

*Ich habe mir immer gewünscht, dass sich Computer so einfach bedienen lassen wie Telefone. Mein Wunsch hat sich erfüllt. Ich weiß nicht mehr, wie ich mein Telefon bedienen muss.*

Bjarne Stroustrup (um 1990), Erfinder der Programmiersprache C++

# 1

## Grundlegende Sprachelemente von Java

### Kapitelübersicht

1.1	Einführung . . . . .	5
1.2	Das erste Programm: Ausgabe eines Textes . . . . .	8
1.3	Elemente eines Java-Programms . . . . .	11
1.4	Strings und Ausgaben . . . . .	15

### 1.1 Einführung

Java ist eine objektorientierte Programmiersprache. Viele ihrer Fähigkeiten und einen großen Teil der Syntax hat sie von C und C++ geerbt. Im Gegensatz zu diesen Sprachen ist Java jedoch nicht primär darauf konzipiert, möglichst kompakte und optimal auf die zugrunde liegende Hardware angepasste Programme zu erzeugen, sondern soll die Programmierer vor allem dabei unterstützen, sichere und fehlerfreie Programme zu schreiben. Daher sind einige der Fähigkeiten von C++ nicht übernommen worden, die zwar mächtig, aber auch fehlerträchtig sind. Viele Konzepte älterer objektorientierter Sprachen (wie z.B. Smalltalk) wurden übernommen. Java vereint also bewährte Konzepte früherer Sprachen. Siehe dazu auch den Stammbaum in Abb. 1.1.

Java wurde ab 1991 bei der Firma Sun Microsystems entwickelt. In einem internen Forschungsprojekt namens *Green* unter der Leitung von James Gosling sollte eine Programmiersprache zur Steuerung von Geräten der Haushaltselektronik entstehen. Gosling nannte die Sprache zunächst *Oak*, da eine Eiche vor seinem Büfenster stand. Als später entdeckt wurde, dass es bereits eine Programmiersprache mit diesem Namen gab, wurde ihr angeblich in einem Café der Name *Java* gegeben. *Java* ist ein umgangssprachliches Wort für Kaffee.

Da sich, entgegen den Erwartungen der Strategen von Sun, der Markt für intelligente Haushaltsgeräte nicht so schnell und tiefgreifend entwickelte, hatte das Green-Projekt große Schwierigkeiten. Es war in Gefahr, abgebrochen zu werden. So war es purer Zufall, dass in dieser Phase 1993 das World Wide Web (WWW) explosionsartig populär wurde. Bei Sun wurde sofort das Potential von Java erkannt, interaktive („dynamische“) Web-Seiten zu erstellen. So wurde dem Projekt schlagartig neues Leben eingehaucht.

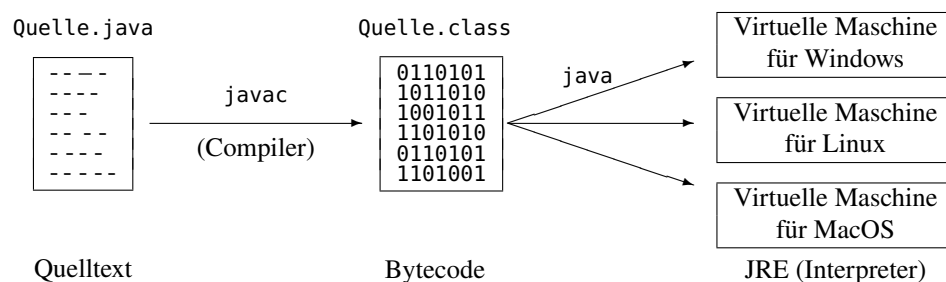


sofort in den Arbeitsspeicher (RAM, *random access memory*) des Computers übersetzt und ausgeführt. Beispiele für Compilersprachen sind C oder C++, Beispiele für Interpretersprachen sind Perl, PHP oder JavaScript. Ein Vorteil von Compilersprachen ist, dass ein kompiliertes Programm schneller („performanter“) als ein vergleichbares zu interpretierendes Programm abläuft, da die Übersetzung in die Maschinensprache des Betriebssystems ja bereits durchgeführt ist, denn ein Interpreter geht das Programm zeilenweise durch und muss jede Anweisung erst übersetzen, bevor er sie ausführt.

Ein weiterer Vorteil der Compilierung aus der Sicht des Programmierers ist, dass ein Compiler Fehler der *Syntax* erkennt, also Verstöße gegen die Grammatikregeln der Programmiersprache. Man hat also bei einem kompilierten Programm die Gewissheit, dass die Syntax korrekt ist, es können also höchstens noch Laufzeitfehler (z.B. nicht verarbeitbare Eingabedaten, Division durch 0) oder logische Fehler enthalten sein. Bei einer Interpretersprache dagegen läuft bei einem Programmierfehler das Programm einfach nicht, und die Ursache ist nicht näher eingegrenzt. Compilersprachen ermöglichen also eine sicherere Entwicklung.

Andererseits haben Compilersprachen gegenüber Interpretersprachen einen großen Nachteil bezüglich der Plattformunabhängigkeit. Für jede Plattform, also jedes Betriebssystem (Windows, Linux, MacOS), auf der das Programm laufen soll, muss eine eigene Compilerdatei erstellt werden; bei Interpretersprachen kann der Quelltext von dem ausführenden Rechner sofort interpretiert werden (vorausgesetzt natürlich, der Interpreter ist auf dem Rechner vorhanden).

Welche der beiden Ausführprinzipien verwendet Java? Nun, wie bei jedem vernünftigen Kompromiss — beide! Der Quelltext eines Java-Programms steht in einer Textdatei mit der



**Abbildung 1.3.** Vom Quellcode zur Ausführung eines Java-Programms (JRE = Java Runtime Environment)

Endung `.java`. Er wird von einem Compiler namens `javac` in eine Datei umgewandelt, deren Namen nun die Endung `.class` trägt. Sie beinhaltet jedoch nicht, wie sonst bei kompilierten Dateien üblich, ein lauffähiges Programm, sondern den so genannten *Bytecode*. Dieser Bytecode läuft auf keiner realen Maschine, sondern auf der *virtuellen Maschine (JVM)*. Das ist de facto ein Interpreter, der für jedes Betriebssystem den Bytecode in den RAM des Computers lädt und das Programm ablaufen lässt. Die Virtuelle Maschine wird durch den Befehl `java` aufgerufen und führt den Bytecode aus. Abb. 1.3 zeigt schematisch die einzelnen Schritte vom Quellcode bis zur Programmausführung. Die JVM ist Teil der *Java Runtime Environment (JRE)*, die es für jedes gängige Betriebssystem gibt.

## 1.1.2 Installation der Java SDK-Umgebung

Ein Java-Programm wird auch *Klasse* genannt. Um Java-Programme zu erstellen, benötigt man die JDK-Umgebung für das Betriebssystem, unter dem Sie arbeiten möchten. (JDK = *Java Development Kit*) Es beinhaltet die notwendigen Programme, insbesondere den Compiler `javac`, den Interpreter `java` und die Virtuelle Maschine. Das JDK ist als freie Software erhältlich unter

<https://jdk.java.net/>

Nach erfolgter Installation<sup>1</sup> wird die Datei mit der Eingabe des Zeilenkommandos

```
javac Klassenname.java
```

compiliert. Entsprechend kann sie dann mit der Eingabe

```
java Klassenname
```

ausgeführt werden (ohne die Endung `.class`!).

### 1.1.3 Erstellung von Java-Programmen

Java-Programme werden als Text in Dateien eingegeben. Hierzu wird ein Texteditor oder eine „integrierte Entwicklungsumgebung“ (*IDE = Integrated Development Environment*) wie IntelliJ oder Eclipse,

<http://www.intellij.org> oder <http://www.eclipse.org>

benutzt. Die IDE's sind geeignet für die fortgeschrittene Programmierung mit grafischen Bedienungselementen oder für größere Software-Projekte. Wir werden zunächst die einfache Programmerstellung mit einem Texteditor verwenden, um die Prinzipien der Kompilier- und Programmstartprozesse zu verdeutlichen. Später werden wir professionellere Werkzeuge einsetzen, weil sie uns ermöglichen besser zu verstehen, was wirklich im System passiert.

## 1.2 Das erste Programm: Ausgabe eines Textes

Wir beginnen mit einer einfachen *Applikation*, die eine Textzeile ausgibt. Dieses (mit einem beliebigen Texteditor) erstellte Programm ist als die Datei `Willkommen.java` abgespeichert und lautet:

```
1 public class Willkommen {  
2     public static void main(String[] args) {  
3         javax.swing.JOptionPane.showMessageDialog(null, "Hallo Welt!");  
4     }  
5 }
```

Das Programm gibt nach dem Start das Fenster in Abbildung 1.4 auf dem Bildschirm aus.



Abbildung 1.4. Ergebnis der Applikation `Willkommen`.

---

<sup>1</sup>Installationshinweise siehe <http://haegar.fh-swf.de/Java/HowTos/Installation-OpenJDK.pdf>

### 1.2.1 Wie kommt die Applikation ans Laufen?

Wie in der Einleitung beschrieben (Abb. 1.3 auf S. 7), muss zunächst aus der `Willkommen.java`-Datei (dem „Source-Code“ oder Quelltext) eine `Willkommen.class`-Datei im Bytecode mit dem Compiler `javac` erstellt werden, und zwar mit der Eingabe des Zeilenkommandos

```
javac Willkommen.java
```

Entsprechend kann sie dann mit der Eingabe

```
java Willkommen
```

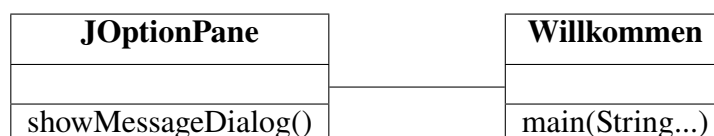
ausgeführt werden.

### 1.2.2 Ein erster Überblick: So funktioniert das Programm

Bevor wir die einzelnen Programmbestandteile genauer betrachten, sollten wir uns die grobe Struktur einer Java-Applikation verdeutlichen.

1. Ein Java-Programm ist stets eine *Klasse* und beginnt mit den Schlüsselworten **public class** sowie dem Namen der Klasse (Zeile 5).
2. Ein Java-Programm ist in Einheiten gepackt, sogenannte *Blöcke*, die von geschweiften Klammern `{ ... }` umschlossen sind und oft vorweg mit einer Bezeichnung versehen sind (z.B. `public class Willkommen` oder `public static void main(...)`). Zur Übersichtlichkeit sind die Blöcke stets geeignet einzurücken.
3. Der Startpunkt jeder Applikation ist die Methode `main`. Hier beginnt der Java-Interpreter (die „virtuelle Maschine“, siehe S. 7), die einzelnen Anweisungen auszuführen. Er arbeitet sie „sequenziell“, d.h. der Reihe nach, ab (Zeile 6).
4. Die Applikation besteht aus einer einzigen Anweisung (Zeile 7): Sie zeigt einen Text in einem Fenster an.
5. Um Spezialfunktionen zu verwenden, die von anderen Entwicklern erstellt wurden, kann man mit der Anweisung **import** bereits programmierte Klassen importieren. Eine Klasse ist in diesem Zusammenhang also so etwas wie ein „Werkzeugkasten“, aus dem wir fertig programmierte „Werkzeuge“ (hier die Methode `showMessageDialog` aus `JOptionPane`) verwenden können. Eine der wenigen Klassen, die wir nicht importieren müssen, ist die Klasse `System`.

Halten wir ferner fest, dass die Grobstruktur unserer kleinen Applikation durch das folgende Diagramm dargestellt werden kann.



Es ist ein *Klassendiagramm*. Es stellt grafisch dar, dass unsere Klasse `Willkommen` die Methode `main` hat und die Klasse `JOptionPane` „kennt“, deren Methode `showMessageDialog` sie verwendet.

### 1.2.3 Die Klassendeklaration und Klassennamen

In der Zeile 5 beginnt die *Klassendeklaration* der Klasse `Willkommen`. Jedes Java-Programm besteht aus mindestens einer Klasse, die definiert ist von dem Programmierer. Die reservierten Worte `public class` eröffnen die Klassendeklaration in Java, direkt gefolgt von dem *Klassennamen*, hier `Willkommen`. *Reservierte Wörter* sind von Java selber belegt und erscheinen stets mit kleinen Buchstaben. Eine Liste aller reservierten Wörter in Java befindet sich in Tabelle 1.1 auf Seite 12 und am Anfang des Indexverzeichnisses des Skripts.

Es gilt allgemein die Konvention, dass alle Klassennamen in Java mit einem Großbuchstaben beginnen und dass jedes neue Wort im Namen ebenfalls mit einem Großbuchstaben beginnt, z.B. `BeispielKlasse` oder `JOptionPane` (*option* = Auswahl, *pane* = Fensterscheibe; das `J` kennzeichnet alle so genannten „Swing-Klassen“).

Der Klassenname ist ein so genannter *Identifizier* oder *Bezeichner*. Ein Bezeichner ist eine Folge von alphanumerischen Zeichen (*Characters*), also Buchstaben, Ziffern, dem Unterstrich (`_`) und dem Dollarzeichen (`$`). Ein Bezeichner darf nicht mit einer Ziffer beginnen und keine Leerzeichen enthalten. Erlaubt sind also

`Raketel`, oder `$wert`, oder `_wert`, oder `Taste7`.

Demgegenüber sind `7Taste` oder `erste Klasse` als Klassennamen *nicht* erlaubt. Im übrigen ist Java schreibungssensitiv (*case sensitive*), d.h. es unterscheidet strikt zwischen Groß- und Kleinbuchstaben — `al` ist also etwas völlig anderes als `A1`.

**Merkregel 1.** Eine Klasse in Java muss in einer Datei abgespeichert werden, die genau so heißt wie die Klasse, mit der Erweiterung „.java“. Allgemein gilt folgende Konvention: Ein Klassenname beginnt mit einem Großbuchstaben, enthält keine Sonderzeichen oder Umlaute und besteht aus einem zusammenhängenden Wort. Verwenden Sie einen „sprechenden“ Namen, der ausdrückt, was die Klasse bedeutet.

In unserem ersten Programmbeispiel heißt die Datei `Willkommen.java`.

### 1.2.4 Der Programmstart: Die Methode `main`

Wie startet eine Applikation in Java? Die zentrale Einheit einer Applikation, gewissermaßen die „Steuerzentrale“, ist die *Methode* `main`. Durch sie wird die Applikation gestartet, durch die dann alle Anweisungen ausgeführt werden, die in ihr programmiert sind. Die Zeile 6,

`public static void main(String... args)` (1.1)

ist Teil jeder Java-Applikation. Java-Applikationen beginnen beim Ablaufen automatisch bei `main`. Die runden Klammern hinter `main` zeigen an, dass `main` eine *Methode* ist. Klassendeklarationen in Java enthalten normalerweise mehrere Methoden. Für Applikationen muss davon *genau eine* `main` heißen und so definiert sein wie in (1.1). Andernfalls wird der Java-Interpreter das Programm nicht ausführen. Nach dem Aufruf der Methode `main` mit dem „Standardargument“ `String[] args` kommt der *Methodenrumpf* (*method body*), eingeschlossen durch geschweifte Klammern (`{ ... }`). Über den tieferen Sinn der langen Zeile `public static void main (String[] args)` sollten Sie sich zunächst nicht den Kopf zerbrechen. Nehmen Sie sie (für's erste) hin wie das Wetter!

Eine allgemeine Applikation in Java muss also auf jeden Fall die folgende Konstruktion beinhalten:

```
public class Klassenname {
    public static void main(String... args) {
        Deklarationen und Anweisungen;
    }
}
```

Das ist gewissermaßen die „Minimalversion“ einer Applikation. In unserem Beispiel bewirkt die Methode `main`, dass der Text „Willkommen zur Java-Programmierung“ ausgegeben wird.

Man kann übrigens auch statt „`String... args`“ auch drei Punkte für die eckige Klammer verwenden und „`String[] args`“ schreiben, was für deutsche Tastaturen etwas weniger angenehm ist.

```
public class Klassenname {
    public static void main(String[] args) {
        Deklarationen und Anweisungen;
    }
}
```

In diesem Skript, in der Vorlesung und in den Praktika werden beide Versionen verwendet. Welche Version Sie selbst benutzen, bleibt Ihnen überlassen.

## 1.3 Elemente eines Java-Programms

### 1.3.1 Anweisungen und Blöcke

Jedes Programm besteht aus einer Folge von *Anweisungen* (*instruction*, *statement*), wohldefinierten kleinschrittigen Befehlen, die der Interpreter zur Laufzeit des Programms ausführt. Jede Anweisung wird in Java mit einem Semikolon (;) abgeschlossen. Vergleichen wir Deklarationen und Anweisungen, so stellen wir fest, dass Deklarationen die Struktur des Programms festlegen, während die Anweisungen den zeitlichen Ablauf definieren:

Ausdruck	Wirkung	Aspekt
Deklaration	Speicherreservierung	statisch (Struktur)
Anweisung	Tätigkeit	dynamisch (Zeit)

Anweisungen werden zu einem *Block* (*block*, *compound statement*) zusammengefasst, der durch geschweifte Klammern ({ und }) umschlossen ist. Innerhalb eines Blockes können sich weitere Blöcke befinden. Zu beachten ist, dass die dabei Klammern stets korrekt geschlossen sind. Die Blöcke eines Programms weisen also stets eine hierarchische logische Baumstruktur aufweisen.

**Merkregel 2.** Zur besseren Lesbarkeit Ihres Programms und zur optischen Strukturierung sollten Sie jede Zeile eines Blocks stets gleich weit **einrücken**. Ferner sollten Sie bei der Öffnung eines Blockes mit { sofort die geschlossene Klammer } eingeben.

Auf diese Weise erhalten Sie sofort einen blockweise hierarchisch gegliederten Text. So können Sie gegebenenfalls bereits beim Schreiben falsche oder nicht gewollte Klammerungen (Blockbildungen!) erkennen und korrigieren. Falsche Klammerung ist keine kosmetische Unzulänglichkeit, das ist ein echter Programmierfehler.

Es gibt zwei verbreitete Konventionen, die öffnende Klammer zu platzieren: Nach der einen Konvention stellt man sie an den Anfang derjenigen Spalte, auf der sich der jeweils übergeordnete

Block befindet, und der neue Block erscheint dann eingerückt in der nächsten Zeile, so wie in dem folgenden Beispiel.

```
public class Willkommen
{
    ...
}
```

Die andere Konvention zieht die Klammer zu der jeweiligen Anweisung in der Zeile darüber hoch, also

```
public class Willkommen {
    ...
}
```

In diesem Skript wird vorwiegend die zweite der beiden Konventionen verwendet, obwohl die erste etwas lesbarer sein mag. Geschmackssache eben.

### 1.3.2 Reservierte Wörter und Literale

Die Anweisungen eines Programms enthalten häufig *reservierte Wörter*, die Bestandteile der Programmiersprache sind und eine bestimmte vorgegebene Bedeutung haben. Die in Java reservierten Wörter sind in Tabelle 1.1 zusammengefasst.

#### Reservierte Wörter

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

#### Literale

false	true	null	0, 1, -2, 5L	1.2, .0, 2.0f	'A', 'b', ' '	"Abc", ""
-------	------	------	--------------	---------------	---------------	-----------

#### In Java nicht verwendete reservierte Wörter

byvalue	cast	const	future	generic	goto	inner
operator	outer	rest				

**Tabelle 1.1.** Die reservierten Wörter in Java

Dazu gehören auch sogenannte *Literale*, z.B. die Konstanten `true`, `false` und `null`, deren Bedeutung wir noch kennen lernen werden. Allgemein versteht man unter einem Literal einen konstanten Wert, beispielsweise also auch eine Zahl wie 14 oder 3.14, ein Zeichen wie `'a'` oder einen String („Text“) wie `"Hallo!"`.

Daneben gibt es in Java reservierte Wörter, die für sich gar keine Bedeutung haben, sondern ausdrücklich (noch) nicht verwendet werden dürfen.

Allgemein bezeichnet man die Regeln der Grammatik einer Programmiersprache, nach denen Wörter aneinander gereiht werden dürfen, mit dem Begriff *Syntax*. In der Syntax einer Programmiersprache spielen die reservierten Wörter natürlich eine Schlüsselrolle, weshalb man sie auch oft „Schlüsselwörter“ nennt.



### 1.3.3 Kommentare

Ein *Kommentar* (*comment*) wird in ein Programm eingefügt, um dieses zu dokumentieren und seine Lesbarkeit zu verbessern. Insbesondere sollen Kommentare es erleichtern, das Programm zu lesen und zu verstehen. Sie bewirken keine Aktion bei der Ausführung des Programms und werden vom Java-Compiler ignoriert. Es gibt drei verschiedene Arten, Kommentare in Java zu erstellen, die sich durch die Sonderzeichen unterscheiden, mit denen sie beginnen und ggf. enden müssen.

- `// ...` Eine Kommentartyp ist der *einzeilige Kommentar*, der mit dem Doppelslash `//` beginnt und mit dem Zeilenende endet. Er wird nicht geschlossen. Quellcode, der in derselben Zeile *vor* den Backslashes steht, wird jedoch vom Compiler ganz normal verarbeitet.
- `/* ... */` Es können auch mehrere Textzeilen umklammert werden: `/*` und `*/`, z.B.:

```
1  /* Dies ist ein Kommentar, der
2     sich über mehrere Zeilen
3     erstreckt. */
```
- `/** ... */` Die dritte Kommentartyp `/** ... */` in den Zeilen 2 bis 4 ist der *Dokumentationskommentar* (*documentation comment*). Sie werden stets direkt vor den Deklarationen von Klassen, Attributen oder Methoden (wir werden noch sehen, was das alles ist...) geschrieben, um sie zu beschreiben. Die so kommentierten Deklarationen werden durch das Programm javadoc-Programm automatisch verarbeitet.

### 1.3.4 Programmierstil: Strukturierung durch Einrücken

Ein Leerzeichen trennt verschiedene Wörter, jedes weitere Leerzeichen direkt dahinter hat jedoch keine Wirkung. Ebenso werden in Java werden Leerzeilen und Tab-Stops vom Compiler nicht verarbeitet. Diese Zeichen heißen *Whitespace* oder *Leerraum*.

Wozu dann extra Leerraum? Das folgende Programm lässt sich compilieren und ausführen:

```
1 public class Willkommen{public static void main(String... args) {javax.
2 swing.JOptionPane.showMessageDialog(null, "Hallo Welt!");}}
```

Der Compiler kann es lesen, aber können Sie das auch? Fügen wir geeignet Leerraum ein, so ergibt sich das folgende Programm:

```
1 public class Willkommen{
2     public static void main(String... args) {
3         javax.swing.JOptionPane.showMessageDialog(null, "Hallo Welt!");
4     }
5 }
```

Das ist auf jeden Fall leichter lesbar. Zusätzlicher Leerraum kann (und soll!) also zur Strukturierung und Lesbarkeit des Programms verwendet werden.

### 1.3.5 import-Anweisung

Generell können fertige Java-Programme zur Verfügung gestellt werden. Dazu werden sie in Verzeichnissen, den sogenannten *Paketen* gespeichert. Beispielsweise sind die die Klassen des wichtigen Swing-Pakets `javax.swing` in dem Verzeichnis `/javax/swing` (bei UNIX-Systemen)

bzw. \javax\swing (bei Windows-Systemen) gespeichert.<sup>2</sup> Insbesondere sind alle Programme des Java-API's in Paketen bereitgestellt.

Um nun solche bereitgestellten Programme zu verwenden, müssen sie ausdrücklich importiert werden. Das geschieht mit der `import`-Anweisung. Beispielsweise wird mit

```
import javax.swing.*;
```





Diese Anweisung muss stets am Anfang eines Programms stehen, wenn Programme aus dem entsprechenden Paket verwendet werden sollen. Das einzige Paket, das nicht eigens importiert werden muss, ist das Paket `java.lang`, welches die wichtigsten Basisklassen enthält. Die `import`-Anweisungen sind die einzigen Anweisungen eines Java-Programms, die *außerhalb* eines Blocks stehen.

### 1.3.6 \* Dialogfenster mit anderen Icons

Es gibt noch eine weitere Version der Methode `showMessageDialog`, die nicht mit zwei, sondern mit vier Parametern aufgerufen wird und mit denen man die erscheinenden Symbole variieren kann, beispielsweise durch die Anweisung

```
JOptionPane.showMessageDialog(null, "Hallo?", "Frage", JOptionPane.PLAIN_MESSAGE);
```

Der dritte Parameter (hier: "Frage") bestimmt den Text, der in der Titelleiste des Fensters erscheinen soll. Der vierte Parameter (`JOptionPane.PLAIN_MESSAGE`) ist ein Wert, der die Anzeige des Message-Dialogtyp bestimmt — dieser Dialogtyp hier zeigt kein Icon (Symbol) links von der Nachricht an. Die möglichen Message-Dialogtypen sind in der folgenden Tabelle aufgelistet:

Message-Dialogtyp	int-Wert	Icon	Bedeutung
<code>JOptionPane.ERROR_MESSAGE</code>	0		Fehlermeldung
<code>JOptionPane.INFORMATION_MESSAGE</code>	1		informative Meldung; der User kann sie nur wegklicken
<code>JOptionPane.WARNING_MESSAGE</code>	2		Warnmeldung
<code>JOptionPane.QUESTION_MESSAGE</code>	3		Fragemeldung
<code>JOptionPane.PLAIN_MESSAGE</code>	-1		Meldung ohne Icon

(Statt der langen Konstantennamen kann man auch kurz den entsprechenden `int`-Wert angeben.)  
Beispielsweise erscheint durch die Anweisung

```
JOptionPane.showMessageDialog(null, "Hallo?", "Frage", JOptionPane.QUESTION_MESSAGE);
```

oder kürzer

```
JOptionPane.showMessageDialog(null, "Hallo?", "Frage", 3);
```

ein Dialogfenster mit einem Fragezeichen als Symbol, dem Text "Hallo?" im Fenster und dem Text "Frage" in der Titelzeile. Das genaue Aussehen der Symbole hängt von dem jeweiligen Betriebssystem ab, unter Windows sehen sie anders als unter Linux oder macOS.

<sup>2</sup>Sie werden die Verzeichnisse und Dateien jedoch auf Ihrem Rechner nicht finden: Sie sind in kompakte Dateien komprimiert und haben in der Regel die Endung `.jar` für *Java Archive* oder `.zip` für das gebräuchliche Zip-Archiv.

## 1.4 Strings und Ausgaben

### 1.4.1 Strings

Der in die Anführungszeichen (") gesetzte Text ist ein *String*, also eine Kette von beliebigen Zeichen. In Java wird ein String stets durch die doppelten Anführungszeichen eingeschlossen, also

"... ein Text "

Innerhalb der Anführungszeichen kann ein beliebiges Unicode-Zeichen stehen (alles, was die Tastatur hergibt ...), also insbesondere Leerzeichen, aber auch „Escape-Sequenzen“ wie das Steuerzeichen `\n` für Zeilenumbrüche oder `\` für die *Ausgabe* von Anführungszeichen (s.u.).

Zur Darstellung von Steuerzeichen, wie z.B. einen Zeilenumbruch, aber auch von Sonderzeichen aus einer der internationalen Unicode-Tabellen, die Sie nicht auf Ihrer Tastatur haben, gibt es die *Escape-Sequenzen*: Das ist ein Backslash (`\`) gefolgt von einem (ggf. auch mehreren) Zeichen. Es gibt mehrere Escape-Sequenzen in Java, die wichtigsten sind in Tabelle 1.2

Escape-Zeichen	Bedeutung	Beschreibung
<code>\uxxxx</code>	Unicode	das Unicode-Zeichen mit Hexadezimal-Code <code>xxxx</code> („Codepoint“); z.B. <code>\u222B</code> ergibt $\int$
<code>\n</code>	line feed LF	neue Zeile. Der Bildschirmcursor springt an den Anfang der nächsten Zeile
<code>\t</code>	horizontal tab HT	führt einen Tabulatorsprung aus
<code>\\</code>	<code>\</code>	Backslash <code>\</code>
<code>\"</code>	<code>"</code>	Anführungszeichen <code>"</code>
<code>\'</code>	<code>'</code>	Hochkomma (Apostroph) <code>'</code>

**Tabelle 1.2.** Wichtige Escape-Sequenzen in Java

aufgelistet. Man kann also auch Anführungszeichen ausgeben:

```
JOptionPane.showMessageDialog(null, "\"Zitat Ende\"");
```

ergibt als Ausgabe **"Zitat Ende"**. Auch mathematische oder internationale Zeichen, die Sie nicht auf Ihrer Tastatur finden, können mit Hilfe einer Escape-Sequenz dargestellt werden. So ergibt beispielsweise

```
JOptionPane.showMessageDialog(null, "2 \u222B x dx = x\u00B2\n|\u2115| = \u2115");
```

die Ausgabe

$$\begin{array}{l} 2 \int x \, dx = x^2 \\ |\mathbb{N}| = \aleph \end{array}$$

Eine Auflistung der möglichen Unicode-Zeichen und ihrer jeweiligen Hexadezimalcodes finden Sie im Web zum Beispiel unter <http://www.isthishthingon.org/unicode/>.

### 1.4.2 Möglichkeiten der Ausgabe

In unserer Applikation ist der Ausgabetext ein String. Durch die Methode

```
JOptionPane.showMessageDialog(null, "...")
```

kann so ein beliebiger String auf dem Bildschirm angezeigt werden. Ein spezieller String ist der leere String `"`. Er wird uns noch oft begegnen.

Weitere einfache Möglichkeiten zur Ausgabe von Strings im Befehlszeilenfenster („Konsole“) bieten die Anweisungen `print` und `println` des Standard-Ausgabestroms `System.out`. Die Methode `print` gibt den eingegebenen String unverändert (und linksbündig) im Konsolenfenster aus. `println` steht für *print line*, also etwa „in einer Zeile ausdrucken“, und liefert dasselbe wie `print`, bewirkt jedoch nach der Ausgabe des Strings einen Zeilenumbruch. Beispiele:

```
System.out.print("Hallo, Ihr 2");
System.out.println("Hallo, Ihr " + 2);
```

Die `print`-Befehle geben auf der Konsole allerdings nur die ASCII-Zeichen aus, Umlaute oder Sonderzeichen können nicht ausgegeben werden. Die Ausgabe allgemeiner Unicode-Zeichen gelingt nur mit Hilfe von `JOptionPane` (bzw. anderen Swing-Klassen). Das folgende Programm fasst die Ausgabemöglichkeiten noch einmal zusammen:

```
1 import javax.swing.*;
2
3 public class Ausgaben {
4     public static void main(String[] args) {
5         // Ausgabe im Konsolenfenster:
6         System.out.print("Dies ist eine ");
7         System.out.println("Ausgabe mit " + (4+3) + " Wörtern.");
8
9         // Ausgabe in eigenem Fenster:
10        JOptionPane.showMessageDialog(
11            null, "Die ist eine Ausgabe \nmit " + 7 + " Wörtern."
12        );
13    }
14 }
```

### 1.4.3 Dokumentation der API

Eine *API* (*application programming Interface*: „Schnittstelle für die Programmierung von Anwendungsprogrammen“) ist allgemein eine Bibliothek von Programmen und Routinen („Methoden“), mit denen über selbsterstellte Programme auf Funktionen des Betriebssystems zugegriffen werden kann. Jede Programmiersprache muss eine API für das jeweilige Betriebssystem bereitstellen, auf dem die erstellten Programme laufen sollen. Die API ist also eine Zwischenschicht zwischen dem Betriebssystem und dem Programmierer.

Die API von Java stellt dem Programmierer im Wesentlichen Programme (meist Klassen) zur Verfügung, gewissermaßen „Werkzeugkästen“, mit deren Werkzeugen verschiedene Funktionen ermöglicht werden. Das wichtigste dieser „Werkzeuge“ ist die `main`-Methode, die das Ablaufen einer Applikation ermöglicht. Weitere wichtige Klassen sind beispielsweise `Math` für mathematische Funktionen, `JOptionPane` zur Programmierung von fensterbasierten Dialogen, oder `System` für systemnahe Funktionen.

Die Java-API besitzt eine detaillierte Dokumentation, die online verfügbar ist:

<https://docs.oracle.com/en/java/javase/21/docs/api/>

In Abbildung 1.5 ist ein Ausschnitt der Dokumentation für die Klasse `Math` gezeigt. Generell kann man eine Klasse über das Suchfeld eingeben. Man erkennt in dem Hauptfenster den unteren Teil der allgemeinen Beschreibung und die beiden für den Programmierer stets sehr wichtigen Elemente einer Klasse, ihre „Attribute“ (*fields*) und Methoden. Die Attribute der Klasse `Math` beispielsweise sind die beiden Konstanten `E` und `PI`, während die alphabetisch

The screenshot shows the Java API documentation for the `Math` class. At the top, there is a navigation bar with links: OVERVIEW, MODULE, PACKAGE, CLASS (highlighted), USE, TREE, DEPRECATED, INDEX, and HELP. The version is Java SE 13 & JDK 13. Below the navigation bar, there is a search bar and a summary section. The summary text states: "arithmetic operations consistently produce correct results, which in some cases means the operations will not overflow the range of values of the computation. The best practice is to choose the primitive type and algorithm to avoid overflow. In cases where the size is `int` or `long` and overflow errors need to be detected, the methods `addExact`, `subtractExact`, `multiplyExact`, and `toIntExact` throw an `ArithmeticException` when the results overflow. For other arithmetic operations such as `divide`, `absolute value`, `increment by one`, `decrement by one`, and `negation`, overflow occurs only with a specific minimum or maximum value and should be checked against the minimum or maximum as appropriate." Below this, it says "Since: 1.0".

**Field Summary**

Modifier and Type	Field	Description
static double	<code>E</code>	The double value that is closer than any other to $e$ , the base of the natural logarithms.
static double	<code>PI</code>	The double value that is closer than any other to $\pi$ , the ratio of the circumference of a circle to its diameter.

**Method Summary**

Modifier and Type	Method	Description
static double	<code>abs(double a)</code>	Returns the absolute value of a double value.
static float	<code>abs(float a)</code>	Returns the absolute value of a float value.
static int	<code>abs(int a)</code>	Returns the absolute value of an int value.

Abbildung 1.5. Ausschnitt aus der API-Dokumentation für die Klasse `Math`.

ersten Methoden die Funktionen `abs(a)` den Absolutbetrag des Parameters `a` und `acos(a)` den arccos des Parameters `a` darstellen.

In diesem Skript werden generell nur die wichtigsten Elemente einer Klasse oder eines Interfaces beschrieben. Sie sollten daher beim Programmieren stets auch in der API-Dokumentation nachschlagen. Sie sollten sich sogar angewöhnen, beim Entwickeln von Java-Software mindestens zwei Fenster geöffnet haben, den Editor und die Dokumentation im Browser.

## Mathematische Funktionen und Konstanten

Zusammengefasst stellt die Klasse `Math` also wesentliche Konstanten und mathematische Funktionen für die Programmierung in Java bereit. Sie liefert die Konstanten

$$\text{Math.E} \quad // \text{ Euler'sche Zahl } e = \sum_{k=0}^{\infty} \frac{1}{k!} \approx 2,718281828 \dots$$

$$\text{Math.PI} \quad // \text{ Kreiszahl } \pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left( \frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \approx 3,14159 \dots$$

und u.a. die mathematischen Funktionen

```
Math.abs(x)      // |x|
Math.acos(x)     // arccos x
Math.asin(x)     // arcsin x
Math.atan(x)     // arctan x
Math.atan2(x,y)  // arctan(x/y)
Math.cos(x)      // cos x
Math.exp(x)      // e^x (e hoch x)
Math.log(x)      // ln x (natürlicher Logarithmus)
Math.max(x,y)    // Maximum von x und y
Math.min(x,y)    // Minimum von x und y
```

```
Math.pow(x,y)    //  $x^y$  (x hoch y)  
Math.random()    // Pseudozufallszahl  $z$  mit  $0 \leq z < 1$   
Math.round(x)    // kaufmännische Rundung von  $x$  auf die nächste ganze Zahl  
Math.sin(x)      //  $\sin x$   
Math.sqrt(x)     //  $\sqrt{x}$   
Math.tan(x)      //  $\tan x$ 
```