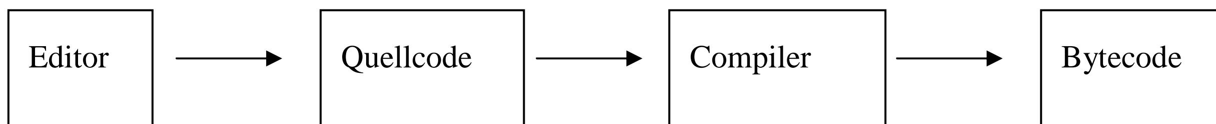
Grundlagen

1. Vom Quellcode zur Programmausführung

Das Besondere an Java ist, dass das geladene Programm nicht selber lauffähig ist, sondern von einem speziellen Programm interpretiert werden muss.

Es beginnt damit, dass man mit einem beliebigen ASCII-Editor den Quellcode eines Java-Programms schreibt.



Beispiel:

```
class HelloWorld
{
   public static void main(String args[])
   {
   System.out.println("Hello, world");
   }
}
```

Dieses Programm muss unter den Dateinamen abgespeichert werden, der den class-Namen entspricht, also im Beispiel ist der Dateiname HelloWorld. Die Extension der Quellcode-Datei lautet java.

Also, obiges Programm wird in der Datei HelloWorld.java abgespeichert.

Aus dem Java-Quellcode entsteht über einen Compiler, z.B. Javac, ein Bytecode, der in der Datei mit der Endung CLASS abgelegt wird. In unserem Beispiel entsteht die Datei Hello-World.class.

```
HelloWorld.class - Editor
Datei  Bearbeiten  Suchen  ?

Êþº¾ ▮ -  ▮ ▮▮ ▮▮ ▮▮ ▮▮ ▮▮ ▮        ▮ ▮▮ ▮ ▮▮ ▮ ▮▮ ▮
▮▮ ▮ ▮▮ ▮()V▮ ▮(Ljava/lang/String;)V▮
▮([Ljava/lang/String;)V▮ ▮<init>▮ ▮Code▮ ▮ConstantValue▮ ▮
Exceptions▮ ▮Hello, world▮ ▮HelloWorld▮ ▮HelloWorld.java▮
▮LineNumberTable▮ ▮Ljava/io/PrintStream;▮
▮LocalVariables▮ ▮SourceFile▮ ▮java/io/PrintStream▮
▮java/lang/Object▮ ▮java/lang/System▮ ▮main▮ ▮out▮
▮println   ▮ ▮     ▮     ▮ ▮ ▮ ▮   % ▮ ▮       ² ▮▮▮¶ ▮±
  ▮ ▮   ▮ ▮   ▮ ▮ ▮   ▮ ▮ ▮ ▮   ▮ ▮ ▮   ▮*·  ▮±    ▮ ▮   ▮
▮   ▮ ▮ ▮   ▮ ▮
```

Bytecode zum "HelloWorld" Beispiel

Besonders am Bytecode liegt es, dass er nicht auf einer tatsächlich existierenden Maschine laufen kann, sondern für eine sogenannte "virtuelle Maschine" erzeugt wurde. Das bedeutet, dass man diesen Bytecode nur durch ein spezielles Programm, die sogenannte Java Virtual Machine (JVM) ausführen lassen kann. In Programmen wie FireFox und Internet Explorer sind JVM integriert. Somit können diese Browser Java-Programme ausführen. Andere Programme wie java oder Jview können ebenfalls benutzt werden, um Bytecode zu interpretieren.

Warum compiliert man die Programme nicht wie bei C oder Pascal für die jeweiligen Plattformen (UNIX, Windows etc.)?

Man möchte erreichen, dass nur noch ein einziges Java-Programm für alle erdenklichen Rechner und Betriebssysteme benötigt wird. Das funktioniert, da der Bytecode auf allen Rechnern und Betriebssystemen immer identisch ist.

Dadurch entfällt das sonst beim Programmieren übliche Linken (Binden) und die Programme werden leichter portierbar.

Der Bytecode ist ausserdem sehr kompakt. Dadurch verringert sich die Ladezeit übers Netz.

Ein zweiter wichtiger Vorteil liegt darin, dass die JVM überprüfen kann, ob das Programm verbotene Zugriffe ausführen kann. Sollte dies der Fall sein, kann die JVM dies verhindern, da sie ja den Code interpretiert und ihn erst dann ausführt, wenn alle Prüfungen erfolgreich waren

Frage: Welche Entwicklungsziele gibt es nun bei der Java-Anwendungsentwicklung

Eine Eigenart von Java ist, dass ein Java-Programm entweder als Applet oder als Application (Standalone-Programm) erzeugt werden kann.

Eine **Application** ist ein Programm, das zwar auch von einem Interpreter ausgeführt wird, ansonsten aber eine eigenständige Anwendung ist. Das Programm java.EXE dient z.B. dazu, eine solche Application auszuführen.

Ein **Applet** wird meist in einen Browser geladen und dort ausgeführt. Die Idee, die hinter Applets steckt ist, dass man auf Web-Seiten Animationen erzeugen will oder dem Benutzer einen richtigen Dialog ermöglichen will.

Hier importiert das Java-Programm alle Klassen des Package *java.applet*. Im Listing geschieht dies folgendermaßen:

```
//Applet01.java

import java.applet.*;

public class Applet01 extends Applet
```

Bei Applets ist im HTML-Code der Web-Seite ein spezielles Applet-Tag im Einsatz, das das Java-Applet an diese Stelle im HTML-Code einbindet und sogar Parameter von HTML in das Java-Applet übergeben lässt.

```
<applet code="Applet01.class" width=200 heigth=50>

</applet>
```

Ein drittes wichtiges Ziel der Java-Entwicklung sind Beans. **JavaBeans** sind Java-Objekte, die durch das JavaBean-API definierte Methoden und Eigenschaften implementieren. Ziel des JavaBean-API ist die Definition eines Modells, das es ermöglicht, sogenannte Software-Komponenten zu entwickeln, die einerseits durch standardisierte Mechanismen und Schnittstellen beliebig miteinander kombiniert werden können, andererseits durch den Benutzer konfigurierbar sind. Z.B. könne ein Werkzeug zur Erstellung von Benutzeroberflächen JavaBeans-konforme GUI-Komponenten eines beliebigen Herstellers verwenden, einbinden und dem Programmierer zur Verfügung stellen. Eine vergleichende Architektur ist für Microsofts OLE (Object Linking and Embedding) definiert.

2. JDK Entwicklungsumgebung

Jede Programmiersprache ist nur so gut, wie ihre Werkzeuge sowie die vorhandenen Bibliotheken zum Erstellen, Debuggen und Warten von Programmen sind.

Das folgende Kapitel beschreibt die im Java Developers Kit (JDK) enthaltenen Werkzeuge.

2007 war JDK in der Version 1.6 verfügbar.

Das JDK enthält

- einen **Compiler** (`javac`), um Java Quelltext-Dateien zu übersetzen,
- einen **Interpreter** (`java`), um Java Applikationen auszuführen,
- einen **Debugger** (`jdb`) zur Fehlersuche in Java Applets und Applikationen,
- einen **Disassembler** (`javap`), um Informationen aus Java Byte-Code-Dateien zu erhalten,
- einen **Schnittstellengenerator für C-Funktionen** (`javah`),
- ein Werkzeug zur **Generierung von Programmdokumentation** im HTML-Format (`javadoc`) sowie
- ein Programm zur **Ausführung von Applets** (`appletviewer`).

Java Compiler

Der Java Compiler `javac` übersetzt Java Quelltext in Java Byte-Code. Java Bytecode repräsentiert Instruktionen für eine virtuelle Maschine, die vom Java Interpreter ausgeführt werden können. Die Quelltextdateien müssen die Endung `.java` haben, die Endung muß beim Aufruf des Compilers mit angegeben werden.

Aufruf des Compilers

```
javac [<Optionen>] <Dateiname>.java ...
```

Für jede in der Datei deklarierte Klasse erzeugt der Compiler eine Datei mit dem Namen

```
<Klassenname>.class.
```

Der Java-Compiler ist in Java implementiert.

Optionen des Java Compilers

Option	Beschreibung	Standardeinstellung
-classpath <Pfad> (Windows)	Spezifiziert den Pfad, in dem der Compiler nach existierenden Klassen suchen soll. Mehrere Verzeichnisse	Systemverzeichnis des JDK.

	werden durch ein Semikolon voneinander getrennt. Diese Option überschreibt die Umgebungsvariable `CLASSPATH`.	
-d <Pfad>	Die -d Option bezeichnet den Pfad, in dem die generierten `.class` Dateien abgelegt werden sollen.	Die generierten `.class` Dateien werden im Verzeichnis der Quelltextdateien abgelegt.
-g	Der Compiler erzeugt ausführliche Debug-Informationen, die in Tabellen abgelegt werden. Diese Option ist notwendig, wenn das Programm mit dem Werkzeug `jdb` untersucht werden soll.	Debug-Information enthalten nur Zeilennummern.
-O	Diese Option erzeugt einen in bezug auf das Zeitverhalten optimierten Byte Code. Der resultierende Code wird etwas größer.	Keine Optimierung.
-verbose	Bei dieser Option erzeugt der Compiler Ausgaben, welche die aktuell übersetzten Quelltextdateien und eingebundenen Klassen anzeigen.	Der Compiler erzeugt keine Ausgaben zu den eingebundenen Klassen und übersetzten Quelltextdateien.
-nowarn	Diese Option schaltet die Ausgabe von Warnungen aus.	Standardmäßig erzeugt der Compiler Warnungen.
-debug	Der Compiler druckt während der Übersetzung Debug-Informationen aus. Diese Option ist nicht zu verwechseln mit der Option -g, bei der für das übersetzte Programm Debuginformationen erzeugt werden.	Der Compiler gibt keine Debug-Informationen aus.
-nowrite	Die Quelltextdateien werden nur übersetzt, es wird kein Code generiert. Mit dieser Option kann eine Quelltextdatei auf syntaktische Korrektheit geprüft werden.	-

Beispiele

Der einfachste Weg, ein Java-Programm zu übersetzen, ist es, den Compiler ohne Optionen zu verwenden. Der Aufruf des Compilers mit dem Programm

```
javac  HelloWorld.java
```

Ist die Entwicklung des Programms abgeschlossen, kann die Optimierungs-Option (O) eingeschaltet werden:

```
javac -O HelloWorld.java
```

Java Interpreter

Der Java-Interpreter wird benötigt, um Java-Applikationen auf dem lokalen Rechner ausführen zu lassen. Der Java-Interpreter führt die durch den Compiler generierten Instruktionen für die virtuelle Maschine aus. Als Argument muß dem Interpreter der vollständig qualifizierte Klassenname übergeben werden. Die Endung .class darf nicht mit angegeben werden.

Aufruf des Java Interpreters

```
java [ <Optionen>] <Klassenname> <Argumentliste>
```

Die auszuführende Klasse (Klassenname) muss eine Funktion main() aufweisen, die wie folgt deklariert ist:

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

Der Interpreter ruft die Funktion main() auf. Dabei wird die Argumentliste an das Programm übergeben.

Optionen des Java Interpreters

Option	Beschreibung	Standardeinstellung
-classpath <Pfad> (Windows)	Spezifiziert den Pfad, in dem der Interpreter nach existierenden Klassen suchen soll. Mehrere Verzeichnisse werden durch ein Semikolon voneinander getrennt. Diese Option überschreibt die Umgebungsvariable CLASSPATH.	Systemverzeichnis des JDK.
-mx <bytes>	Definiert die maximale Größe des Heap. Der Wert muß größer als 1KB sein. Angaben können in Byte, Kilobyte (z.B. 100k) oder Megabyte (z. B. 10m) erfolgen.	16MB

-ms <bytes>	Definiert die initiale Grösse des Heap. Der Wert muß größer als 1KB sein. Angaben können in Byte, Kilobyte (z.B. 100k) oder Megabyte (z. B. 10m) erfolgen.	1MB
-ss<bytes>	Definiert die maximale Stackgröße für nativen Programmcode (C) innerhalb eines Thread. Der Wert muß größer als 1KB sein.	128KB
-oss<bytes>	Definiert die maximale Stackgröße für Java Programmcode innerhalb eines Thread. Der Wert muß größer als 1KB sein	400KB
-noasyncgc	Schaltet die asynchrone Garbage Collection aus. Der Garbage Collector muß dann explizit in Anwenderklassen aufgerufen werden.	Der Garbage Collector läuft asynchron in einem eigenen Thread.
-verbosegc	Der Garbage Collector erzeugt eine Ausgabe, wenn Objekte aus dem Heap entfernt werden.	Es werden keine Ausgaben erzeugt.
-cs -checksource	Vergleicht das Datum der Code-Datei mit dem Datum der Quelltext-Datei. Ist die Quelltext-Datei neuer als die Code-Datei, wird der Quelltext neu übersetzt und dann geladen.	Die Zeitstempel der Quelltext- und Code-Dateien werden nicht miteinander verglichen.
-verify	Jede Klasse wird verifiziert.	Nur der durch `classloader` eingebundene Code wird verifiziert.
-noverify	Schaltet die Verifikation für alle verwendeten Klassen aus.	Nur der durch `classloader` eingebundene Code wird verifziert
-verifyremote	Nur der durch `classloader` eingebundene Code wird verifiziert	
-debug	Diese Option ermöglicht ein Anbinden des Debuggers an den laufenden Interpreter. Der Interpreter zeigt nach dem Start ein Passwort an, das eingegeben werden muß, um den Debugger an den Interpreter zu binden.	Der Java Debugger kann sich nicht an einen laufenden Interpreter binden.

-t	Diese Option gilt nur für `java_g`. Sie erzeugt einen Trace der ausgeführten Instruktionen.	Es wird kein Trace erzeugt.
-verbose -v	Bei dieser Option erzeugt der Interpreter Ausgaben, wenn neue Klassen geladen werden.	Der Interpreter erzeugt keine Ausgaben während der Ausführung.
-help	Anzeigen der Optionen	
-version	Anzeigen der Versionsnummer	

Beispiele

Das Programm kann mit Hilfe des Interpreters ohne Angabe weiterer Optionen gestartet werden.

```
java ToolBeispiel
```

Der optionale Parameter für die Methode main()der Klasse ToolBeispiel wird dem Aufruf angehängt:

```
java ToolBeispiel 5
```

Java-Debugger

Der Java Debugger ist ein kommandozeilen-basiertes Werkzeug zur Fehlersuche in Java Programmen. Er ist wie der Compiler in Java implementiert. Klassen, die mit Hilfe von jdb untersucht werden sollen, müssen mit der Option -g des Compilers übersetzt werden. Diese Option erzeugt Debug-Informationen für jdb. Als kommandozeilen-basiertes Werkzeug besitzt der Debugger keine graphische Benutzeroberfläche. Stattdessen wird eine Reihe von Kommandos unterstützt, die der Benutzer über die Tastatur am Prompt des Debuggers eingeben muss.

Es existieren zwei Möglichkeiten, den Debugger zu verwenden.

Aufruf des Debuggers mit Angabe eines Klassennamens

Bei dieser Möglichkeit startet der Debugger den Java-Interpreter und lädt die als Parameter angegebene Klasse. Anschließend meldet sich der Debugger mit seinem Prompt ">" und wartet auf eine Eingabe des Benutzers. Dem Benutzer steht dann eine Reihe von Befehlen zur Fehlersuche zur Verfügung.

```
jdb [ <Optionen>] <Klassenname> <Argumentliste>
```

Anbinden des Debuggers an einen laufenden Interpreter

Diese Möglichkeit erlaubt das Remote Debugging von Java-Applikationen. Der Debugger bindet sich an einen laufenden Interpreter. Debugger und Interpreter können auf verschiedenen Rechnern gestartet werden. Der Interpreter muss hierzu mit der Option `-debug` gestartet werden und gibt dann anschliessend ein Passwort aus, das zum Anbinden des Debuggers an diesen Interpreter notwendig ist. Ist das Passwort korrekt, meldet sich der Debugger mit seinem Prompt `>` und wartet auf Benutzereingaben.

```
jdb [ <Optionen>]
```

Kommandos des Java-Debuggers

Nach dem Start des Debuggers meldet sich dieser stets mit seinem Prompt `>`. Der Debugger wartet dann auf Benutzereingaben. Eine Liste der unterstützten Kommandos gibt der Debugger nach Eingabe des Kommandos `help` aus.

Kommando	**Beschreibung**
threads [<threadgroup>]	Erzeugt eine Liste der aktiven Threads. Jeder Thread wird durch eine eindeutige ganzzahlige Zahl, die <thread id>, identifiziert. Die optionale Angabe einer Threadgruppe erzeugt eine Liste der aktiven Threads innerhalb dieser Gruppe.
thread <thread id>	Definiert den aktuellen Thread.
suspend [<thread id(s)>]	Hält den Thread <thread id> an. Wird keine <thread id> angegeben, werden alle Threads angehalten.
resume [<thread id(s)>]	Führt den Thread <thread id> fort. Wird keine <thread id> angegeben, werden alle Threads fortgesetzt.
where [<thread id>] | all	Druckt den Stack des durch das thread-Kommando definierten Thread aus. Dabei wird sichtbar, in welcher Methode sich das Programm befindet und welche Methoden in der Aufrufhierarchie darüber liegen.
Threadgroups	Erzeugt eine Liste der aktiven Threadgruppen.
Threadgroup <name>	Definiert die aktuelle Threadgruppe.
print <id> [<id(s)>]	Ruft die Methode `toString()` eines Objektes auf. Das Objekt wird durch seine <id> identifiziert.

dump <id> [<id(s)>]	Zeigt den Inhalt der Felder eines Objektes an. Das Objekt wird durch seine <id> identifiziert.
Locals	Erzeugt eine Liste der lokalen Variablen des aktuellen Thread. Der aktuelle Thread wird mit Hilfe des thread Kommandos gesetzt. Diese Liste kann nur generiert werden, wenn die Klasse mit der Option -g übersetzt wurde.
Classes	Erzeugt eine Liste aller in dem Programm verwendeten Klassen. Jeder Klasse wird eine <class id> vorangestellt. Die <class id> muß bei einigen Kommandos als Argument angegeben werden.
methods <class id>	Erzeugt eine Liste mit den Methoden einer Klasse.
stop in <class id>.<method>	Definiert einen Breakpoint zu Beginn einer Methode. Die <class id> kann mit Hilfe des classes Kommandos ermittelt werden.
stop at <class id>:<line>	Definiert einen Breakpoint an einer Zeile im Quelltext. Die <class id> kann mit Hilfe des classes Kommandos ermittelt werden.
up [<n frames>] down [<n frames>]	Erlaubt das gezielte Springen innerhalb des Stacks. Der Stack kann mit Hilfe des where Kommandos ermittelt werden. Der Parameter gibt an, wieviel Methoden überspringen werden sollen.
clear <class id>:<line>	Löschen eines Breakpoints.
Step	Führt die nächste Zeile aus. Dieses Kommando ermöglicht das Abarbeiten des Programms Zeile für Zeile.
Cont	Setzt die Ausführung des Programms bis zum nächsten Breakpoint fort.
Catch <class id>	Dieses Kommando ermöglicht das Abfangen von Exceptions innerhalb des Debuggers. Die Exception wird durch die class id identifiziert. Die class id wird durch das Kommando classes ermittelt.
Ignore <class id>	Hebt das durch das Catch Kommando spezifizierte Abfangen einer Exception innerhalb des Debuggers auf. Argument ist die <class id> der zu ignorierenden Exception.
list [<line number>]	Erzeugt ein Listing des Java Quelltextes. Diese Option wird nur unterstützt, wenn das Programm mit der Option -g übersetzt wurde.
use [<source file path>]	Ohne Parameter zeigt dieses Kommando den verwendeten Pfad für die Quelltextdateien an. Wird ein Parameter angegeben, spezifiziert dieser den neu zu verwendenden Pfad.

Memory	Zeigt den insgesamt zur Verfügung stehenden Speicher und den belegten Speicher an.
Gc	Dieses Kommando startet den Garbage Collector. Alle nicht mehr referenzierten Objekte werden vom Heap entfernt.
load <classname>	Dieser Befehl wird benötigt, wenn der Debugger ohne Angabe eines Klassennamens gestartet wird. Die hinter dem Kommando angegebene Klasse wird geladen und kann untersucht werden.
run <class> [<args>]	Startet die Ausführung einer geladenen Klasse. Der Klassenname kann im Klartext angegeben werden. Die <class id> wird nicht benötigt. Hinter dem Kommando können Parameter für die `main()`-Funktion angegeben werden.
!!	Dieser Befehl wiederholt das zuletzt ausgeführte Kommando.
help (oder ?)	Erzeugt eine Liste der unterstützten Kommandos.
exit (oder quit)	Dieser Befehl beendet den Debugger.

Optionen des Java-Debuggers

Wird der Debugger als Ersatz für den Interpreter verwendet (lokales Debugging), akzeptiert `jdb` die gleichen Optionen wie der Java-Interpreter. Die beim Aufruf des Debuggers angegebenen Optionen werden an den gestarteten Interpreter weitergereicht.

Zum Anbinden des Debuggers an einen laufenden Interpreter sind die folgenden zwei Optionen anzugeben:

Option	Beschreibung	Standardeinstellung
-host	Spezifiziert den Host, auf dem der Interpreter läuft.	-
-password	Spezifiziert das Passwort, welches der Interpreter nach dem Start mit der Option `-debug` ausgegeben hat. Ohne dieses Passwort ist eine Anbindung des Debuggers an den Interpreter nicht möglich.	-

muß beim Aufruf des Debuggers der Rechner, auf dem der Interpreter läuft, und das vom Interpreter ausgewiesene Passwort angegeben werden:

```
jdb -host morgaine -password 5shzr8
```

Erstes Standalone-Beispiel

Wenden wir uns nun unserem ersten Java-Programm zu. Beachten Sie bitte Groß- und Kleinschreibung!

```
class MeinErstes                              //1

{

  public static void main(String args [])          //3

  {

    System.out.print("\nMit print und println kann");   //5

    System.out.println(" man schreiben\nund rechnen:");//6

    System.out.println("6 * 7 = \t" + 6 * 7);        //7

  }

}
```

Alles was innerhalb einer Zeile hinter doppelten Schrägstrichen (//) steht, wird von Java als Kommentar betrachtet.

Speichern Sie MeinErstes unter der Datei MeinErstes.java und compilieren sie es anschliessend.

Starten sie das Programm! Auf der DOS-Ebene lautet der Befehl

```
java MeinErstes
```

Die Ausgabe des Programms sieht wie folgt aus:

Mit print und println kann man schreiben und rechnen:

```
6 * 7 =       42
```

Ein Java-Programm besteht aus Klassen. Jede Klassendefinition wird mit dem Schlüsselwort *class* eingeleitet. Dahinter folgt der Name der Klasse

Die Definition einer Klasse beginnt mit einer offenen und endet mit einer geschlossenen geschweiften Klammer. Klammern dienen zur Bildung eines Blocks von Anweisungen.

Damit eine Application starten kann, muss sie mindestens eine Funktion enthalten, und der Name der Funktion muss *main* sein.

Die Begriffe *public, static* und *void* müssen hier in dieser Reihenfolge erscheinen. Ihre Bedeutung wird an gegebener Stelle ausführlich erläutert. So sagt *void* aus, dass die Funktion *main* keinen Wert zurückliefert. *static* besagt, dass *main* zu einer Klasse gehört und nicht zu individuellen Objekten, und *public* sagt aus, welche Teile eines Programms diese Funktion benutzen dürfen.

Der Ausdruck *String args[]* definiert ein Feld von Zeichenketten (args). In diese Zeichenketten werden die Parameter der Kommandozeile eingestellt, die beim Aufruf des Programms angegeben wurden

z.B. *java MeinErstes abc 123*

dann werden die Parameter abc und 123 an das Programm übergeben und sind in *args[0]* und *args[1]* ansprechbar.

Die Funktion *main* beginnt mit geschweiften Klammern und endet mit geschweiften Klammern.

Der Compiler orientiert sich nicht an Zeilenenden. So könnten sie den Befehl

```
System.out.println(" man schreiben\nund rechnen:"); //6
```

auch schreiben als

```
System.
        out.println(" man schreiben\nund rechnen:");  //6
```

oder

```
System.out.println
        (" man schreiben\nund rechnen:");//6
```

An Hand des Semikolon erkennt der Compiler das Befehlsende.

Die Funktion zum am Bildschirmausdrucken lautet *System.out.print* bzw. *System.out.println*.

println macht nach dem gedruckten Text einen Zeilenvorschub, während *print* an der aktuellen Bildschirmposition stehen bleibt.

Der auszugebende Text steht in Anführungszeichen. Innerhalb des Textes kann manEscape-Sequenzen zur Textsteuerung einfügen

\b	Backspace	\t	horizontal tab
\n	Line feed	\f	form feed
\r	Carriage return	\"	Anführungszeichen
\´	Hochkomma	\ß	backslash \

In Zeile 7 erfolgt die eigentliche Berechnung. Die Zeichenkette und die numerischen Werte werden durch ein Pluszeichen (+) verknüpft. Damit wird der numerische Wert automatisch in eine Zeichenkette umgewandelt.

Um Ausdrücke zu addieren, zu multiplizieren usw., kennt Java folgende **arithmetische Operatoren**

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo

Frage: In Sprachen wie HTML ist es egal, ob ich die Schlüsselwörter gross schreibe. Wie steht es damit in Java. Eine zweite Frage, wie schreibe ich Kommentare?

Java unterscheidet zwischen **Gross- und Kleinschreibung**.

`MEINERSTES` ist etwas anderes als `MeinErstes`. Auch ist `System.out.print` etwas anderes als `System.Out.print`.

Kommentare sind einerseits // (einzeilige Kommentare) bzw. /* (hier kann man einen Kommentar beginnen, der über mehrere Zeilen geht und mit */ beendet wird).

Literatur:

- Christian Ullenboom, Java ist auch eine Insel, Programmieren mit der Standard Edition Version 8, Kap. 1.6, 1.7

Kontrollfragen:

1. Was ist an der folgenden Zeile falsch?

```
public static void Main(String args [])
```

2. Welche Escape-Sequenzen muss man in die Zeichenkette

```
System.out.println("abcdefg");
```

einfügen, damit folgende Ausgabe erscheint

```
aceg
```

3. Welche Aussagen zum Java Compiler sind richtig?

Der Java Compiler generiert Programme, die

a) auf jeder Maschine direkt ausführbar sind.
b) in Abhängigkeit von der Zielarchitektur nur auf bestimmten Maschinen ausführbar sind.
c) in einem Java Interpreter ausführbar sind.

4. Welche Aussagen zum Java-Interpreter sind richtig ?

Der Java Interpreter wird benötigt, um

a) Applets auszuführen.
b) Applikationen auszuführen.
c) sowohl Applets als auch Applikationen auszuführen.

5. Welche Aussagen zum Java Debugger jdb treffen zu?

a) Er kann an Stelle des Interpreters verwendet werden und ermöglicht die Fehlersuche in Programmen.
b) Es können nur Applikationen mit Hilfe des Debuggers bearbeitet werden.
c) Es können sowohl Applikationen als auch Applets mit Hilfe des Debuggers bearbeitet werden. Zum Debuggen von Applets ist der Appletviewer mit der Option -debug zu starten.
d) Zum Debuggen von Java-Programmen ist dem Debugger ein durch den Programmierer spezifiziertes Passwort mitzuteilen.
e) Zum Remote Debuggen von Programmen ist dem Debugger das vom Interpreter ausgegebene Passwort mitzuteilen.

6. Welche Aussagen zum Appletviewer treffen zu?

a) Er wird benötigt, um Java-Programme innerhalb eines Fensters anzeigen zu lassen.
b) Er wird benötigt, um Java Applets ohne einen WWW-Browser auszuführen.
c) Der Appletviewer wird mit einem Klassennamen als Parameter aufgerufen.
d) Der Appletviewer wird mit einer URL als Parameter aufgerufen.

7. Können innerhalb einer Java Quelltextdatei mehrere öffentliche Klassen deklariert werden?

a) Ja
b) Nein

8. Wozu verwenden Sie Kommentare, die durch /** eingeleitet und */ beendet werden?

e) Zum Kennzeichnen mehrzeiliger Kommentare
f) Diese Form ist analog zu /* und */
g) Kommentare zur automatischen Generierung einer Programmdokumentation im HTML Format