Kapitel 03: Grundelemente der Programmierung

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

In diesem Kapitel werden wir erste Programme in der Programmiersprache Java schreiben. Wir werden die zwei grundlegenden Varianten zur Ausführung von Programmen vorstellen (Übersetzung und direkte Interpretation) und die Variante diskutieren, die für Java gewählt wurde.
Im Mittelpunkt stehen danach die Begriffe Wert, Ausdruck und Typ. Unsere ersten Java-Programme verwenden Datentypen, die in der Sprache integriert sind: primitive Datentypen (`int`, `double`, `boolean`, `char`) und Strings. Mit Hilfe von mathematischen und logischen Operatoren formulieren wir Ausdrücke und besprechen deren Auswertungssemantik. Als weiteres grundlegendes Programmierkonstrukt führen wir außerdem Variablen ein.
Als erste Interaktionsmöglichkeit mit dem Benutzer werden wir die Verwendung von Befehlszeilenargumente beim Programmstart und die Ausgabe auf Kommandozeile kennenlernen. Dies erfordert bisweilen die Umwandlung von Werten eines Typs in Werte eines anderen Typs.
Das Kapitel schließt mit einem Beispielprogramm, das die typische Verwendungen diese Konstrukte illustriert.

Lernziele dieses Kapitels:

- Den Unterschied zwischen Übersetzung und Interpretation zu beschreiben.
- Einfache Java-Programme zu erstellen, zu übersetzen und auszuführen.
- Daten durch geeignete Basisdatentypen zu modellieren.
- Boolsche und arithmetische Ausdrücke in Programmen zu verwenden.
- Befehlszeilenargumente zu verwenden.
- Die Notwendigkeit von Typumwandlungen zu erkennen und diese durchzuführen.
- Verschiedene Fehler in Software-Systemen zu kategorisieren.
- Eingabedaten zum Testen von Programmen sinnvoll auszuwählen.

1 Erste Schritte in Java

Als erstes Java Programm werden wir hier das “Hello World!” - Programm betrachten. Das Programm soll den Text “Hello World!” auf der Konsole ausgeben. Es wird klassischerweise als Beispiel verwendet, um eine Programmiersprache kurz vorzustellen.

```
1 /* Ausgabe von "Hello World!" auf der Kommandozeile */
2 public class Hello {
3   public static void main (String[] args){
4     System.out.println("Hello, World!");
5   }
6 }
```

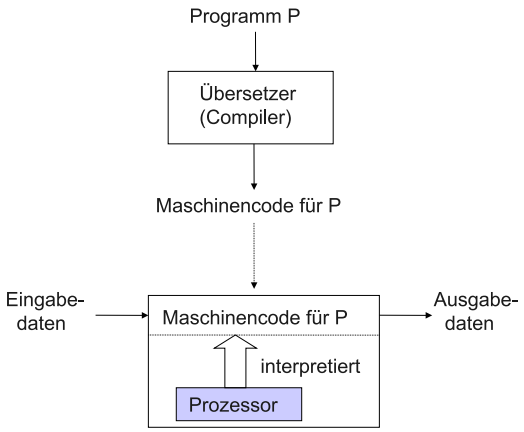
Dieses einfache Java-Programm besteht aus den folgenden Elementen:

- Die erste Zeile `/* ... */` ist ein Kommentar, der das Programm informell beschreibt.
- Das Programm besteht aus einer Klasse, die wir `Hello` genannt haben. Es muss daher in einer Textdatei namens `Hello.java` gespeichert werden.
- In dieser Klasse gibt es eine `main()` - Methode; hier startet die Programmausführung.
- Das Programm selbst besteht aus einer Sequenz von Anweisungen, die durch `;` voneinander getrennt werden. Diese Anweisungen bilden den ***Rumpf*** der Methode.
- Die Anweisung `System.out.println()` ruft eine Bibliotheksfunktion auf, die einen Text auf der Konsole ausgibt. Der auszugebende Text wird dabei zwischen die Klammern geschrieben.
- Befehlszeilenargumente können mittels `String[] args` übergeben werden (siehe Abschnitt 4.1).

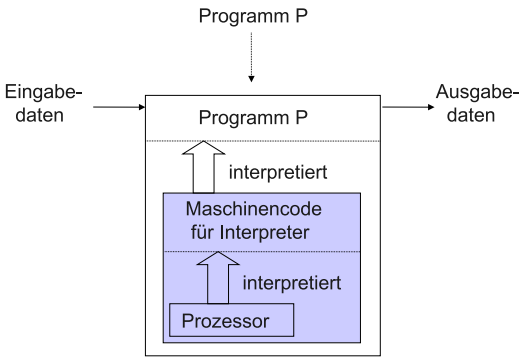
1.1 Prinzipielle Ausführungsvarianten

Programme können auf verschiedene Arten ausgeführt werden:

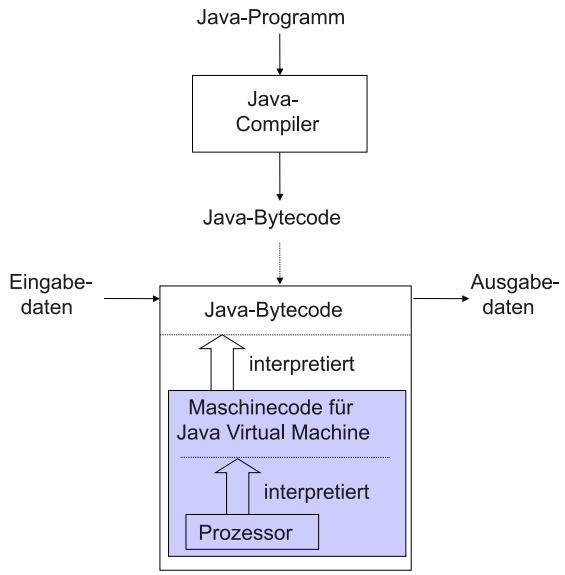
Mittels Übersetzung: Dabei wird der Programmtext in Maschinensprache übersetzt und dann direkt vom Prozessor ausgeführt.



Mittels direkter Interpretation: Hier wird der Programmtext von einem Interpreter ausgeführt, der wiederum – zusammen mit dem interpretierten Code – vom Prozessor ausgeführt wird.



Java verwendet eine Mischform aus Übersetzung und Interpretation:



Das Programmieren in Java umfasst grob die folgenden Schritte:

1. Programmtext schreiben
2. Programm übersetzen / compilieren
3. Programm ausführen

Editor ⟶ Hello.java ⟶ Compiler ⟶ Hello.class ⟶ JVM ⟶ “Hello, World!”

Zwei Schritte haben wir hier zunächst vernachlässigt:

- Integration/Komposition mehrerer Programmteile, die unabhängig von einander entwickelt und evtl. übersetzt wurden
- Installation der Programme auf der Host-Maschine

In der Praxis gewinnen diese Schritte durch die Vielzahl an bereits existierenden Teil-Programmen und Plattformen von Bedeutung.

2 Basisdatentypen

Literaturhinweis: Kapitel 1.2 aus R. Sedgewick, K. Wayne: Einführung in die Programmierung mit Java. 2011, Pearson Studium.

Programme verarbeiten Informationen in Form von Daten. Die Art der Daten wird durch den entsprechenden ***Datentyp*** festgelegt. Ein Datentyp hat einen Wertebereich und eine dazugehörige Menge an Operationen.
Beispiel:

- Messwerte werden oft als Zahlen repräsentiert; sie können z.B. addiert, multipliziert, verglichen werden.
- Texte und Zeichenfolgen werden als String behandelt; sie können z.B. zusammengefügt oder in Großbuchstaben umgewandelt werden.

In Java gibt es zwei Arten von ***Werten*** (engl. *values*):

- elementare Datenwerte (Zahlen, Wahrheitswerte, Zeichen, ...)
- Referenzen auf Objekte (⇒ näheres dazu im Kapitel zu "Objekte")

Wie für abstrakte Entitäten oder Begriffe typisch, besitzen Werte

- keinen Ort,
- keine Lebensdauer,
- keinen veränderbaren Zustand,
- kein Verhalten.

Ein ***Typ*** (engl. *type*) charakterisiert Werte, auf denen die gleichen Operationen zulässig sind. Typisierte Sprachen besitzen ein Typsystem, das für jeden Wert festlegt, von welchem Typ er ist.
Ausdrücke sind das Sprachmittel zur Beschreibung von Werten. Ein ***Ausdruck*** (engl. ***expression***) in Java ist (u.a.)

- ein Literal,
- ein Bezeichner,
- die Anwendung einer Operation auf einen oder mehrere Ausdrücke,
- oder aufgebaut aus Sprachmitteln, die erst später behandelt werden.

Jeder Ausdruck hat einen Typ:

- Der Typ eines Literals ergibt sich aus dem entsprechenden Wertebereich.

- Der Typ eines Bezeichners ergibt sich aus dem Wert, den er bezeichnet.
- Der Typ einer Operation ist der Ergebnistyp der Operation.

Wir unterscheiden verschiedene Arten von Operatoren:

- *Unäre* Operatoren haben einen Operanden, *binäre* Operatoren haben zwei Operanden, *ternäre* Operatoren drei, etc.
- Präfix-Operatoren stehen *vor* dem Operator, Infix-Operatoren stehen *zwischen* den Operanden, Postfix-Operatoren *hinter* den Operanden.

Wir betrachten zunächst die Basisdatentypen und elementare Operatoren für Werte dieser Datentypen. Die Basisdatentypen (engl. *basic / primitive data types*) bilden die Grundlage zur Definition weiterer Typen und Datenstrukturen. Man findet sie in nahezu jeder Programmier-, Spezifikations- und Modellierungssprache.

2.1 Ganze Zahlen / Integer

Dem Typbezeichner `int` ist in Java eine Wertemenge zugeordnet, die die ganzen Zahlen von -2147483648 (-2^{31}) bis 2147483647 ($2^{31}-1$) enthält.

Typbezeichner	`int`
Werte	ganzen Zahlen von -2^{31} bis $2^{31}-1$
Typische Literale	`1, 0, -99`
Operationen	`+` (Addition) `-` (Subtraktion) `*` (Multiplikation) `/` (Division) `%` (Modulo)

Beispiele:

Ausdruck	Wert	Hinweis
5 - 3	2	
5 / 3	1	ganzzahlige Division
5 % 3	2	Rest bei ganzzahliger Division
1 / 0		Laufzeitfehler: `java.lang.ArithmeticException: / by zero`
3 - 5 - 4	-6	Linksassoziativität der Operatoren
3 - (5 - 4)	2	
3 * 5 - 4	11	Multiplikation hat höhere Priorität als Addition

Innerhalb der Wertemenge $[-2^{31}, 2^{31}-1]$ sind die Operatoren auf beschränkten ganzen Zahlen gleich den üblichen mathematischen Operatoren. Außerhalb der Wertemenge ist ihr Verhalten nicht definiert. So wertet beispielsweise 2147483647 + 1 zu -2147483648 aus (***Überlauf***).
Weitere Datentypen für ganze Zahlen: `long` (64-bit Integer), `short` (16-bit Integer), `byte` (8-bit Integer).

2.2 Gleitkommazahlen

Dem Typbezeichner `double` ist in Java die (endliche!) Wertemenge der 64-bit Gleitkommazahlen zugeordnet.

Typbezeichner	`double`
Werte	reelle Zahlen (nach IEEE-Standard 754)
Typische Literale	`3.1415, 1.98e20, -1.0`
Operationen	`+` (Addition) `-` (Subtraktion) `*` (Multiplikation) `/` (Division)

Beispiele:

Ausdruck	Wert	Hinweis
3.14 + .03	3.17	
5.0 / 3.0	1.6666666666666667	Rundungsfehler möglich
1.0 / 0.0	`Infinity`	spezieller Wert

Mit Gleitkommazahlen sind im Allg. keine beliebig präzisen Berechnungen möglich. Sie bieten nur eine Näherung an die reelle Zahlen $\mathbb{R}$, haben aber nur endliche Darstellung mit eingeschränkter Präzision. So sind die Integer in vielen Programmiersprachen keine Teilmenge der Gleitkommazahlen! Nicht alle Integer können z.B. präzise als Gleitkommazahl dargestellt werden. Durch die eingeschränkte Präzision kann es leicht zu Rundungsfehlern kommen, die sich akkumulieren können und Ergebnisse unter Umständen sogar unbrauchbar machen. Das Fachgebiet der Numerik befasst sich mit dieser Problematik.
Die Standardisierung der Gleitkommazahlen nach IEEE-Standard 754 definiert neben den Zahlenwerten selbst zwei spezielle Werte:

- `Infinity` bzw. `-Infinity`
- `NaN` (not a number; wenn Berechnung undefiniert)

Weiterer Datentyp für Gleitkommazahlen: `float` (32-bit Gleitkommazahlen)

Frage 1: Zu welchen Werten ergeben sich folgende Ausdrücke?

- `1.0 / 0.0`
- `-1.0 / 0.0`
- `0.0 / 0.0`
- `1 / 0`

2.3 Boolesche Werte

Dem Typbezeichner `boolean` ist die Wertemenge der Wahrheitswerte {`true`, `false`} zugeordnet.

Typbezeichner	`boolean`
Werte	wahr, falsch
Literale	`true`, `false`
Operationen	`&&` (Und) `||` (Oder) `!` (Nicht)

Semantik der Operationen:

- `a && b` ist `true`, wenn beide Operanden den Wert `true` haben; sonst `false`.
- `a || b` ist `false`, wenn beide Operanden den Wert `false` haben; sonst `true`.
- `!a` ist `true`, wenn `a` `false` ist; und `false`, wenn `true` ist.

Frage 2: Zu welchen Werten ergeben sich folgende Ausdrücke?

```
true || false
true || true
(7 >= 45) == true
sein || !sein
```

für geeignete Variable `sein`

Vergleichsoperatoren

Die folgenden Operatoren vergleichen primitive numerische Werte *gleichen Typs* miteinander und liefern als Ergebnis einen boolschen Wert.

`==`	Gleichheit
`!=`	Ungleichheit
`>, >=`	größer bzw. größer gleich
`<, <=`	kleiner bzw. kleiner gleich

Wenn unterschiedliche Funktionen oder andere Programmelemente den gleichen Bezeichner haben, spricht man vom ***Überladen*** des Bezeichners (engl. *overloading*).
Wie in den obigen Datentypen gezeigt, können Operatorbezeichner in Java **überladen** werden, d.h. in Abhängigkeit vom Typ ihrer Argumente bezeichnen sie unterschiedliche Funktionen.
Beispiele:

- `+` arbeitet z.B. auf `int` und `double`
- `==` arbeitet z.B. auf `int` und `boolean`

2.4 Character

Der Datentyp `char` repräsentiert einzelne Zeichen aus dem Unicode-Zeichensatz, z.B. Ziffern, Groß- und Kleinbuchstaben, Satzzeichen, White space (Leerzeichen, Tabulatorzeichen, Zeilenumbruchzeichen), etc. Unicode ist ein Standard für die Repräsentierung, Kodierung und Darstellung von Schriftzeichen aller bekannter internationaler Schriftkulturen und Zeichensysteme.

Typbezeichner	`char`
Werte	Zeichen
Typische Literale	`'a', '7', 'B'`

`char`-Werte stellen Unicode-Zeichen als Zahlen zwischen 0 und 65535 (16-bit) dar. Beispiele: `'a' == 97` , `'b' == 98` , `'A' == 65` , `'B' == 66` , `'0' == 48` und `'1' == 49` . Man kann sie daher vergleichen und auch mit ihnen rechnen:

```
// Ausdruck, der testet, ob Character c ein Buchstabe?
(c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')

System.out.println('3' - '0');  // liefert 3
System.out.println('a' - 'z');  // liefert -25
System.out.println('a' < 'A');  // liefert false
```

Spezielle Zeichen:

'\"'	doppeltes Anführungszeichen
'\''	einfaches Anführungszeichen
'\n'	Zeilenumbruch
'\t'	Tabulatorzeichen
'\b'	Backspace
'\\'	Backslash-Zeichen

2.5 Strings

Dem Typbezeichner `String` ist in Java die Wertemenge der Zeichenketten zugeordnet.

Typbezeichner	`String`
Werte	Zeichenketten, Text
Literale	`"Hallo", "Die 7 Zwerge",`
Operation	`+` (Konkatenation, Verkettung)

Beispiele:

`"Hello, " + "World!"`	ergibt sich zu	`"Hello, World!"`
`"12 " + "34"`	ergibt sich zu	`"12 34"`
`"12" + "34"`	ergibt sich zu	`"1234"`
`"Die " + 7 + " Zwerge"`	ergibt sich zu	`"Die 7 Zwerge"`

Strings sind in Java keine primitiven Datentypen, jedoch in die Sprachdefinition fest integriert. Wenn primitive Datentypen mit einem String konkateniert werden, werden sie automatisch in Strings umgewandelt. Strings können auf der Kommandozeile direkt ausgegeben oder auch von dort eingelesen werden, wie wir im nächsten Abschnitt sehen werden.

Frage 3: Zu welchen Werten ergeben sich folgende Ausdrücke?

`"seven" + "three"` ergibt sich zu

`"4" + "5"` ergibt sich zu

`"" + 9 + 7` ergibt sich zu

`9 + 7 + ""` ergibt sich zu

`"7" + "+" + "5"` ergibt sich zu

2.6 Präzedenzregeln bei Operatoren

Wenn Ausdrücke nicht vollständig geklammert sind, ist im Allgemeinen nicht klar, wie ihr Syntaxbaum aussieht und wie die Auswertung zu erfolgen hat.

Frage 4: Zu welchen Werten ergeben sich folgende Ausdrücke?

```
false == true || true
false && true || true
3 == 5 == true
true == 5 == 3
```

Präzedenzregeln legen fest, wie Ausdrücke zu strukturieren sind:

- Am stärksten binden unäre Operatoren in Präfixform.
- Binäre Infix-Operatoren in Java sind (in der Regel) linksassoziativ.
- Präzedenzregeln für Infix-Operatoren:

  `*, / ,%` binden stärker als `+` und `-`

  `+,-` binden stärker als `<=, <, >=, >`

  `<=, <, >=, >` binden stärker als `==, !=`

  `==, !=` binden stärker als `&&`, und `&&` stärker als `||`

2.7 Striktheit von Operatoren

Ein Operator ist ***strikt***, wenn gilt: Ist einer der Operanden undefiniert, so ist das Ergebnis der Auswertung des Operators ebenfalls undefiniert.
Boolsche Operatoren `&&` und `||` sind ***nicht-strikt***; d.h. die Auswertung beginnt mit dem ersten (linken) Operanden, und nur wenn dieser Wert den Wert des Gesamtausdrucks nicht bestimmt, wird der rechte Operand ausgewertet.

Beispiel: Die Auswertung des Ausdrucks `(x != 0) && (y / x == 5)` testet zunächst, ob `x != 0`; nur wenn dieser Teilausdruck zu `true` auswertet, wird der zweite Teilausdruck `(y / x == 5)` ausgewertet. Falls `x != 0` zu `false` auswertet, ergibt sich der Gesamtausdruck zu `false`.
Im Gegensatz zu den boolschen Operatoren sind die arithmetischen Operatoren *strikt*; d.h. hier werden immer zunächst alle Operanden ausgewertet.

3 Variablen

Eine ***Speichervariable*** (oder einfach nur: Variable) ist ein Speicher/Behälter für Werte. Der Wert einer Variablen kann sich – im Unterschied zur mathematischen Verwendung – im Laufe eines Programms verändern. Charakteristische Operationen auf einer Variablen v sind:

- ***Zuweisen*** eines Werts w an v;
- ***Lesen*** des Wertes, den v enthält/speichert/hat.

Der Zustand einer Variablen v ist undefiniert, wenn ihr noch kein Wert zugewiesen wurde; andernfalls ist der Zustand von v durch den gespeicherten Wert charakterisiert. Variablen stellen wir graphisch meist durch Rechtecke dar:

v: 

v *enthält/speichert* den Wert `true`

x: 

x *enthält/speichert* den Wert 7

Variablen werden über ***Bezeichner*** (engl. *identifier*) referenziert. Für Bezeichner muss in Java gelten:

- Bezeichner bestehen aus einer Teilmenge der Unicode-Zeichen (insbesondere Buchstaben, Ziffern und Unterstrich _).[1]
- Sie starten nicht mit einer Ziffer.
- Bezeichner dürfen nicht mit Schlüsselwörtern übereinstimmen.

[1] Weitere Details unter `https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html#isJavaIdentifierPart-char-`.

- Bezeichner sind *case-sensitive*, d.h. es wird zwischen Groß- und Kleinschreibung unterschieden.

Eine Variablendeklaration hat in Java die Form:

```
Variablendeklaration →
    Typ ≪ Bezeichner ≫;
```

Sie definiert eine neue Variable vom angegebenen Typ mit dem angegebenem Bezeichner. Variablen müssen immer mit einem Typ deklariert werden und ihnen darf nur ein Wert mit "kompatiblem" Typ zugewiesen werden. Welche Typen miteinander "kompatibel" sind, werden wir im Laufe der Vorlesung sehen. Der Typ einer Variablen wird bei der Deklaration festgelegt und kann sich im weiteren Programmverlauf nicht ändern.
Beispiele:

```
int a;
boolean b;
double meineGleitkommavariable;
```

3.1 Zuweisungen

Syntax in Java:

```
Zuweisung →
    ≪ Bezeichner ≫ = Ausdruck;
```

Semantik: Werte den Ausdruck aus und weise das Ergebnis der Variablen zu.

In Java ist eine Zuweisung syntaktisch ein Ausdruck, liefert also einen Wert und zwar das Ergebnis der Auswertung der rechten Seite der Zuweisung.

Beispiele:

```
a = 27 % 23;
b = true;
meineGleitkommavariable = 3.14;
```

Sprechweise: "Variable `b` ergibt sich zu `true`" oder "Variable `b` wird `true` zugewiesen".

Variablen in Java müssen vor Verwendung deklariert und initialisiert werden. Andernfalls meldet der Compiler einen Übersetzungfehler.

4 Programmbeispiel: Schaltjahre

Betrachten wir zunächst ein Programm, das die bisher vorgestellten Konzept an einem praktischen Beispiel erläutert: Wir wollen Programm schreiben, das für eine eingegebene Jahreszahl bestimmt, ob es sich um ein Schaltjahr handelt. Wenn der erste Programmparameter ein Schaltjahr ist, soll `true` ausgegeben werden, andernfalls `false`. Wann ist ein Jahr ein Schaltjahr?

- Die durch 4 ganzzahlig teilbaren Jahre sind Schaltjahre. [...]
- Die durch 100 ganzzahlig teilbaren Jahre (z.B. 1700, 1800, 1900, 2100 und 2200) sind keine Schaltjahre. [...]
- Schließlich sind die ganzzahlig durch 400 teilbaren Jahre doch Schaltjahre. Damit sind 1600, 2000, 2400, ... jeweils wieder Schaltjahre. [...]

Eine mögliche Implementierung in Java ist diese hier (`Schaltjahr.java`):

```
1  public class Schaltjahr {
2    public static void main(String[] args) {
3      String eingabe;
4      int jahr;
5      boolean istSchaltjahr;
6
7      eingabe = args[0];
8      jahr = Integer.parseInt(eingabe);
9
10     istSchaltjahr = (jahr % 4 == 0) && (jahr % 100 != 0);
11     istSchaltjahr = istSchaltjahr || (jahr % 400 == 0);
12
13     System.out.println(istSchaltjahr);
14   }
15 }
```

4.1 Eingabe/Ausgabe von Daten

In unseren Beispielen verwenden wir zunächst die Eingabe auf Kommandozeilenebene bei Programmstart. Das erste Argument ist in der `main`-Methode unter `args[0]` verfügbar; das zweite Argument unter `args[1]` verfügbar, etc. Da die Argumente immer als `String`-Werte übergeben werden, müssen sie evtl. in einen anderen Datentyp umgewandelt werden (⇒ siehe Abschnitt 4.2).

Beispiel Compilieren und Ausführen von `Schaltjahr` mit Parameter:

```
> javac Schaltjahr.java
> java Schaltjahr 2004
true
> java Schaltjahr 1900
false
```

Fehlt der Parameter beim Aufruf, wird ein Fehler ausgegeben!

4.2 Typumwandlungen

Bisweilen muss man Werte eines Types in Werte eines anderen Typs umwandeln. So wurde im Programmbeispiel zur Ermittlung von Schaltjahren in Abschnitt 4 das Kommandozeilenargument `args[0]`, das vom Typ `String` ist, in einen Integer umgewandelt.

Zur Umwandlung von Strings in numerische Werte gibt es spezielle Bibliotheksmethoden:

`int Integer.parseInt(String s)`	Wandelt `s` in `int`-Wert um
`double Double.parseDouble(String s)`	Wandelt `s` in `double`-Wert um
`boolean Boolean.parseBoolean(String s)`	Wandelt `s` in `boolean`-Wert um

Frage 5: Zu welchem Wert ergeben sich die folgenden Ausdrücke?

- `Integer.parseInt("123")`
- `Integer.parseInt("007")`
- `Integer.parseInt("Hallo Welt")`
- `Double.parseDouble("4")`

Expliziter Typumwandlung (Cast) Java hat auch Typumwandlungen für primitive Datentypen vorgesehen. Dieser muss explizit im Programm angegeben werden, in dem der Typname in Klammern einem Ausdruck vorangestellt wird. Dabei muss man verstärkt auf die Klammerung achten, da Casts stärker binden als die arithmetischen und logischen Operatoren.
Achtung: Bei dieser Art von Typumwandlung kann es zu Informationsverlust kommen. *Beispiel:* `(int)2.71828` ergibt den `int`-Wert `2`, `(int)-2.9472` ergibt den `int`-Wert `-2`. Der Nachkomma-Teil wird dabei "abgeschnitten".

Alternativ kann man Runden mittels der Bibliotheksfunktion `long Math.round(double d)` und explizitem Cast von `long` auf `int`.
Beispiel: `(int) Math.round(2.71828)` liefert `3`.

Automatische Promotion von Zahlen Daten von primitiven numerischen Typen können in einem Kontext verwendet werden, wenn in diesem ein Wert eines Datentyps verwendet wird, der einen größeren Wertebereich abdeckt.
Beispiel: Bei der Auswertung von `4.0 * 3` wird `3` zunächst automatisch in einen `double`-Wert umgewandelt; danach wird die Multiplikation auf `double` durchgeführt.

Frage 6: Auch bei der automatischen Promotion von Zahlen entsteht unter Umständen ein Informationsverlust!
Welcher Wert wird hier ausgegeben?

```
int x = 16_777_218;
int y = 16_777_217;
float xx = x;
System.out.println(xx - y);
```

Um die Ausgabe von Werten primitiven Datentyps zu erleichtern, wandelt Java diese automatisch in Strings um, wenn diese Werte mit anderen Strings konkateniert werden. *Beispiel:* Bei der Auswertung von `"Ocean's " + 11` wird `11` zunächst automatisch in einen `String`-Wert `"11"` umgewandelt, danach wird die Konkatenation auf `String` durchgeführt.

5 Testen von Programmen

Softwaretests sind eine der wichtigsten Maßnahmen zur Qualitätssicherung in der Software-Entwicklung. Sie helfen uns Fehler in der Software zu erkennen.

> „Ein Test [...] ist der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen "(Denert, 1991)

5.1 Fehlerquellen in der Programmierung

Bei der Erstellung von Programmen können verschiedene Arten von Fehlern passieren.

Lexikalische und syntaktische Fehler sind Verstöße im Programmtext gegen die grammatischen Regeln der Programmiersprache. Hierzu zählen die Verwendung reservierter Schlüsselwörter als Bezeichner, fehlende Klammern. Weitere Fehler können auftreten, wenn Programme syntaktisch fehlerfrei sind, aber die Kontextbedingungen der Programmiersprache verletzt sind. Typische Fehler dieser Art sind Typfehler, falsche Anzahl an Parametern beim Methodenaufruf und Verwendung nicht deklarierter Variablen.

Logische Fehler entstehen durch einen falschen Problemlösungsansatz. Dies kann beispielsweise auf Grund einer falsch verstandenen Aufgabenstellung bzw. falsch interpretierten Spezifikation entstehen. Aber auch Unachtsamkeiten bei der Umsetzung einer Lösungsidee kann zu fehlerhaften Verhalten eines Programms, falschen Ergebnissen, Sicherheitslücken oder Programmabstürzen führen.

Laufzeitfehler sind alle Arten von Fehlern, die auftreten, während ein Programm ausgeführt wird. Während die bisher beschriebenen Fehler durch ein tatsächlich fehlerhaftes Programm induziert werden, das entweder nicht ausführbar ist oder fehlerhafte Ergebnisse liefert, kann auch ein eigentlich „korrektes" Programm bei seiner Ausführung zu Fehlern führen, z.B. durch falsche Eingabedaten.

Designfehler sind Fehler, die bereits bei der Definition der Anforderungen an die Software oder auch bei der Entwicklung des Design entstehen. Diese haben ihren Ursprung oft in mangelnder Kenntnis des Fachgebiets oder entstehen auf Grund von Missverständnissen zwischen Auftraggeber und Entwicklern.

5.2 Fehlererkennung und -vermeidung

Lexikalische und syntaktische Fehler sowie Verletzungen von Kontextbedingungen werden durch den Compiler erkannt. Diese Fehler verhindern in der Regel eine Übersetzung des fehlerhaften Programms und können früh im Entwicklungsprozess erkannt werden. Um logische Fehler zu erkennen, testet man Software-Systeme. Diese Tests können entweder vollständige Programme oder auch einzelne Komponenten testen. Um das Software-Systeme systematisch zu testen, gibt es verschiedene Frameworks, die es erlauben, Tests zu verfassen oder auch automatisch zu generieren. In einem späteren Kapitel werden wir JUnit kennenlernen, ein Testframework für Java.
Um die Programme diese Kapitels zu testen, führen wir sie mit verschiedenen Eingabedaten aus und vergleichen die tatsächliche Ausgabe des Programms mit der erwarteten Ausgabe.

Da wir tatsächliche Ausgabe mit erwarteter Ausgabe zeichenweise vergleichen wollen, ist es wichtig, dass Programme **genau** ihrer Spezifikation folgen.
Wenn Beispielsweise die Spezifikation des Schaltjahr-Programms fordert, dass für Schaltjahre "`true`" ausgeben soll, dann wäre es ein Fehler statt dessen "Das Jahr ist ein Schaltjahr." auszugeben. Ein automatischer Test, der Ausgaben vergleicht, wird dieses Problem schnell aufdecken.

5.3 Testmethodik

Unser Ziel ist es eine möglichst hohe Abdeckung des Codes durch Testfälle zu erreichen, um möglichst viele Fehler auszuschließen. Jedes Programm sollte daher mit verschiedenen Eingaben getestet werden. In der Regel ist es nicht möglich, alle möglichen Eingabedaten zu testen. Die Testeingaben sollten allerdings so gewählt sein, dass möglichst jeder der verschiedenen Fälle bei der Ausführung der Tests durchlaufen wird.

Frage 7: Warum sind die Eingabedaten 1992, 2016, 2040 **allein** keine geeigneten Testdaten, um das Verhalten des `Schaltjahr`-Programms zu testen?

Randfälle sind dabei besonders wichtig, da sie oft zu Implementierungsfehlern führen. Typischerweise wählt man dabei folgende Eingaben:

- Bei Integer-Werten: 0, 1, -1, ...
- Bei Arrays: Leere Arrays, einelementige Arrays, ...
- Bei Strings: Leerer String, Strings der Länge 1, ...

In der Praxis können nicht beachtete Randfälle zu kuriosen Problemen führen. 1978: Ein Vorzeichenfehler in der Software des Kampfflugzeug F-16 sorgte dafür, dass der Autopilot das Flugzeug in Rückenlage brachte, wenn es den Äquator überflog. Der

Fehler entstand, da man keine negativen Breitengrade als Eingabedaten bedacht hatte. Diese kritische Situation wurde erst spät in der Entwicklungsphase des F-16 während einer Simulation entdeckt.
Wenn sich die Randfälle aus der Problembeschreibung abgeleitet werden, da Implementierungsdetails nicht bekannt sind, spricht man von *Black-Box Tests*. Bei *White-Box Tests* ist die Implementierung bekannt, und man kann aus ihr Randfälle ableiten.
Man beachte aber:

> „Program testing can be used to show the presence of bugs, but never show their absence! “(Edsger W. Dijkstra)

Hinweise zu den Fragen

Hinweise zu Frage 1: Das folgende Programm berechnet die Werte der Ausdrücke und gibt sie auf die Kommandozeile aus.

```
1 public class ArithOperationen {
2   public static void main(String[] args){
3     System.out.println(" 1.0 / 0.0  liefert " + 1.0/0.0);
4     System.out.println("-1.0 / 0.0  liefert " + (-1.0/0.0));
5     System.out.println(" 0.0 / 0.0  liefert " + 0.0/0.0);
6     System.out.println(" 1   / 0    liefert " + 1/0);
7   }
8 }
```

Hinweise zu Frage 2: Die Ausdrücke `true || false`, `true || true` ergeben sich zu `true`.
Der Ausdruck `(7 >= 45) == true` ergibt sich zu `false`, da `7 >= 45` sich zu `false` ergibt und `false == true` sich zu `false` ergibt.
Für den Ausdruck `sein || !sein` betrachten wir die zwei möglichen Fälle:

- Falls die Variable `sein` den Wert `true` hat, ergibt sich der Ausdruck zu `true || !true`, was sich zu `true` ergibt.
- Falls die Variable `sein` den Wert `false` hat, ergibt sich der Ausdruck zu `false || !false`, was sich ebenfalls zu `true` ergibt.

Daher ergibt sich der Ausdruck `sein || !sein` in jedem Fall zu `true`.

Hinweise zu Frage 3:

`"seven" + "three"`	ergibt sich zu	`"seventhree"`
`"4" + "5"`	ergibt sich zu	`"45"`
`"" + 9 + 7`	ergibt sich zu	`"97"`
`9 + 7 + ""`	ergibt sich zu	`"16"`
`"7" + "+" + "5"`	ergibt sich zu	`"7+5"`

Hinweise zu Frage 4: `false == true || true` und `false && true || true` ergeben sich zu `true`. Der Ausdruck `3 == 5 == true` hat den Wert `false`. Der Ausdruck `true == 5 == 3` kann nicht ausgewertet werden, da der Vergleich `true == 5` nicht typkorrekt ist.

Hinweise zu Frage 5: `Integer.parseInt("123")` liefert 123. `Integer.parseInt("007")` liefert 7. `Double.parseDouble("4")` liefert 4.0. Beim Auswerten des letzen Aufrufs gibt es einen Laufzeitfehler: `java.lang.NumberFormatException: For input string: "Hallo Welt!"`

Hinweise zu Frage 6: Der ausgegebene Wert ist 2.0.

Hinweise zu Frage 7: Die Jahre 1992, 2016, 2040 sind alles Schaltjahre. Daher wird durch diese Eingaben nur ein Teil der Spezifikation des Programms abgedeckt.
Besser wären: 1992 (durch 4 teilbar, aber nicht durch 100), 1990 (nicht durch 4 teilbar), 1900 (durch 100 teilbar, aber nicht durch 400), 2000 (durch 4, 100 und 400 teilbar).