-Objekt aufgenommenen Komponenten (Alternative zu <code>setSize()</code>)
<code>Toolkit getToolkit()</code>	zugehöriges Window-Objekt wird zurückgegeben

spezielle Listener/Events:

a) `WindowListener` bzw. `WindowEvent`

Ein `WindowEvent` (abgeleitet von `ComponentEvent`, dieser abgeleitet von `AWTEvent`) wird vom `Window` ausgelöst, sobald es geöffnet oder geschlossen, iconifiziert bzw. deiconifiziert, aktiviert oder deaktiviert ist.

Interface-Methoden:	<code>void windowOpened(WindowEvent e)</code> <code>void windowClosing(WindowEvent e)</code> <code>void windowClosed(WindowEvent e)</code> <code>void windowIconified(WindowEvent e)</code> <code>void windowDeiconified(WindowEvent e)</code> <code>void windowActivated(WindowEvent e)</code> <code>void windowDeactivated(WindowEvent e)</code>
Registrierung:	<code>void addWindowListener(WindowListener wl)</code>
Event-IDs:	<code>WindowEvent.WINDOW_OPENED</code> <code>WindowEvent.WINDOW_CLOSED</code> <code>WindowEvent.WINDOW_CLOSING</code> <code>WindowEvent.WINDOW_ICONIFIED</code> <code>WindowEvent.WINDOW_DEICONIFIED</code>
Methoden des Events:	<code>Window getWindow()</code> gibt Verursacher als Referenz auf <code>Window</code> zurück (vgl. <code>getSource()</code>)

b) `ComponentListener` bzw. `ComponentEvent`
(erbt von `Component`)

Ein `ComponentEvent` (abgeleitet von `AWTEvent`) wird vom `Window` ausgelöst, sobald es von der Größe verändert, bewegt, angezeigt oder nicht mehr angezeigt wird.

Interface-Methoden:	<code>void componentResized(ComponentEvent e)</code> <code>void componentMoved(ComponentEvent e)</code> <code>void componentShown(ComponentEvent e)</code> <code>void componentHidden(ComponentEvent e)</code>
Registrierung:	<code>void addComponentListener(ComponentListener wl)</code>
Event-IDs:	<code>ComponentEvent.COMPONENT_MOVED</code> <code>ComponentEvent.COMPONENT_RESIZED</code> <code>ComponentEvent.COMPONENT_SHOWN</code> <code>ComponentEvent.COMPONENT_HIDDEN</code>
Methoden des Events:	<code>Component.getComponent()</code> gibt Verursacher als Referenz auf <code>Component</code> zurück (vgl. <code>getSource()</code>)

wichtigste ererbte Methoden von `Component/Container`:

<code>Dimension getSize()</code>	gibt Größe des Windows zurück
<code>void setSize(Dimension dim)</code>	setzt die Größe des Windows
<code>LayoutManager getLayout()</code>	gibt den Layoutmanager des Windows zurück
<code>void setLayout(LayoutManager lm)</code>	setzt den Layoutmanager des Windows
<code>add(Component c)</code>	fügt gemäß dem verwendeten Layoutmanager die angegebene Komponente zum Window hinzu Ein Anzeigen der Komponente ist damit u.U. noch nicht erfolgt (evtl. aktualisieren)
<code>add(Component c, int pos)</code>	fügt gemäß dem verwendeten Layoutmanager die angegebene Komponente an die angegebene Position zum Window hinzu Dabei ist zu beachten, daß der Container eine interne Liste aller Komponenten verwaltet.
<code>add(Component c, Object constr)</code>	fügt gemäß dem verwendeten Layoutmanager die angegebene Komponente an mit den angegebenen Nebenbedingungen zum Window hinzu Z.B. wird das zweite Argument beim BorderLayout verwendet, um die Position festzulegen (<code>\North\</code> , <code>\South\</code> , ...)
<code>add(Component c, Object constr, int index)</code>	Kombination der beiden oberen Methoden
<code>void remove(Component c)</code>	löscht die angegebene Komponente aus dem Container Eine Aktualisierung des Containers ist damit u.U. noch nicht erfolgt (evtl. explizit aktualisieren)
<code>void remove(int pos)</code>	löscht die angegebene Komponente mit der Position der internen Komponentenliste aus dem Container
<code>void removeAll()</code>	löscht alle Komponenten des Containers

<code>Component getComponent(int pos)</code>	gibt die Komponenten an der angegebenen Position der internen Komponentenliste des Containers zurück
<code>Component getComponent(int x, int y)</code>	gibt die (oberste) Komponente an der angegebenen Pixelposition des Containers zurück Liegt der Punkt im Container, befindet sich aber keine Komponente dort, wird der Container selbst zurückgegeben. Liegt er außerhalb, wird <code>null</code> zurückgegeben.
<code>Component [] getComponents()</code>	gibt alle Komponenten des Containers als Array zurück (gemäß interner Komponentenliste)
<hr/> <code>void validate()</code>	validiert alle Komponenten des Containers, d.h. stellt sie ggf. neu dar im Container, falls sie "invalid" sind (d.h. aktualisiert werden müssen, weil ihre interne(n) Darstellung/Eigenschaften nicht mit der Darstellung am Bildschirm übereinstimmen)
<code>void invalidate()</code>	markiert den Containers als "invalid" (und auch alle seine Vorfahren) Es wird ein neues Layout des Containers und seiner Vorfahren nötig.

3.3.3 * Frame

vgl. zusätzlich die Vorlesung

Ein **Frame** ist ein Fenster, das kein "parent"-Objekt benötigt, um angezeigt zu werden. Es ist ein selbständiges Fenster, das wie **Window** ein Container ist. Ein **Frame**-Objekt kann einen Titel haben, ein eigenes Icon, eine Menüleiste und kein Verändern seiner Größe verhindern. Häufig wird ein **Frame**-Objekt in Applikationen gebraucht, die GUIs verwenden. Der **Frame** wird dann als Ersatz des Containers verwendet, der im Appletviewer oder im WWW-Browser automatisch vorhanden ist.

Klassenname:

Frame

Oberklasse:

Window

implementiert:

MenuContainer

Konstruktoren:

Frame(): Frame ohne Beschriftung des Rahmens

Frame(String s): Frame mit angegebener Rahmenbeschriftung

wichtigste Methoden von Frame:

<code>void dispose()</code>	die vom Frame -Objekt reservierten Ressourcen werden freigegeben
<code>String getTitle()</code>	gibt die Rahmenbeschriftung zurück
<code>void setTitle(String s)</code>	setzt die Rahmenbeschriftung
<code>MenuBar getMenuBar()</code>	gibt die Menüleiste zurück
<code>void setMenuBar(MenuBar mb)</code>	setzt die Menüleiste des Frames
<code>void remove(MenuComponent mc)</code>	löscht die Menüleiste des Frames
<code>Image getIconImage()</code>	gibt das Icon als <code>Image</code> zurück
<code>void setIconImage(Image im)</code>	setzt das Icon mit dem angegebenen Bildobjekt In einigen Java-Ports werden iconifizierte Frames mit diesem Icon symbolisiert
<code>void setTitle(String s)</code>	setzt die Rahmenbeschriftung
<code>void setResizable(boolean flag)</code>	setzt/verhindert die Möglichkeit der Größenveränderung des Frame -Objekts
<code>boolean isResizable()</code>	gibt zurück, ob das Frame -Objekt von der Größe verändert werden kann

spezielle Listener/Events:

keine eigenen

wichtigste ererbte Methoden:

vgl. die Methoden von `Window`, insbes. `pack()`, `getToolkit()`, `isShowing()` und die ererbten Methoden von `Component/Container` (siehe `Window`)

3.3.4 * Dialog

vgl. zusätzlich die Vorlesung

Ein **Dialog** wird verwendet, um aus einem Fenster eine Dialogbox zu öffnen. Dabei wird zwischen

modalem Dialog

(solange Dialog besteht, d.h. die Methoden `hide()/setVisible(false)` bzw. `dispose()` wurden für Dialog nicht aufgerufen, kann man den übergeordneten Frame nicht anklicken)

und

modalem Dialog

unterschieden. Ein **Dialog** öffnet ein neues Fenster am Bildschirm, kann einen eigenen Titel haben und kann das Ändern der Größe untersagen. Es ist jedoch nicht selbständig, da es aus einem "parent"-Objekt erzeugt werden muß.

- Verwendung:
- Eingabe von Daten für Folgeoperationen
 - Abfrage von User-ID, Password, ...
 - kurze Fehler-/Warnmeldungen
 - Erzwingen einer Bestätigung

Klassenname:

Dialog

Oberklasse:

Window

Konstruktoren:

Dialog(Frame f): Dialog im nichtmodalen Modus ohne Titel, erzeugt aus angegebenem Frame

Dialog(Frame f, boolean modal): Dialog mit modalem Modus (true) oder nichtmodalen, erzeugt aus angegebenem Frame

Dialog(Frame f, String title): Dialog im nichtmodalen Modus mit angegebenem Fenstertitel, erzeugt aus angegebenem Frame

Dialog(Frame f, String title, boolean modal): Kombination der oberen beiden Konstruktoren

wichtigste Methoden von Dialog:

<code>void show()</code>	setzt modalen Dialog oder verhindert ihn
<code>String getTitle()</code>	gibt die Rahmenbeschriftung zurück
<code>void setTitle(String s)</code>	setzt die Rahmenbeschriftung
<code>void setResizable(boolean flag)</code>	setzt/verhindert die Möglichkeit der Größenveränderung des Dialog-Objekts
<code>boolean isResizable()</code>	gibt zurück, ob das Dialog-Objekt von der Größe verändert werden kann
<code>void setModal(boolean flag)</code>	setzt modalen Dialog oder verhindert ihn
<code>boolean isModal()</code>	gibt zurück, ob das Dialog-Objekt im modalen Dialog läuft oder nicht

spezielle Listener/Events:

keine eigenen

wichtigste ererbte Methoden:

vgl. die Methoden von Window, insbes. `pack()`, `getToolkit()`, `isShowing()` und die ererbten Methoden von Component/Container (siehe Window)

Ein Dialog kann – zumindest nicht unmittelbar – aus einem Applet verwendet werden, da Applet nicht von Frame abgeleitet ist.

3.3.5 * **FileDialog**

vgl. zusätzlich die Vorlesung

Ein **FileDialog** ist eine stets modale Dialogbox mit der speziellen Aufgabe, eine Auswahl eines Dateinamens zum Laden oder Speichern einer Datei zu ermöglichen. Es dient *nicht* dazu, diese Datei wirklich zu laden oder zu speichern. Mit Methoden der Klasse kann man den im Dialog selektierten Verzeichnis- und Dateinamen erhalten. Eine Voreinstellung von Verzeichnis- und Dateinamen kann ggf. im **FileDialog** erfolgen. Der Auswahlmechanismus sowie das Blättern in Verzeichnissen ist dabei schon implementiert und muß nicht selbst programmiert werden.

Ein **FileDialog** öffnet wie **Dialog** ein neues Fenster am Bildschirm, kann einen eigenen Titel haben und kann das Ändern der Größe untersagen. Es ist jedoch nicht selbständig, da es aus einem "parent"-Objekt erzeugt werden muß.

Klassenname:

`FileDialog`

Oberklasse:

`Dialog`

Konstruktoren:

`FileDialog(Frame f): FileDialog`, erzeugt aus angegebenem `Frame` ohne Titel

`FileDialog(Frame f, String title): FileDialog` im nichtmodalen Modus mit angegebenem Fenstertitel, erzeugt aus angegebenem `Frame`

`FileDialog(Frame f, String title, int mode):` wie oben, nur wird entweder der `Ladedialog` begonnen, falls `mode` mit `FileDialog.LOAD` übereinstimmt, oder der `Speicherdialog`, falls `mode` mit `FileDialog.SAVE` übereinstimmt

Der Modus ändert dabei nur Aussehen/Beschriftung des Dialogs, aber nichts an der Funktionalität

wichtigste Methoden von FileDialog:

<code>String getDirectory()</code>	gibt das ausgewählte Verzeichnis als <code>String</code> zurück
<code>void setDirectory(String name)</code>	stellt das Verzeichnis als Voreinstellung bei der Auswahl ein
<code>String getFile()</code>	gibt den ausgewählten Dateinamen zurück
<code>void setFile(String name)</code>	stellt den Dateinamen als Voreinstellung bei der Auswahl ein
<code>int getMode()</code>	gibt den Dialogmodus als eine der beiden Klassenkonstanten <code>FileDialog.LOAD</code> oder <code>FileDialog.SAVE</code> zurück
<code>void setMode(int mode)</code>	setzt den Dialogmodus mit einer der beiden Klassenkonstanten
<code>FilenameFilter getFilenameFilter()</code>	gibt den verwendeten <code>FilenameFilter</code> zur Einschränkung der Auswahl zurück
<code>void setFilenameFilter(FilenameFilter fn_filter)</code>	setzt den angegebenen <code>FilenameFilter</code> zur Einschränkung der Auswahl Zur Implementierung von <code>FilenameFilter</code> muß eine Methode <code>boolean accept(File dir, String filename)</code> geschrieben werden, die prüft, ob die Datei im angegebenen Verzeichnis mit dem angegebenen Namen das Auswahlkriterium erfüllt (Rückgabe <code>true</code>) oder nicht (Rückgabe <code>false</code>).

spezielle Listener/Events:

keine eigenen

wichtigste ererbte Methoden:

vgl. die Methoden von `Dialog`, insbes. `getTitle()`, `setTitle()` und die ererbten Methoden von `Component/Container` (siehe `Dialog`)

Wird die Eingabe abgebrochen, erhalten Verzeichnis- und Filename den Wert `null` bzw. die Vorgaben, die mit `setDirectory()/setFile()` eingestellt wurden. Wird ein `FilenameFilter` gesetzt, wird die Auswahl auf die durch den Filter bestimmten Dateien eingeschränkt.

```
// akzeptiert nur Java-Sourcefiles mit Endung ".java"
// oder Verzeichnisse

boolean accept(File dir, String filename)
{
```

```
    if (filename.endsWith(".java"))
        return true;
    else
        return (new File(dir, name)).isDirectory();
}
```

Ein `FileDialog` kann – zumindest nicht unmittelbar – aus einem Applet verwendet werden, da Applet nicht von `Frame` abgeleitet ist.

3.3.6 * Scrollpane

vgl. zusätzlich die Vorlesung

Ein **ScrollPane** ist ein Container, der automatisch mit horizontalem und vertikalem Scrollbalken ausgestattet sein kann. Damit kann auf (aufwendiges) Selbsthinzufügen von **Scrollbars** verzichtet werden. Die Standardgröße ist (ohne Beeinflußung eines Layoutmanagers) ist 100×100 und kann mit `setSize()` geändert werden.

Klassenname:

`ScrollPane`

Oberklasse:

`Container`

Konstruktoren:

`ScrollPane()`: leeres `ScrollPane`, fügt ggf. Scrollbalken hinzu, falls dies gebraucht werden

`ScrollPane(int policy)`: leeres `ScrollPane` mit spezifiziertem Scrollbarmodus. Eingestellt wird mit einer der Klassenkonstanten

`ScrollPane.SCROLLBARS_ALWAYS`

`ScrollPane.SCROLLBARS_AS_NEEDED`

`ScrollPane.SCROLLBARS_NEVER`

wichtigste Methoden von ScrollPane:

<code>int getScrollbarDisplayPolicy()</code>	gibt den Scrollbar-Anzeigemodus als eine der oben genannten drei Klassenkonstanten zurück
<code>Adjustable getHAdjustable()</code>	gibt den horizontalen Scrollbar als <code>Adjustable</code> -Referenz zurück
<code>Adjustable getVAdjustable()</code>	gibt den vertikalen Scrollbar als <code>Adjustable</code> -Referenz zurück
<code>Point getScrollPosition()</code>	gibt die Position des "Kindes" von <code>ScrollPane</code> als Punkt zurück
<code>void setScrollPosition(Point p)</code>	setzt die Position des "Kindes" von <code>ScrollPane</code> als <code>Point</code> -Objekt
<code>void setScrollPosition(int x, int y)</code>	setzt die Position des "Kindes" von <code>ScrollPane</code> in x - y -Koordinaten
<code>Dimension getViewPortSize()</code>	gibt die Breite und Höhe des sichtbaren Ausschnitts der <code>ScrollPane</code> als <code>Dimension</code> -Objekt zurück
<code>int getHScrollbarHeight()</code>	gibt die Höhe des horizontalen Scrollbars zurück (unabhängig davon, ob er gerade angezeigt wird oder nicht)
<code>int getVScrollbarWidth()</code>	gibt die Breite des vertikalen Scrollbars zurück (unabhängig davon, ob er gerade angezeigt wird oder nicht)

spezielle Listener/Events:

keine eigenen

wichtigste ererbte Methoden:

vgl. die ererbten Methoden von `Component/Container`

Minimum, Maximum und `VisibleAmount` der Scrollbars einer `ScrollPane` werden intern gesetzt und dürfen nicht über die Referenzen von `getHAdjustable` oder `getVAdjustable` geändert werden.

Unabhängig davon, ob Scrollbars angezeigt werden oder nicht, kann man mit `setScrollPosition()` die Position des "Kind"-Elements verändern.

4 Strings

4.1 * Anlegen von Strings

Strings oder auch Zeichenketten genannt sind in Java einfach eine Folge von Zeichen beliebiger Länge (also z.B. ein Text mit Leerzeichen, ein Wort, ein Buchstabe, ...).

In Java sind Strings

Objekte der Klasse String.

Strings können dabei *variable* oder *konstante* Objekte sein, letztere erkennt man daran, daß sie in Doppelanführungszeichen

"

eingeschlossen sind.

Beispiele für konstante Strings:

""	leerer String, enthält keine Zeichen, Länge: 0
"A"	String aus einem Buchstaben, enthält das Zeichen A, Länge: 1
" "	String aus einem Buchstaben, enthält das Leerzeichen, Länge: 1
"\""	String aus einem Buchstaben, enthält das Doppelanführungszeichen, Länge: 1
"\n"	String aus einem Buchstaben, enthält das Newline-Zeichen, Länge: 1
"Java"	String aus 4 Buchstaben, Länge: 4
"Java 1.1"	String aus 8 Buchstaben mit eingeschlossenem Leerzeichen, Länge: 8
"Er sagte: \"Hallo!\""	String aus 18 Buchstaben mit zwei enthaltenen Doppelanführungszeichen, Länge: 18

Beachte: Bestimmte Sonderzeichen wie Doppelanführungs- oder Newline-Zeichen müssen im String durch das beginnende Erkennungszeichen '\ ' markiert werden (auch Maskierung genannt).

Beispiele für variable Strings:

String s = new String();	leerer String s
String s = new String("");	leerer String s
String s;	String implizit mit Referenz null vorbesetzt
String s = null;	String explizit mit Referenz null vorbesetzt
String s = new String("A");	String mit einem Zeichen
String s = new String("Java 1.1");	String mit zwei Wörtern

abkürzende Schreibweise (nur für Strings erlaubt):

String s = "";	leerer String s
String s = "A";	String mit einem Zeichen
String s = "Java 1.1";	String mit zwei Wörtern

Anlegen von Stringobjekten:

Stringobjekte werden in Funktionen oder Klassen definiert.

a) Definition innerhalb von Funktionen:

Dazu werden die Definitionen der Stringobjekte nach dem Funktionsnamen und zwischen '{' und '}' eingefügt.

```
public class Test
{
    public static void main(String [] argv)
    {
        String s1 = "Java";           // s1 erhaelt den Text 'Java'
        String s2 = "";              // s2 ist der leere String

        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
    }
}
```

Abkürzend kann man schreiben:

```
public class Test
{
    public static void main(String [] argv)
    {
        // s1 UND s2 sind Strings
        String s1 = "Java", s2 = ""; // s1 erhaelt den Text "Java"
                                    // s2 ist der leere String

        System.out.println("s1 = " + s1);
        System.out.println("s2 = " + s2);
    }
}
```

In Applets fügt man die Stringdefinition z.B. in die Methode "paint()" ein.

```
import java.applet.Applet;
import java.awt.Graphics;

public class StringApplet_1 extends Applet
{
    public void paint(Graphics g)
    {
        String titel = "Kurzer Titel";
        g.drawString(titel, 20, 20);
    }
}
```

Da die Methode "drawString()" als 1. Argument ein Stringobjekt erwartet, kann man statt dem konstanten String "Kurzer Titel" auch den variablen String "titel" an die Methode übergeben.

b) Definition außerhalb von Funktionen:

Oftmals bietet es sich an, Stringobjekte außerhalb den Methoden der Klasse, aber innerhalb der Klasse zu definieren. Damit werden diese Stringobjekte zu Datenelementen der Klasse, die in allen (objektabhängigen) Methoden der Klasse verfügbar sind. Das hat den Vorteil, daß man jetzt in allen Methoden auf die Stringobjekte zugreifen kann.

```
import java.applet.Applet;
import java.awt.*;          // fuer Color und Graphics

public class StringApplet_2 extends Applet
{
    // text ist Datenelement der Klasse StringApplet_2
    String text = "wenig Neues";

    public void init()
    {
        // richtig: text ist in allen Methoden vorhanden
        System.out.println("Es gibt " + text + "!");

        // Fehler: titel nur in paint() definiert
        // System.out.println("Titel: " + titel);
    }

    public void paint(Graphics g)
    {
        // titel ist kein Datenelement
        String titel = "Kurzer Titel";
        g.drawString(titel, 20, 20);

        // setzt die Farbe auf Blau dauerhaft um
        // String wird in blau ausgegeben
        g.setColor(Color.blue);

        // richtig: text ist in allen Methoden vorhanden
        g.drawString(text, 20, 100);
    }
}
```

einzelne Zeichen von Strings:

Einzelne Zeichen des Strings haben den Datentyp char (bedeutet: character). Strings erlauben zwar das Auslesen einzelner Zeichen mit der Methode `charAt(int index)`, nicht aber das Ändern einzelner Zeichen. Dazu muß man `StringBuffer`-Objekte verwenden, die eine Methode `setCharAt(int index, char neues_zeichen)` anbieten.

```

String gruesse = "Hallo";
System.out.println("1. Zeichen: " + gruesse.charAt(0));
System.out.println("2. Zeichen: " + gruesse.charAt(1));
int index = gruesse.length() - 1;
System.out.println("letztes Zeichen: " + gruesse.charAt(index));

// Durchlauf aller Zeichen
for (int i = 0; i < gruesse.length(); i++)
    System.out.println("Zeichen " + (i+1) + ": " + gruesse.charAt(i));

```

Obwohl sich *einzelne* Zeichen nicht ändern lassen, kann man den gesamten String ändern (genauer: man ändert die Referenz auf das Stringobjekt).

```

String prog = "applet";
System.out.println("prog vor der \u00C4nderung: " + prog);
prog = "Applet";
System.out.println("prog nach der \u00C4nderung: " + prog);

```

'+' bei Strings

Die Klasse `String` erlaubt mit einer leichten Notation, an einen vorhandenen String etwas anzuhängen. Man kann durchaus mehrfach anhängen.

```

String prog = "Java";

System.out.print("prog = ");
// Wert "Java" wird ohne Doppelanführungszeichen ausgegeben
System.out.println(prog);

// String "Java-Programm" steht in prog_2
String prog_2 = prog + "-Programm";
System.out.print("prog_2 = ");
System.out.println(prog_2);

// String "prog = Java" steht in s
String s = "prog = " + prog;
System.out.println(s);
// kuerzer:
System.out.println("prog = " + prog);

String s2 = "", typ = "Applet";

// String "Java-Applet" steht in s2
s2 = prog + "-" + typ;
System.out.println("s2 = " + s2);

```

Bei der Verwendung von "+" müssen nicht beide Operanden Strings sein. Alle Standarddatentypen wie "int", "double", ... und die meisten vordefinierten Klassen sind in Strings umwandelbar.

Beispiele:

```
public void paint(Graphics g)
{
    Dimension dim = getSize();
    int breite = dim.width;
    int hoehe = dim.height;

    // Kurzform fuer:
    // System.out.print("Breite des Appletfensters: ");
    // System.out.println(breite); // Ausgabe des int-Wertes (als String)
    System.out.println("Breite des Appletfensters: " + breite);

    System.out.println("Hoehe des Appletfensters: " + hoehe);
    System.out.println("Groesse des Appletfensters: " + dim);
}
```

Bei dem obigen Programm wird mit der Ausgabemethode "println()" des Objekts "System.out" folgendes durch Umwandlung in Strings ausgegeben:

"Breite des Appletfensters: " + breite: wandelt int-Wert `breite` in einen String um (Wert als Zeichenkette) und hängt diesen an den String "Breite des Appletfensters: "
"
Ist die aktuelle Breite 400 Pixel, wird also der String "Breite des Appletfensters: 400" ausgegeben

"Hoehe des Appletfensters: " + hoehe: wandelt int-Wert `hoehe` in einen String um (Wert als Zeichenkette) und hängt diesen an den String "Hoehe des Appletfensters: "
Ist die aktuelle Höhe 300 Pixel, wird also der String "Hoehe des Appletfensters: 300" ausgegeben

"Groesse des Appletfensters: " + dim: wandelt das Objekt `dim` der Klasse `Dimension` in einen String um (Bezeichnung + Datenelemente als Zeichenkette) und hängt diesen an den String "Groesse des Appletfensters: "
"
Ist die aktuelle Größe 400 x 300 Pixel, wird also ein String der Art "Groesse des Appletfensters: `java.awt.Dimension[width=400,height=300]`" ausgegeben

einige wichtige Methoden für Strings:

Eine Klasse wie z.B. `String` bietet Funktionen an, die über Objekte und dem `'.'` aufgerufen werden.

Methode	Beispielaufruf	Bedeutung
<code>length(..)</code>	<pre>String user = getParameter("name"); if (user.length() > 8) { System.out.println("Zu lang!"); }</pre>	Stringobjekt <code>user</code> wird aus dem <code>PARAM</code> -Tag des <code>HTML</code> -Files ausgelesen und dessen Länge ermittelt.
<code>equals(..)</code>	<pre>String user = getParameter("name"); if (user.equals("administrator")) { System.out.println("Superuser!"); }</pre>	Stringobjekt <code>user</code> wird aus dem <code>PARAM</code> -Tag des <code>HTML</code> -Files ausgelesen und dann mit dem String <code>"administrator"</code> auf Gleichheit geprüft.
<code>substring(..)</code>	<pre>String flugnr = "LH 1234"; String kuerzel = flugnr.substring(0, 2); String nr = flugnr.substring(3);</pre>	Stringobjekt <code>flugnr</code> wird besetzt mit <code>"flugnr"</code> , das Stringobjekt <code>kuerzel</code> enthält die ersten beiden Buchstaben <code>"LH"</code> (0 ist Startindex, <code>2-1 = 1</code> ist Endindex) <code>nr</code> enthält ab dem vierten Buchstaben alle Zeichen des Strings, also <code>"1234"</code> (3 ist Startindex)

Beispiel:

```
String s1 = "Applet";
String s2 = "Applikation";

System.out.print("String s1: ");
System.out.println(s1);

// kuerzer, aber aequivalent fuer s2:
System.out.println("String s2: " + s2);

if (s1.equals(s2))
    System.out.println("Nanu? Strings sind gleich!");
else
    System.out.println("Strings sind verschieden!");

if (s1.equals("Applet"))
{
```

```

        System.out.println("Habe ich mir gedacht!");
        System.out.println("In s1 steht Applet!");
    }
    else
    {
        System.out.println("Katastrophe!");
        // Abbruch mit Fehlercode 1
        System.exit(1);
    }

    System.out.println("Laenge von String s1: " + s1.length());
    System.out.println("Laenge von String s2: " + s2.length());

    // n erhaelt Laenge vom konstanten String "Hallo" (also 5)
    int n = "Hallo".length();
    System.out.print("Laenge von \"Hallo\": ");
    System.out.println(n);

    s1 = "Java ist meine Nummer 1!";
    System.out.println("s1 = " + s1);
    System.out.println("s1 (kleingeschrieben) = " + s1.toLowerCase());

```

Die Klasse `String` bietet ca. 50 verschiedene Methoden an, davon haben 26 Methoden verschiedenen Namen. Es gibt also Methoden mit gleichem Funktionsnamen, aber *unterschiedlichen* Übergabeparametern.

In der Dokumentation zu den APIs (Application Programming Interface = API) des JDK finden Sie eine detaillierte Beschreibung der Klasse `String` im Paket `java.lang`.

4.2 * Konvertierung von/in Strings

Beispiel 4.1 *Es soll ein Applet geschrieben werden, das im HTML-File angibt, welche Pizza mit welchem Preis und welches Getränk mit welchem Preis in einem Restaurant bestellt wurde.*

Den Preis der Pizza und des Getränks soll auch nach dem Start des Applets noch änderbar sein.

Das Applet soll den (nicht veränderbaren) Gesamtpreis berechnen und ausgeben (alle Preise sollen in Euro sein).

Vorgehen: *Das HTML-File muss einen Parameter mit einem Namen, z.B. "NamePizza", in einem PARAM-Tag innerhalb des APPLET-Tags spezifizieren und ihm einen Wert (immer ein String) zuweisen.*

HTML-File:

```

<APPLET CODE="Restaurant.class" WIDTH="850" HEIGHT="250">
  <PARAM NAME="NamePizza" VALUE="Calzone">
  ...

```

Entweder ist der WWW-Browser nicht javafähig oder die Ausführung von Java-Programmen ist bei den Optionen untersagt worden.
</APPLET>

*Im Applet wird ein Datenelement vom Datentyp "String" eingeführt, z.B. mit Namen "name_pizza". Mit der Applet-Methode "getParameter(..)" (einziges Argument ist der Name des Parameters, hier "NamePizza"), kann dieser Wert ermittelt werden. Da damit ein Datenelement initialisiert wird, sollte man dies in der Methode "init()" durchführen. Ist dieser Parameter im NAME-Attribut eines PARAM-Tag in dem HTML-File vorhanden, wird der Wert des zugehörigen VALUE-Attributs (immer) als String zurückgegeben. Fehlt diese Parameterangabe, wird stattdessen die (ungültige) Referenz "null" zurückgegeben. Dieser Fall sollte berücksichtigt werden und ggf. das Datenelement einen Standardwert erhalten (oder ein Abbruch/Ende des Applets eingeleitet werden).
Ausschnitt aus dem Java-Applet:*

```
public class Restaurant extends Applet ...
{
    // Datenelemente definieren
    // vom HTML-File uebergegebener Wert des Namens
    String name_pizza;
    // Defaultwert fuer Datenelement name_pizza
    String def_name_pizza;
    ...

    public void init()
    {
        // Defaultwert setzen
        def_name_pizza = "Maestro";

        name_pizza = getParameter("NamePizza");
        if (name_pizza == null)
        {
            // z.B. Fehler ausgeben
            System.out.println("Fehler! Parameterwert \"NamePizza\" fehlt");

            // Wert mit Defaultwert vorbesetzen
            name_pizza = def_name_pizza;
        }

        ...
    }
}
```

Mit der Preisangabe geht man mit einem double-Datenelement `preis_pizza` analog vor (auch die Zahl wird - leider - als String übermittelt). Man muss daher zunächst den String

ermitteln und dann diesen in einen *double*-Wert umwandeln. Dabei kann jedoch eine Umwandlung unmöglich sein, wenn der String gar kein gültiges Zahlenformat hat. Diese Ausnahmesituation (in Java ein automatisch erzeugtes Objekt der Klasse *NumberFormatException*) muss in einer *try-catch*-Anweisung abgefangen werden. Der *try*-Block wird sofort abgebrochen, wenn ein Fehlerfall eintritt, und es wird in den *catch*-Block verzweigt, wenn der Fehlerfall mit dem Datentyp der Ausnahmesituation (hier: *NumberFormatException*) übereinstimmt. In dem *catch*-Block sollte eine Fehlerwarnung ausgegeben werden und ggf. ein Standardwert für den Preis gesetzt werden (oder das Applet beendet werden).
Ausschnitt aus dem Java-Applet:

```
public class Restaurant extends Applet ...
{
    // Datenelemente definieren
    // Preis der Pizza als Datenelement
    double preis_pizza;
    // Default-Preis der Default-Pizza
    double def_preis_pizza;
    ...

    public void init()
    {
        // Defaultwert setzen
        def_preis_pizza = 7.10;

        String param_preis_pizza = getParameter("PreisPizza");
        if (param_preis_pizza == null)
        {
            // z.B. Fehler ausgeben
            System.out.println("Fehler! Parameterwert \"PreisPizza\" fehlt");

            // Wert mit Defaultwert vorbesetzen
            preis_pizza = def_preis_pizza;
        }
        else
        {
            try
            {
                // versuchte Umwandlung von String in double-Wert
                // ueber den Umweg Konstruktor von Wrapperklasse "Double"
                Double d_obj = new Double(param_preis_pizza);

                // im Fehlerfall wird autom. ein Objekt der Klasse
                // NumberFormatException erzeugt und diese Anweisung
                // nicht mehr ausgefuehrt -> Sprung in den catch-Block
                preis_pizza = d_obj.doubleValue();
            }
            // wird nur im passenden Fehlerfall ausgefuehrt
```

```

catch (NumberFormatException exc)
{
    System.out.println("Fehler! Umwandlung des Strings \""
        + param_preis_pizza + "\"");
    System.out.println("      f\u00fcr Preis gescheitert!");

    // Default-Werte setzen
    name_pizza = def_name_pizza;
    preis_pizza = def_preis_pizza;
}
}
...
}
}

```

*Die so erhaltenen Variablen werden benutzt, um die Vorgabewerte für die Texteingabefelder und die Labels zu setzen sowie den Gesamtpreis zu berechnen. Allerdings muss der Preis in der **double**-Variable zur Anzeige im Texteingabefeld wieder in einen String umgewandelt werden.*

Ausschnitt aus dem Java-Applet:

```

public class Restaurant extends Applet ...
{
    // Datenelemente definieren
    Label lab_name_pizza;
    TextField tf_preis_pizza;
    ...

    public void init()
    {
        ...

        // uebergebenen Name der Pizza als Label setzen
        //   (Verkettung von Strings)
        lab_name_pizza = new Label("Pizza " + name_pizza + ":");

        // uebergebenen Preis der Pizza (double-Variable) in
        //   String umwandeln und damit Texteingabefeld initialisieren
        //   (30 = Anzahl der zu zeigenden Zeichen des Texteingabefelds)
        tf_preis_pizza = new TextField("" + preis_pizza, 30);

        ...
    }
}

```

Es gibt zahlreiche Möglichkeiten, einen beliebigen Datentyp in einen String umzuwandeln (oder in der umgekehrten Richtung). Ein gängiger Weg ist jeweils als erster Punkt beschrieben.

- (i) beliebigen Datentyp in String umwandeln einfachste Möglichkeit (geht mit allen Datentypen): Anhängen des Wertes/der Variable von einem Datentyp an den leeren String:

```
int n = 123;
String n_als_string = "" + n;          // enthaelt String "123"

String zahl_als_string = "" + 234;    // enthaelt String "234"

double x = 6.78;
String x_als_string = "" + x;         // enthaelt String "6.78"

Dimension dim = new Dimension(100, 200);
String dim_als_string = "" + dim;
// enthaelt String "java.awt.Dimension[width=100,height=200]"

// oder:
// String dim_als_string = dim.toString();
```

Für Objekte kann man die objektabhängige Methode `toString()` verwenden, die es für alle Klassen gibt.

weitere Möglichkeiten:

- a) objektunabh. Methode `valueOf()` von `String`

```
static String valueOf(boolean b)
static String valueOf(char c)
...
static String valueOf(float f)
static String valueOf(double d)
static String valueOf(char [] values)
static String valueOf(Object obj)
```

Bsp.:

```
double x = 0.5;
String s = String.valueOf(x);
```

- b) Konstruktoren von `String`

```
public String (String s)
public String (StringBuffer sb)
public String (byte [] bytes)
public String (char [] values)
```

Bsp.:

```
char [] wort = { 'g', 'u', 't' };
String s = new String(wort);
```

oder:

```
s = String.valueOf(wort);
```

c) objektunabh. Methode `toString()` der Wrapperklassen

```
static String toString(<zugeh. Standarddatentyp> value)
```

für die Wrapperklassen

```
Byte, Double, Float, Integer, Long, Short,
```

nicht für Boolean und Character.

Bsp.:

```
double x = 0.5;
String s = Double.toString(x);
```

d) objektabh. Methode `toString()` der Wrapperklassen

```
String toString()
```

(vgl. c), aber auch für Boolean und Character).

Bsp.:

```
Double x_obj = new Double(0.5);
String s = x_obj.toString();
```

e) weitere klassenspezifische Umwandlungen

```
static String toBinaryString(int i)
static String toOctalString(int i)
static String toHexString(int i)
static String toBinaryString(long l)
static String toOctalString(long l)
static String toHexString(long l)
static String copyValueOf(char [] values)
```

Bsp.:

```
int i = 31;
String s_bin = String.toBinaryString(i);
String s_oct = String.toOctalString(i);
String s_hex = String.toHexString(i);
```

f) Stringkonkatenation

```
"" + <Ausdruck eines Standarddatentyps>
<Ausdruck eines Standarddatentyps> + ""
<Ausdruck vom Typ String> + <Ausdruck eines Standarddatentyps>
<Ausdruck eines Standarddatentyps> + <Ausdruck vom Typ String>
```

Bsp.:

```
double x = 0.5;
int i = 31;
String s = "" + i;
s = "i = " + i;
s = x + "";
s = x + "(Wert von x)";
```

- (ii) String in beliebigen Datentyp umwandeln eine Möglichkeit (geht mit allen Standard-datentypen außer mit `char`):

Finde die passende Wrapperklasse zum Standarddatentyp (im Beispiel unten die Klasse `Integer` zum Standarddatentyp `int`), erzeuge ein Objekt durch den Konstruktor mit einem Stringargument (hier: `Integer n_obj = new Integer(s)`), rufe deren objektabhängige Methode `xyzValue()` mit diesem Objekt auf (hier: `n_obj.intValue()`), die Rückgabe ist dann ein Wert des Standarddatentyps (hier: `int`).

Allerdings muss man den Aufruf in eine `try-catch`-Umgebung einbauen, damit man Fehler beim Umwandeln (in Java sogenannte `Exceptions` = Ausnahmesituationen) abfangen kann. Im Fehlerfall sollte man entweder abbrechen oder einen sinnvollen Defaultwert setzen.

```
String int_als_string = "123";
int n;

try
{
    Integer n_obj = new Integer(int_als_string);
    n = n_obj.intValue();
}
catch (NumberFormatException exc)
{
    System.out.println("Fehler bei der Umwandlung int -> String!");
    System.out.println("Systemmeldung:");
    System.out.println(exc);
    n = 0;
}

String double_als_string = "123";
double x;

try
{
    Double x_obj = new Double(double_als_string);
    x = x_obj.doubleValue();
}
catch (NumberFormatException exc)
{
    System.out.println("Fehler bei der Umwandlung double -> String!");
    System.out.println("Systemmeldung:");
    System.out.println(exc);
    x = 0.0;
}
```

Dies geht für alle Standarddatentypen bis auf `char`, `boolean` spielt eine Sonderrolle, da beim Anlegen des Objekts keine Exception erzeugt wird.

Standarddatentyp	Wrapperklasse	Umwandlungsmethode
bool	Boolean	boolean booleanValue()
byte	Byte	byte byteValue()
double	Double	double doubleValue()
float	Float	float floatValue()
int	Integer	int intValue()
long	Long	long longValue()
short	Short	short shortValue()

Eine andere Möglichkeit für alle Standarddatentypen außer boolean und char ist folgende:

Finde die passende Wrapperklasse zum Standarddatentyp (im Beispiel unten die Klasse `Integer` zum Standarddatentyp `int`), rufe deren objektunabhängige Methode `parseXYZ(...)` mit einem String auf (hier: `Integer.parseInt(...)`), die Rückgabe ist dann ein Wert des Standarddatentyps (hier: `int`).

Allerdings muss man den Aufruf in eine `try-catch`-Umgebung einbauen, damit man Fehler beim Umwandeln (in Java sogenannte `Exceptions` = Ausnahmesituationen) abfangen kann. Im Fehlerfall sollte man entweder abbrechen oder einen sinnvollen Defaultwert setzen.

```
String int_als_string = "123";
int n;

try
{
    n = Integer.parseInt(int_als_string);
}
catch (NumberFormatException exc)
{
    System.out.println("Fehler bei der Umwandlung int -> String!");
    System.out.println("Systemmeldung:");
    System.out.println(exc);
    n = 0;
}

String double_als_string = "123";
double x;

try
{
    x = Double.parseDouble(double_als_string);
}
catch (NumberFormatException exc)
{
    System.out.println("Fehler bei der Umwandlung double -> String!");
    System.out.println("Systemmeldung:");
}
```

```

    System.out.println(exc);
    x = 0.0;
}

```

Wieder muss man wie oben eine Exception abfangen.
weitere Möglichkeiten (meistens ist auch hier eine Ausnahmebehandlung nötig):

- a) objektabh. Methoden von String

```

char charAt(int index)
byte [] getBytes() // höherwertige Bytes gehen verloren
void getChars(int posStart, int posEnd,
               char [] dest, int posDestStart)
char [] toCharArray()

```

Bsp.:

```

String s = "k";
char c = s.charAt(0);
String s = "gut";
char [] values = s.toCharArray();

```

- b) Konstruktoren der Wrapperklassen

```

public <Wrapperklasse>(String s)

```

für die Wrapperklassen

Boolean, Byte, Double, Float, Integer, Long, Short,

nicht für Character.

Bsp.:

```

String s = "0.5";
Double x_obj = new Double(s);
double x = x_obj.doubleValue();

```

kürzer:

```

String s = "0.5";
double x = new Double(s).doubleValue();

```

- c) objektunabh. Methode valueOf() der Wrapperklassen

```

static <Wrapperklasse> valueOf(String s)

```

für alle Wrapperklassen außer Character.

Bsp.:

```

String s = "31";
Integer i_obj = Integer.valueOf(s);
int i = i_obj.intValue();

```

kürzer:

```

String s = "31";
int i = Integer.valueOf(s).intValue();

```

d) weitere klassenspezifische Umwandlungen

Klasse Byte:

```
static Byte decode(String s)
static byte parseByte(String s)
static byte parseByte(String s, int radix)
```

Klasse Double:

```
static double parseDouble(String s)
```

Klasse Float:

```
static float parseFloat(String s)
```

Klasse Integer:

```
static Integer decode(String s)
static int parseInt(String s)
static int parseInt(String s, int radix)
```

Klasse Long:

```
static Long decode(String s)
static long parseLong(String s)
static long parseLong(String s, int radix)
```

Klasse Short:

```
static Short decode(String s)
static short parseShort(String s)
static short parseShort(String s, int radix)
```

Bsp.:

```
String s = "31";
Integer i_obj = Integer.decode(s);
int i = i_obj.intValue();
i = Integer.parseInt(s);
i = Integer.parseInt(s, 10);
i = Integer.parseInt(s, 16);
```

4.3 * Hauptanwendung von Strings

Strings werden in Java vielfältig eingesetzt:

- (i) Übergabe von Kommandozeilenargumenten an Applikationen
Dies erfolgt über den Funktionsparameter `argv` in der Funktion `main()`. `argv` ist dabei ein Array von Strings (d.h. `argv` besteht aus einzelnen Komponenten, die alle Strings sind; die Komponenten werden über die Indizes 0 bis Länge-1 angesprochen).

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

Dann ist `argv.length` die Anzahl der übergebenen Argumente und

```

argv[0]:          1. Argument als String
argv[1]:          2. Argument als String
                :
                :
argv[argv.length-1]: letztes Argument als String

```

Der Aufruf "java Zins Robert.Baier 5 3.5" übergibt also die drei Strings Robert.Baier" in argv[0], "5" in argv[1] und "3.5" in argv[2] (und nicht etwa den int-Wert 5 und den double-Wert 3.5). Hier ist argv.length gleich 3. Wird nichts übergeben, ist argv.length gleich 0.

(ii) Übergabe von Parametern an Applets

Dies erfolgt über die HTML-Seite innerhalb des APPLET-Befehls im PARAM-Befehl. Dabei gibt NAME den Namen des Parameters für das Java-Programm an und VALUE den Wert des Parameters als String.

```

<APPLET CODE="ZinsApplet.class" HEIGHT=300 WIDTH=400>
  <PARAM NAME="namen" VALUE="Robert.Baier">
  <PARAM NAME="jahre" VALUE="5">
  <PARAM NAME="zinssatz" VALUE="3.5">
</APPLET>

```

Im Java-Applet-Programm kann man dann durch Aufruf von "getParameter()" mit Angabe der Namensbezeichnung des Parameters den Wert als String erhalten:

```

String person = getParameter("namen");
String zinsjahre = getParameter("jahre");
String zinsprozente = getParameter("zinssatz");

```

Wird nichts übergeben (d.h. ist kein PARAM-Tag im HTML-File mit dem gewünschten Namen vorhanden), ist der String (z.B. person) gleich der Nullreferenz null.

```

String person = getParameter("namen");
if (person == null)
{
  System.out.println("Fehler!");
  System.out.println("Der Parameter \"namen\" wurde nicht im HTML-File vorbes
  // mit Defaultwert besetzen (oder abbrechen)
  person = "Unbekannt";
}

```

(iii) Umwandlung von Parameterstrings in Standarddatentypen

Die übergebenen Strings aus (i) und (ii) müssen dann in einen int- oder einen double-Wert umgewandelt werden (→ die Exception NumberFormatException muß mit try- und catch-Block abgefangen werden).

vgl. (i), Java-Applikation:

```

int anzahlJahre;
double prozente;
try
{
    anzahlJahre = Integer.parseInt(argv[1]);
    prozente = Double.valueOf(argv[2]).doubleValue();
}
catch (NumberFormatException exc)
{
    System.err.println("Fehler bei der Umwandlung aufgetreten!");
    anzahlJahre = 0;
    prozente = 0.0;
}

```

vgl. (ii), Java-Applet:

```

int anzahlJahre;
double prozente;
try
{
    anzahlJahre = Integer.parseInt(zinsjahre);
    preisEinheit = Double.valueOf(zinsprozente).doubleValue();
}
catch (NumberFormatException exc)
{
    System.err.println("Fehler bei der Umwandlung aufgetreten!");
    anzahlJahre = 0;
    prozente = 0.0;
}

```

(iv) Umwandlung von Standarddatentypen in Parameterstrings

Bei Aufruf vieler Methoden von GUI-Komponenten in Applets muß der Parameter ein String sein, d.h. es muß zunächst eine Umwandlung in einen String erfolgen.

```

int anz = 10;
g.drawString("Anzahl: " + n, 30, 50);

```

(v) Abfrage von Texteingaben in Applets

Die Abfrage von Texteingaben in `TextField`- bzw. `TextArea`-Objekten erfolgt durch Einlesen des Strings mit der Methode `getText()` der gemeinsamen Oberklasse `TextComponent`. Dieser muß ggf. wieder in einen Standarddatentyp umgewandelt werden.

```

// fieldAnzahl hat Textvorgabe "0" und 4 als max. Anzahl der Zeichen
TextField fieldAnzahl = new TextField("0", 4);

```

```

TextField fieldPreis = new TextField("1.99", 10);
...
String eingabeAnzahl = fieldAnzahl.getText();
String eingabePreis = fieldPreis.getText();

int anz;
double preisEinheit;
try
{
    anz = Integer.parseInt(eingabeAnzahl);
    preisEinheit = Double.valueOf(eingabePreis).doubleValue();
}
catch (NumberFormatException exc)
{
    System.err.println("Fehler bei der Umwandlung aufgetreten!");
    anz = 0;
    preisEinheit = 0.0;
}

```

(vi) Änderung von Text in GUI-Komponenten des Applets

Auch die Änderung von Text in `TextField`- bzw. `TextArea`-Objekten erfolgt durch Übergabe eines Strings mit der Methode `setText()` der gemeinsamen Oberklasse `TextComponent`.

```

TextField fieldSumme = new TextField("0.0", 4);

double summe = anz*preisEinheit;
fieldSumme.setText(Double.toString(summe));

```

(vii) Angabe von Labels in `Frame`, `Button`, ...

viele GUI-Komponenten besitzen Labels oder Namen, die meistens als Strings realisiert sind

```

Button button_ok = new Button("OK");
Frame newWindow = new Frame("Neues Fenster");
..

```

(viii) Eingabe von Zeilen/Werten in Applikationen

vgl. Vorlesungsfile "`SwitchCountdown.java`"

Die Eingabe von Zeilen/Werten in Applikationen erfolgt über Eingabeströme (input streams). Komfortablere Eingabeströme können nicht nur byteweise einlesen, sondern auch ganze Zeilen. Bei Objekten der Klasse "`BufferedReader`" gibt es eine Methode "`readLine()`", die eine eingelesene Zeile als String zurückgibt.

```

BufferedReader is = new BufferedReader(
    new InputStreamReader(System.in));

String antw = "";
int sekunden = 0;
try
{
    System.out.print("Anzahl der Sekunden vor dem Countdown: ");
    antw = is.readLine();
    is.close();
    sekunden = Integer.valueOf(antw).intValue();
}
catch (IOException exc)
{
    System.err.println("Exception abgefangen: " + exc);
}
}

```

4.4 * Beispielprogramme

Testprogramme:

```

StringAnhang.java
StringApplet_1.java
StringApplet_2.java
StringApplet_3.java
StringChange.java
StringDef.java
StringTest_1.java
StringTest_2.java
String.html

```