Kapitel 1

Objektorientierte Programmierung

1.1 Einleitung

Die Basis der objektorientierte Programmierung liegt bei drei Grundelemente: Unterstützung vomn Vererbungsmechanismen, Unterstützung von Datenkapselung und Unterstützung von Polymorphie.

Die Unified Modelling Language (UML) ist ein Modellierungsmittel. Die objektorientierte Analyse betrachtet eine Domäne als System von kooperierenden Objekten. Entwurfsmuster sind Elemente wiederverwendbarer objektorientierter Software und dienen als objektorientierte Methoden.

Jede Sprache hat ihre eigene Stärken, jede macht bestimmte Sachen einfach, und jede hat ihre eigenen Idiome und Muster. Je mehr Programmiersprachen man kennt, desto mehr Vorgehensweisen lernt man oft.

Korrektheit: Software soll genau das tun, was von ihr erwartet wird. Benutzerfreundlichkeit: Software soll einfach und intuitiv zu benutzen sein. Effizient: Software soll mit wenigen Ressourcen auskommen und gute Antwortzeiten für Anwender haben. Wartbarkeit: Software soll mit wenig Aufwand erweiterbar und änderbar sein.

1.2 Basis oder Objektorientierung

Die **Datenkapselung**, die **Polymorphie** und die **Vererbung** sind drei Werkzeugen, die einem erlaubt, die Zielsetzungen der Entwicklung von Software anzugehen. Der Speicher enthält Daten, die bearbeitet werden, andererseits enthält er Instruktionen, die bestimmen, was das Programm macht.

Routinen sind das Basiskonstrukt der strukturierten Programmierung. Indem ein Programm in Unterprogramme zerlegt wird, erhält es seine grundsätzliche Struktur. Eine Routine kann entweder Parameter haben oder ein Ergebnis zu-

rückgeben. Eine Routine wird auch als Unterprogramm bezeichnet.

Eine **Funktion** ist eine Routine, die einen speziellen Wert zurückliefert, ihren Rückgabewert.

Eine **Prozedur** ist eine Routine, die keinen Rückgabewert hat. Eine Übergabe von Ergebnissen an einen Aufrufer kann trotzdem über die Werte der Parameter erfolgen.

Die **Strukturierung** der Instruktionen und der Daten erfolgen durch Verzweigungen, Zyklen und Routinen. Man benutzt globale und lokale, statisch und dynamisch allozierte Variablen, deren Inhalte definierte Strukturen wie Datentypen, Zeiger, Records, Arrays, Listen, Bäume oder Mengen haben. Das Strukturieren ist die Beherrschung der Komplexität.

Daten und Routinen sind voneinander getrennt. Die Objektorientierung verändert diese zwei durch die Einführung von Objekten. Daten gehören nun explizit einem Objekt. Objekte können ihre Daten verändern oder auch lesend auf sie zuzugreifen. Ein Objekt kann selbst dafür sorgen, dass die Konsistenz der Daten gewahrt bleibt.

Das Prinzip der **Datenkapselung** stellt die Konsistenz von Daten sicher und somit ist die Korrektheit gewährleistet. Vorgehensweisen und interne Daten können reduziert werden, ohne das Resultat zu beeinflussen.

Die **Polymorphie** ist die Vielgestaltigkeit. Ein Bezeichner nimmt abhängig von seiner Verwendung Objekte unterschiedlichen Datentyps an. Die Nutzung der Polymorphie führt zu wesentlich flexibleren Programmen. Sie steigert damit die Wartbarkeit und Änderbarkeit.

Objekte erhalten durch **Vererbung** die Attribute und Methoden anderer Objekte.

1.3 Prinzipien des objektorientierten Entwurfs

Unter einem **Modul** (Objjekte, Klassen, Datentypen) versteht man einen überschaubaren und eigenständigen Teil einer Anwendung. Solch ein Modul hat nun innerhalb einer Anwendung eine ganz bestimmte Aufgabe, für die es die Verantwortung trägt. Ein Modul hat eine oder mehrere Verantwortungen. Module bestehen aus weitere abhängigen Untermodulen.

Das Prinzip der einzigen Verantwortung besagt, dass jedes Modul genau eine Verantwortung übernimmt und ede Verantwortung genau einem Modul zugeordnet werden soll. Die Verantwortung setzt bestimmte zeitliche ändernbare Anforderungen des Moduls um.

Ein Modul soll zusammenhängend (kohäsiv) sein. Alle Teile eines Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein. Haben Teile eines Moduls keinen Bezug zu anderen Teilen, kann man

davon ausgehen, dass man diese Teile als eigenständige Module implementieren kann. Eine Zerlegung in Teilmodule bietet sich damit an.

Wenn für die Umsetzung viele Module zusammenarbeiten müssen, bestehen Abhängigkeiten zwischen diesen Modulen. Die Module sind gekoppelt. Die Kopplung zwischen Module sollte möglichst gering sein. Das kann man erreichen, indem man die Verwntwortung für die Koordination der Abhängigkeiten einem neuen Modul zuweist. Hierbei sollte man aber darauf achten, dass man bestehende Abhängigkeiten durch die Einführung eines vermittelten Moduls nicht verschleiert.

Eine Aufgabe, die ein Programm umsetzen muss, betrifft häufig mehrere voneinander zusammenhängende und abgschlossene Anliegen, die getrennt betrachtet und als getrennte Anforderungen formuliert werden können. Eine identifizierbare Funktionalität eines Systems sollte innerhalb dieses Systems nur einmal umgesetzt sein, Wiederholungen vermeiden.

Module sollten angepasst werden, offen für Erweiterung und geschlossen für Änderung. Das Modul kann so strukturiert werden, dass die Funktionalität, die fpr eine Variante spezifisch ist, sich durch eine andere Funktionalität leicht ersetzen lässt. Die Funktionalität der Standardvariante muss dabei nicht unbedingt in ein separates Modul ausgelagert werden. Das Modul soll definierte Erweiterungspunkte bieten, an die sich die Erweiterungsmodule anknüpfen lassen.

Erweiterungspunkte kann man in der Regel durch das Hinzufügen einer **Indirektion** erstellen. Das Modul darf die variantenspezifische Funktionalität nicht direkt aufrufen. Das Modul konsultiert eine Stelle, die bestimmt, ob die Standardimplementierung oder ein Erweiterungsmodul aufgerufen werden soll.

Jede Abhängigkeit zwischen zwei Module sollte explizit formuliert und dokumentiert sein. Ein Modul sollte immer von einer klar definierten Schnittstelle des anderen Moduls abhängig sein und nicht von der Art der Implementierung der Schnittstelle. Die Schnittstelle eines Modulssollte getrennt von der Implementierung betrachtet werden können.

Eine Abstraktion beschreibt das in einem gewählten Kontext Wesentliche eines Gegenstands oder eines Begriffs. Durch eine Abstraktion werden die Details ausgeblendet, die für eine bestimmte Betrachtungsweise nicht relevant sind. Abstraktionen ermöglichen es, unterschiedliche Elemente zusammenzufassenm, die unter einem bestimmten Gesichtspiunkt gleich sind.

1.4 Die Struktur der Obektorientierung

Kapitel 2

Java

2.1 Elemente der Programmiersprache Java

2.1.1 Bytecode

Der Java-Compiler erzeugt aus den Quellcode-Dateien den so genannten **Bytecode**. Dieser Code ist binär und Ausgangspunkt für die virtuelle Machine zur Ausführung. Der Bytecode ist wie ein Prozessor, der Anweiungen wie arithmetische Operationen, Sprünge und Weiteres kennt.

2.1.2 Java Virtual Machine

Die Java Virtual Machine (JVM) kümmert sich um den Bytecode, den Quellcode auszuführen. Die Laufzeitumgebung lädt den Bytecode, prüft ihn und führt ihn in einer kontrollierten Umgebung aus. Java ist Plattform- und Betriebssystemunabhängig. Zu der JVM und der Programmiersprache kommen Standardbibliotheken für Datenstrukturen, Zeichenkettenverarbeitung, Datumverarbeitung, grafische Oberflächen, Ein- und Ausgabe, Netzwerkoperationen und mehr.

2.1.3 Objektorientierung

Eine Laufzeitumgebung eliminiert viele Fehler. Objektorientierte Programmierung versucht, die Komplexität des Softwareproblems besser zu modellieren. Menschen denken objektorientiert, darum Java bildet diese ab. Objekte bestehen aus **Eigenschaften**, also Dinge, die ein Objekt “hat” und “kann”. Objekte entstehen aus **Klassen**, das sind Beschreibungen für den Aufbau von Objekten.

Primitive Datentypen für numerische Zahlen oder Unicode-Zeichen werden nicht als Objekte betrachtet. Das **Java-Security-Modell** sicherstellt den Programmablauf. Der **Verifier** liest Code und überprüft die Korrektheit und Typsicherheit. Treten Sicherheitsprobleme auf, werden sie durch Exceptions zur Laufzeit gemeldet. Das Security-Manager überwächt Zugriffe auf das Dateisystem, die Netzwerk-Ports, externe Prozesse und weitere Systemressourcen.

In Java gibt es keine Zeiger auf Speicherbereiche, dagegen führt Java **Referenzen** ein. Eine Referenz repräsentiert ein Objekt, und eine Variable speichert

diese Referenz, sie wird Referenzvariable genannt. JVM verbindet die Referenz mit einem Speicherbereich und einem Referenztyp; der Zugriff, Dereferenzierung genannt, ist indirekt. Referenz und Speicherblock sind getrennt.

In Java gibt es keine benutzerdefinierten überladenen Operatoren. Da das Operatorzeichen auf unterschiedlichen Datentypen gültig ist, nennt sich so ein Operator **Überladen**. Bei Zeichenketten werden Pluszeichen als **Konkatenation** angewendet. Java braucht keine **Präprozessoren**.

2.1.4 Java Platform

Mit dem Java Development Kit (JDK) lassen sich Java SE-Applikationen entwickeln. Dem JDK sind Hilfsprogramme beigelegt, die für die Java-Entwicklung nötig sind. Dazu zählen der essenzielle Compiler, aber auch andere Hilfsprogramme, etwa zur Signierung von Java-Archiven oder zum Start einer Management-Konsole.

Das Java SE Runtime (JRE) enthält genau das, was zur Ausführung von Java-Programmen nötig ist. Die Distribution umfasst nur die JVM und Java-Bibliotheken, aber weder den Quellcode der Java-Bibliotheken noch Tools wie Management-Tools.

2.1.5 Das erste Programm compilieren und testen

```
1
2  public class Squares
3  {
4
5      static int quadrat(int n)
6      {
7          return n*n;
8      }
9
10     static void ausgabe(int n)
11     {
12         for(int i=1; i <=n; i=i+1)
13         {
14             String s = "Quadrat(" + i + ") = " + quadrat(i);
15             System.out.println(s);
16         }
17     }
18
19     public static void main(String[] args)
20     {
21         ausgabe(6);
22     }
23
24 }
```

Ein Compiler übersetzt bzw. transformiert das geschriebene Programm in eine andere Repräsentation nämlich den Bytecode und erzeugt aus dem Program mit Endung `.java` die Datei `.class`, welche Bytecode enthält.

Wenn der Compiler aufgrund eines syntaktischen Fehlers eine Übersetzung in Java-Bytecode nicht durchführen kann, spricht man von einem Compilerfehler.

Eine Laufzeitumgebung liest die Bytecode-Datei Anweisung für Anweisung aus und führt sie auf den konkreten Mikroprozessor aus. Der Interpreter bringt das Programm zur Ausführung.

Ein Java-Projekt braucht eine ordentliche Ordnerstruktur, und hier gibt es zur Organisation der Dateien unterschiedliche Ansätze. Die einfachste Form ist, Quellen, Klassendateien und Ressourcen in ein Verzeichnis zu setzen. Es gibt zwei Verzeichnisse `src` für die Quellen und `bin` für die erzeugten Klassendateien. Ein eigener Ordner `lib` ist sinnvoll für Java-Bibliotheken.

Das Programm sitzt in einer Klasse, die drei Methoden enthält. Die Methode `quadrat(int)`, bekommt als Übergangsparameter eine ganze Zahl und berechnet daraus die Quadratzahl, die sie anschliessend zurückgibt. Eine weitere Methode übernimmt die Ausgabe der Quadratzahlen bis zu einer vorgegebenen Grenze. Die Methode `main()`, als Anfangspunkt, ruft die Methode `ausgabe(int)` auf.

2.2 Imperative Sprachkonzepte

2.2.1 Elemente der Programmiersprache Java

Unter dem Begriff **Semantik** versteht man die Lexikalik, Syntax und Semantik eines Programms. Der Compiler verläuft diese Schritte bevor er den Bytecode erzeugt.

Ein **Token** ist eine lexikalische Einheit, die dem Compiler die Bausteine des Programms liefert. Der Compiler erkennt an der Grammatik einer Sprache, welche Folgen von Zeichen ein Token bilden.

Whitespaces sind Leerzeichen, Tabulatoren, Zeilenvorschub und Seitenvorschubzeichen.

Neben den Trennern gibt es noch zwölf ASCII-Zeichen geformte Tokens, die als **Separator** definiert werden: `( ) { } [ ] ; , . ... @ ::`

Für Variablen, Methoden, Klassen und Schnittstellen werden **Bezeichner**, auch **Identifizierer** genannt, vergeben. Unter **Variablen** sind dann Daten verfügbar. **Methoden** sind die Unterprogramme in objektorientierten Programmiersprachen, und **Klassen** sind die Bausteine objektorientierter Programme. Ein Bezeichner ist eine Folge von Zeichen, die fast beliebig sein kann. Die Zeichen sind Elemente aus dem Unicode-Zeichensatz. Der Bezeichner muss mit einem Java-Buchstaben beginnen. String ist eine Klasse und kein Datentyp.

Ein Java-Buchstabe umfasst unsere lateinische Buchstaben “A” bis “Z”, “a” bis “z”, sondern auch viele Zeichen aus dem Unicode-Alphabet, den Unterstrich, Währungszeichen, griechische oder arabische Buchstaben, Akzente. Java unterscheidet zwischen Gross- und Kleinschreibung. Nicht erlaubt sind Zahlen am Anfang, Leerzeichen, Ausrufezeichen, reservierte Wörter oder reservierte Schlüsselwörter.

Ein **Literal** ist ein konstanter Ausdruck wie die Wahrheitswerte `true` und `false`, Integrale Literale für Zahlen, Fliesskommaliterale, Zeichenliterale wie \n, String-Literale für Zeichenketten wie “Hello World”, Referenztypen wie `null`.

Bestimmte Wörter sind reservierte Schlüsselwörter com Compiler besonders behandelt. Schlüsselwörter bestimmen die Sprache eines Compilers. Es können keine eigenen Schlüsselwörter hinzugefügt werden. Schlüsselwörter sind:

`abstract`, `assert`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extends`, `final`, `finally`, `float`, `for`, `goto`, `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`, `void`, `volatile`, `while`

Der Compiler überliest alle Kommentare und die Trennzeichen bringen den Compiler von Token zu Token. **Zeilenkommentare** kann man mit Schrägsstrichen `//` und kommentieren den Rest einer Zeile bis Zeilenumbruchzeichen aus. **Blockkommentare** (“Wie”) kommentiert in Blöcke mit `/* */` aus. **Javadoc-Kommentare** (“Was”) sind besondere Blockkommentare mit `/** */` und beschreibt die Methode oder die Parameter, aus denen sich später die API generieren lässt. Kein Kommentar kommt in den Bytecode.

2.2.2 Anweisungen

Programme sind Ablauffolgen, die im Kern aus **Anweisungen** bestehen. Sie werden zu grösseren Bausteinen zusammengesetzt, den Methoden, die wiederum Klassen bilden. Klassen selbst werden in Paketen gesammelt, und eine Sammlung von Paketen wird als Java-Archiv ausgeliefert.

Durch Anweisungen werden **Algorithmen** geschrieben. Anweisungen können Ausdrucksanweisungen für Zuweisungen oder Methodenaufrufe, auch Fallunterscheidungen, oder Schleifen für Wiederholungen sein.

Anweisungen müssen in einen Rahmen gepackt werden. Dieser Rahmen heisst **Kompilationseinheit** und deklariert eine Klasse mit ihren Methoden und Variablen. Anweisungen ausserhalb von Klassen sind nicht erlaubt. Der Klassenname ist ein Bezeichner und beinhaltet die gleiche Dateiname. Klassennamen beginnen mit Grossbuchstabe und Methoden sind kleingeschrieben. Zwischen den geschweiften Klammern folgen Deklarationen von Methoden und zwischen den Methoden die Anweisungen.

Eine besondere Methode ist `public static void(String[] args){}`. Die Methode ist für die Laufzeitumgebung etwas Besonders, denn beim Aufruf des Java-Interpreters mit einem Klassennamen wird diese Methode als Erstes ausgeführt. Demnach werden die Anweisungen ausgeführt, die innerhalb der geschweiften Klammern stehen. Der Parameter `args` wird immer verwendet.

Haltet man sich nicht an die Syntax für den Startpunkt, so kann der Interpreter die Ausführung nicht beginnen und man hätte einen semantischen Fehler produziert, obwohl die Methode korrekt gebildet ist.

Die Methode `println(...)` gibt Meldungen auf der Konsole aus. Innerhalb der Klammern können Argumente angegeben werden wie Zeichenketten oder **Strings** oder eine Folge von Buchstaben, Ziffern oder Sonderzeichen in doppelten Anführungszeichen. Die Methode `println(...)` gehört zum Typ `out` und diese zu `System`.

```
public class PrimeraClase
{
        public static void main(String args[])
        {
                System.out.println("Hola Mauri");
        }

}
```

Java erlaubt Methoden, die gleich heissen, denen aber unterschiedliche Dinge übergeben werden können; diese Methoden nennt man **überladen**. Viele `println()`-Methoden akzeptieren zahlartige Argumente und sind überladen.

```
public class OverloadedPrintln
{
    public static void main( String[] args )
    {
        System.out.println( "Hallo" );
        System.err.println( "Fehler" );
        System.out.println( true );
        System.out.println( -273 );
        System.out.println(); // Gibt eine Leerzeile aus
        System.out.println( 1.6180339887498948 );
    }
}
```

Die Methode `printf()` ermöglicht variable Argumentenlisten gemäss einer Formatierungsanweisung. Die Formatierungsanweisung `\n` setzt einen Zeilenumbruch, `\d` ist ein Platzhalter für eine ganze Zahl, `\f` ist ein Platzhalter für eine Fliesskommazahl, `\s` ist eine Zeichenkette oder etwas, was in einen String konvertiert werden soll.

```
public class VarArgs
{
    public static void main( String[] args )
    {
        System.out.printf( "Was sagst du?%n" );
        System.out.printf( "%d Kanaele und ueberall nur %s.%n",
            220, "Katzen" );
    }
}
```

Methodenaufrufe lassen sich als Anweisungen einsetzen, wenn sie mit einem Semikolon abegschlossen sind, man spricht von einer **Ausdrucksanweisung** (expression statement). Jeder Methodenaufruf mit Semikolon bildet eine Ausdrucksanweisung. Dabei ist es egal, ob die Methode selbst eine Rückgabe liefert oder nicht.

Die Methode `Math.random()` liefert eine Fliesskommazahl zwischen 0 (inklusiv) und 1 (exklusiv). In einer objektorientierte Programmiersprache sind alle Methoden an bestimmte Objekte mit einem Zustand gebunden. Alle Operationen und Zustände sind an Objekte bzw. Klassen gebunden. Der Aufruf einer Methode auf einem Objekt richtet die Anfrage genau an dieses bestimmte Objekt.

Die Deklaration einer Klasse oder Methode kann einen oder mehrere **Modifizierer** enthalten, die zum Beispiel die Nutzung einschränken oder parallelen Zugriff synchronisieren. Der Modifizierer `public` ist ein Sichtbarkeitsmodifizierer. Er bestimmt, onb die Klasse bzw. die Methode für Programmcode anderer Klassen sichtbar ist oder nicht. Der Modifizierer `static` zwingt den Programmierer nicht dazu, vor dem Methodenaufruf ein Objekt der Klasse zu bilden. Dieser Modifizierer bestimmt die Eigenschaft, ob sich eine Methode nur über ein konkretes Objekt aufrufen lässt oder eine Eigenschaft der Klasse ist, sodass für den Aufruf kein Objekt der Klasse nötig wird.

Ein **Block** fasst eine Gruppe von Anweisungen,die hintereinander ausgeführt werden. Ein Block `{}` ist eine Anweisung, die in geschweiften Klammern eine Folge von Anweisungen zu einer neuen Anweisung zusammenfasst. Ein Block kann überall dort verwendet werden, wo auch eine einzelne Anweisung stehen kann. Der neue Block hat jedoch eine Besonderheit in Bezug auf Variablen, da er einen lokalen Bereich für die darin befindlichen Anweisungen inklusive der Variablen bildet.

Ein Block ohne Anweisung nennt sich ein leerer Block. Er verhält sich wie eine leere Anweisung, also wie ein Semikolon. Es gibt innere und äussere Blöcke. Blöcke fassen Anweisungen zusammen.

2.3 Datentypen, Variablen und Zuweisungen

Java speichert Variablen. Eine Variable ist ein reservierter Speicherbereich und belegt eine feste Anzahl von Bytes. Variablen und Ausdrücke haben einen **Datentyp** und einen **Datenwert**. Der Datentyp bestimmt die zulässigen Operationen. Java ist eine streng typisierte Programmiersprache. Datentypen werden unterteilt in **primitive Datentypen** (Zahlen, Unicode-Zeichen und Wahrheitswerte) und **Referenztypen** (Zeichenketten, Datenstrukturen, Zwergpinscher) und Bytecode durch den Compiler einfacher erzeugt.

Typ	Grösse	Belegung (Wertebereich)
boolean	1 Bit	`true` oder `false`
char	16Bit	0x0000 ... 0xFFFF
byte*	8 Bit	-2^7 bis $2^7 - 1$
short*	16 Bit	-2^{15} bis $2^{15} - 1$
int*	32 Bit	-2^{31} bis $2^{31} - 1$
long*	64 Bit	-2^{63} bis $2^{63} - 1$
float	32 Bit	$1,4023 \cdot 10^{-45} \ldots 3,4028 \cdot 10^{38}$
double	64 Bit	$4,9406 \cdot 10^{-324} \ldots 1,7976 \cdot 10^{308}$

Tab. 2.1: Java-Datentypen, Grössen und Formate.
**Zweierkomplement*

Es gibt mehr negative Werte als positive Werte, das liegt an der Kodierung im Zweierkomplement. Bei `float` und `double` ist das Vorzeichen nicht angegeben, die Wertebereiche unterscheiden sich nicht, die kleinsten und grössten darstellbaren Zahlen können sowohl positiv als auch negativ sein.

2.3.1 Variablendeklarationen

Mit Variablen lassen sich Daten speichern, die vom Programm gelesen und geschrieben werden können. Variablen müssen deklariert werden. Hinter dem Typnamen folgt der Name der Variablen. Die **Deklaration** ist eine Anweisung und wird daher mit einem Semikolon abgeschlossen.

```
1  public class FirstVariable
2  {
3      public static void main(String[] args) {
4          String name = "Mickey Mouse";
5          int age = 100;
6          double income = 400000, weight = 12.5;
7          char gender = 'm';
8          boolean isPresident = false;
9      }
10 }
```

Gleich bei der Deklaration lassen sich Variablen mit einem Anfangswert initialisieren. Hinter einem Gleichheitszeichen steht der Wert, der oft ein Literal ist.

Eine Konsoleneingabe. Eine Variante ist die Klasse `java.util.Scanner`. Folgende Tabelle zeigt die Eingabe von drei verschiedenen Datentypen.

Eingabe	Anweisung
`String`	`String s = new java.util.Scanner(System.in).nextLine();`
`int`	`int i = new java.util.Scanner(System.in).nextInt();`
`double`	`double d = new java.util.Scanner(System.in).nextDouble();`

Tab. 2.2: Einlesen einer Zeichenkette, Ganz- und Fliesskommazahl von der Konsole

Folgendes Beispiel zeigt eine Anwendung aller Eingabenmöglichkeiten mit der Klasse `java.util.Scanner`.

```java
public class SmallConversation
{
    public static void main(String[] args)
    {
        System.out.println("Hallo, wie heissen Sie?");
        String name = new java.util.Scanner(System.in).nextLine
            ();
        System.out.printf("Hallo %s. Wie alt sind Sie?%n", name)
            ;
        int age;
        age = new java.util.Scanner(System.in).nextInt();
        System.out.printf("Aha, %s Jahre, das ist ja die Haelfte
             von %s.%n", age, age*2);
        System.out.printf("Sag mal, was ist deine
            Lieblingsfliesskommazahl?");
        double value = new java.util.Scanner(System.in).
            nextDouble();
        System.out.printf("%s? Aha, meine ist %s.%n", value,
            Math.random()*100000);
    }
}
```

2.3.2 Fliesskommazahlen

Java bietet die Datentypen `float` und `double`. Fliesskommazahl können einen Vorkommateil und einen Nachkommateil besitzen, die durch einen Dezimalzahl getrennt sind. Standardmässig sind die Fliesskommaliterale vom Typ `double`. Ein nachgestelltes `f` oder `F` zeigt dem Computer an, dass es sich um einen `float` handelt.

So ist beispielsweise `1+2+4.0` eine Addition aus `1+2` dann in `double` transformiert und anschliessend `3.0+4.0`. Die Standardbibliothek `java.math` bietet die Klasse `BigDecimal` an. Diese Klasse eignet sich gut für gute Genauigkeit wie Währungen.

2.3.3 Ganzzahlige Datentypen

Java stellt fünf ganzzahlige Datentypen zur Verfügung: `byte`, `short`, `char`, `int` und `long`. Ganzzahlige Datentypen sind immer vorzeichenbehaftet (mit Ausnahme von `char`). Einen Modifizierer `unsigned` gibt es nicht. Java reserviert nicht so viele Bits wie benötigt und wählt nicht automatisch den passenden Wertebereich. Dabei ist `System.out.println( 12232342343452534534543 5);` fehlerbehaftet. Der Datentyp `int` ist in Java standardmässig.

An das Ende von Ganzzahlliteralen vom Typ `long` wird ein `l` oder ein `L` gesetzt. Dabei wird `System.out.println( 12232342343452534534543 5L);` gültig.

Ein `byte` ist ein Datentyp mit einem kleineren Wertebereich. Eine Initialisierung `byte b = 200;` ist fehlerbehaftet. Eine explizite Typumwandlung lässt Zahlen in einem `byte` speichern und zwar `byte b = (byte) 200;` $\Longrightarrow$ `-56`.

Der Datentyp `short` stehen 16 Bits, (1 Bit für das Vorzeichen und 15 Bit für

die Zahlen) Speicher zur Verfügung. Ein `short` ohne Vorzeichen kann folgendermassen initialisiert werden: `short s = (short) 3300;` $\Longrightarrow$ `-32536`

2.3.4 Wahrheitswerte

Der Datentyp `boolean` beschreibt einen Wahrheitswert, der entweder `true` oder `false` ist. Diese sind reservierte Wörter und bilden neben konstanten Strings und primitiven Datentypen Literale. Numerische Werte werden nicht als Wahrheitswerte interpretiert. Der boolesche Typ wird für Bedingungen, Verzweigungen oder Schleifen benötigt. Ein Wahrheitswerte ergibt sich aus Vergleichen.

2.3.5 Unterstriche in Zahlen

Eine Variante um grosse Zahlen mit viele Nullen zu schreiben ist es, Unterstriche in Zahlen einzusetzen, denn ein Unterstrich gliedert die Zahl in Blöcke. Unterstriche machen Tausender-Blöcke gut sichtbar. Hilfreich ist die Schreibweise auch bei Literalen in Binär- und Hexadezimaldarstellung. Mit `0b` beginnt ein Literal in Binärschreibweise und mit `0x` beginnt ein Literal in Hexadezimalschreibweise. Zwei aufeinanderfolgende Unterstriche sind aber nicht erlaubt und er darf nicht am Anfang stehen.

2.3.6 Alphanumerische Zeichen

Der alphanumerische Datentyp `char` ist 2 Byte gross und nimmt ein Unicode-Zeichen auf. Ein `char` ist nicht vorzeichenbehaftet. Die Literale werden in Hochkommata (nicht Anführungszeichen) gesetzt. Ein `char` kann automatisch in ein `int` konvertiert werden.

2.3.7 Initialisierung von lokalen Variablen

Die Laufzeitumgebung bzw. der Compiler initialisiert lokale Variablen nicht automatisch mit einem Nullwert bzw. einen `false`. Sind Variablen nicht initialisiert, so gibt es Fehlermeldungen.

2.4 Ausdrücke, Operanden und Operatoren

Mathematische Ausdrücke bestehen aus **Operanden** und **Operatoren**. Ein Operand ist eine Varaible, ein Literal oder Rückgabe eines Methodenaufrufs. Die Operatoren verknüpfen die Operanden. Je nach Anzahl der Operanden unterscheidet man folgende Arten von Operatoren:

- Ist ein Operator auf genau einem Operanden definiert, so nennt er sich unärer Operator. Bsp: Negatives Vorzeichen.
- Die üblichen Operatoren für mathematische Ausdrücke sind binäre Operatoren.
- Das Fragezeichen-Operator für bedingte Ausdrücke ist ein tertiäres Operator.

2.4.1 Zuweisungsoperator

Das Gleichheitszeichen = dient in Java der Zuweisung. Der Zuweisungsoperator ist ein binärer Operator, bei dem auf der linken Seite due zu belegende Variable steht und auf der rechten Seite ein Ausdruck. Erst nach dem Auswerten des Ausdrucks kopiert der Zuweisungsoperator das Ergebnis in die Variable. Division durch Null, so gibt es keinen Schreibzugriff auf die Variable. Zuweisungen können geschachtelt werden.

2.4.2 Arithmetische Operatoren

Ein arithmetischer Operator verknüpft die Operanden mit den Operatoren Addition (+), Subtraktion (-), Multiplikation (*), Division (/) und den Rest-Operator (%). Die arithmetische Operatoren sind binär.

Bei Ausdrücken mit unterschiedlichen numerischen Datentypen, bringt der Compiler vor der Anwendung der Operation alle Operanden auf den umfassenderen Typ. Vor der Auswertung von `1+2.0` wird die Ganzzahl `1` in ein `double` konvertiert und dann die Addition vorgenommen - das Ergebnis ist auch vom Typ `double`. Das nennt sich **numerische Umwandlung**. Die Operation wird ausgeführt, und der Ergebnistyp entspricht dem umfassenden Typ.

Der binäre Operator bildet den Quotienten aus Dividend und Divisor. Die Division ist für Ganzzahlen und für Fliesskomazahlen definiert. Bei der Ganzzahldivision wird zu null hin gerundet und das Ergebnis ist keine Fliesskomazahl. Den Datentyp des Ergebnisses bestimmen die Operanden und nicht der Operator. Soll das Ergebnis vom Typ `double` sein, muss mindestens ein Operand ebenfalls double sein.

2.4.3 Der Restwert-Operator %

Der Restwert-Operator liefert der Rest einer Division zweier Ganzzahlen und Fliesskomazahlen. Die DIvision und der Restwert richten sich nach einer einfachen Formel: `(int)(a/b)*b+(a%b)=a`. Das Ergebnis ist nur dann negativ, wenn der Dividend negativ ist; das Ergebnis ist nur dann positiv, wenn der Dividend positiv ist. Um mit `value%2 == 1` zu testem, ob `value` eine ungerade Zahl ist, muss `value` positiv sein.

2.4.4 Präfix- oder Postfix-Inkrement und -Dekrement

Die Operatoren ++ und - kürzen die Programmzeilen zum Inkrement und Dekrement ab. Eine lokale Variable muss allerdings vorher initialisiert sein, da ein Lesezugriff vor einem Schreibzugriff stattfindet. Beide Operatoren erfüllen somit zwei Aufgaben: Neben der Wertrückgabe gibt es eine Veränderung der Variablen.

	Präfix	Postfix
Inkrement	Prä-Inkrement, `++i`	Post-Inkrement, `i++`
Dekrement	Prä-Dekrement, `-i`	Post-Dekrement, `i-`

Tab. 2.3: Präfix- oder Postfix-Inkrement und -Dekrement

Die beiden Operatoren liefern einen Ausdruck und geben daher einen Wert zurück. Es macht jedoch einen feinen Unterschied, wo dieser Operator platziert wird: Er kann vor der Variablen stehen, wie `++i` oder dahinter wie `i++`. Der **Präfix-Operator** verändert die Variable vor der Auswertung des Ausdrucks, und der **Postfix-Operator** verändert die Variable nach der Auswertung des Ausdrucks.

```
1  public class Prefixen
2  {
3      public static void main(String[] args)
4      {
5          int i = 10, j=20;
6          //Pre-Inkrement
7          System.out.println(++i);     //i=11
8          System.out.println(i);       //i=11
9          //Post-Inkrement
10         System.out.println(j++);         //j=20
11         System.out.println(j);           //j=21
12
13         int a = 10, b = 20;
14         //Pre-Inkrement
15         System.out.println(--a);     //a=9
16         System.out.println(a);       //a=9
17         //Post-Dekrement
18         System.out.println(b--);     //a=20
19         System.out.println(b);       //a=19
20
21     }
22 }
```

2.4.5 Auswertung bei Array-Zugriffen

Falls die linke Seite beim Verbundoperator ein Array-Zugriff ist, wird die Indexberechnung nur einmal vorgenommen. Dies ist wichtig beim Einsatz vom Präfix-/Postfix-Operator oder von Methodenaufrufen, die Nebenwirkungen besitzen, also etwa Zustände wie einen Zähler verändern.

2.4.6 Zuweisung mit Operation (Verbundoperator)

Zuweisungen lassen sich mit numerischen Operatoren kombinieren. Für einen binären Operator (symbolisch `#` genannt) im Ausdruck `a = a#(b)` kürzt der Verbundoperator den Ausdruck zu `a#b` ab. Der Verbundoperator erlaubt eine kompakte Schreibweise.

2.4.7 Relationale und Gleichheitsoperatoren

Relationale Operatoren sind Vergleichsoperatoren, die Ausdrücke miteinander vergleichen und einen Wahrheitswert vom Typ `boolean` ergeben. Die numeri-

sche Vergleiche sind: grösser (>), kleiner (<), grösser/gleich (≥), kleiner/gleich (≤), Gleichheit (==), Ungleichheit (!=).

2.4.8 Logische Operatoren

Die Programmierung ist an Bedingungen verknüpft. Diese Bedingungen sind komplex zusammengesetzt, wobei drei Operatoren am häufigsten vorkommen. **Nicht !:** (Negation) dreht die Aussage um; aus `wahr` wird `falsch` und aus `falsch` wird `wahr`. **Und &&:** (Konjunktion) beide Aussagen müssen `wahr` sein, damit die Gesamtaussage `wahr` wird. **Oder ||:** (Disjunktion) eine der beiden Aussagen muss `wahr` sein, damit die Gesamtaussage `wahr` wird. **Xor ^:** (Exklusives Oder) Operation, die nur dann `wahr` liefert, wenn genau einer der beiden Operanden `wahr` ist. Sind beide Operanden gleich, so ist das Ergebnis `false`.

boolean a	boolean b	!a	a&&b	a||b	a^b
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	true

Tab. 2.4: Verknüpfungen der logischen Operatoren.

2.4.9 Rang der Operatoren

Neben Plus und Mail gibt es eine Vielzahl von Operatoren., die alle ihre eigenenn Vorrangregeln besitzen. Der Multiplikationsoperator besitzt eine höhere Priorität als der Plus-Operator. Der **arithmetische Typ** steht für Ganz- und Fliesskommazahlen, der **integrale Typ** für `char` und Ganzzahlen und der Eintrag primitiv für jegliche primitiven Datentypen.

Operator	Rang	Typ	Beschreibung
++, -	1	arithmetisch	Inkrement und Dekrement
+, -	1	arithmetisch	unäres Plus und Minus
~	1	integral	bitweises Komplement
!	1	boolean	logisches Komplement
(Typ)	1	jeder	Cast
*, /, %	2	arithmetisch	Multiplikation, Division, Rest
+, -	3	arithmetisch	Addition und Subtraktion
+	3	String	String-Konkatenation
«	4	integral	Verschiebung links
»	4	integral	Rechtsverschiebung mit Vorzeichenerweiterung
»>	4	integral	Rechtsverschiebung ohne Vorzeichenerweiterung
<, <=, >, >=	5	arithmetisch	Numerische Vergleiche
instanceof	5	Objekt	Typvergleich
==, !=	6	primitiv	Gleich-/Ungleichheit von Werten
==, !=	6	Objekt	Gleich-/Ungleichheit von Referenzen
&	7	integral	bitweises Und
&	7	boolean	logisches Und
^	8	integral	bitweises XOR
^	8	boolean	logisches XOR
|	9	integral	bitweises Oder
|	9	boolean	logisches Oder
&&	10	boolean	logisches konditionales Und, Kurzschluss
||	11	boolean	logisches konditionales Oder, Kurzschluss
?:	12	jeder	Bedingungsoperator
=	13	jeder	Zuweisung
*=, /=, %=	13	arithmetisch	Zuweisung mit Operation
+=, =, «=	13	arithmetisch	Zuweisung mit Operation
»=, »>=, &=	13	arithmetisch	Zuweisung mit Operation
^=, |=	13	arithmetisch	Zuweisung mit Operation
+=	14	String	Zuweisung mit String-Konkatenation

Tab. 2.5: Operatoren mit Rangordnung

2.4.10 Die Typumwandlung (Casting)

Datentypen können konvertiert werden, dies nennt sich **Typumwandlung**. Java unterscheidet zwischen zwei Arten der Typumwandlung. EIne Typumwandlung hat eine sehr hohe Priorität. DAher muss der Ausdruck gegebenfalls geklammert werden.

- **Implizite Typumwandlung:** Daten eines kleineren Datentyps werden automatisch dem grösseren angepasst. Der Compiler nimmt die Anpassung selbständig vor.
- **Explizite Typumwandlung:** Ein grösserer Typ kann einem kleineren Typ mit möglichem Verlust von Informationen zugewiesen werden.

Werte der Datentypen `byte` und `short` werden bei Rechenoperationen automatisch in den Datentyp `int` umgewandelt. Ist ein Operand vom Datentyp `long`, dann werden alle Operanden auf `long` erweitert. Wird aber `short` oder `byte` als

Ergebnis verlangt, dann ist dieses durch einen expliziten Typecast anzugeben, und nur die niederwertigen Bits des Ergebniswerts werden übergeben.

Vom Typ	in den Typ
`byte` (8 Bit)	`short`, `int`, `long`, `float`, `double`
`short` (16 Bit)	`int`, `long`, `float`, `double`
`char` (16 Bit)	`int`,`long`, `float`, `double`
`int` (32 Bit)	`long`, `float`, `double`
`long` (64 Bit)	`float`, `double`
`float` (32 Bit)	`double`
`double` (64 Bit)	`double`

Tab. 2.6: Implizite Typumwandlungen

Die Anpassung ist eine Erweiterung des Wertebereichs (widening conversion). Der Typ `boolean` taucht nicht auf, er lässt sich in keinen anderen primitiven Typ konvertieren. Dass ein `long` auf ein `double` gebracht werden kann bzw. ein `int` auf ein `float` ist als Fehler in der Java zu sehen, denn es gehen Informationen verloren. Ein `double` kann die 64 Bit für Ganzzahlen nicht effizient nutzen wie ein `long`.

Die explizite Anpassung engt einen Typ ein (narrowing conversion). Der gewünschte Typ für eine Typumwandlung wird vor den umzuwandelnden Datentyp in Klammern gesetzt. Bei jeder expliziten Typumwandlung geht Information verloren.

Bei der expliziten Typumwandlung von `double` und `float` in einen Ganzzahltyp kann es selbstverständlich zum Verlust von Genauigkeit kommen sowie zur Einschränkung des Wertebereichs. Bei der konvertierung von Fliesskommazahlen verwendet Java eine Rundung gegen null, schneidet also schlicht den Nachkommaanteil ab.

```
1  public class Typumwandlung
2  {
3      public static void main(String[] args)
4      {
5          System.out.println((int)+12.34);     //12
6          System.out.println((int)-12.34);     //-12
7          System.out.println((int)(-12.34+2.1));     //-12
8          int r = (int)(Math.random()*5);      //0<=r<=4, r in N
9          //Initialisierungen
10         short short1=1, short2 = 2;
11         short byte1=1, byte2 = 2;
12         int int1=1, int2 = 2;
13         long long1=1, long2 = 2;
14         //Summenarten
15         short short3 = (short)(short1 + short2);
16         short byte3 = (short)(byte1 + byte2);
17         int int3 = int1 + int2;
18         long long3 = long1 + long2;
19         System.out.println(short3);
20         System.out.println(byte3);
21     }
22 }
```

Die String-Konkatenation ist strikt von links nach rechts und natürlich nicht kommutativ wie die numerische Addition. Besteht der Auddruck aus mehreren Teilen, so muss die Auswertungsreihenfolge beachtet werden, andernfalls kommt es zu seltsamen Zusammensetzungen.

```
1  public class PlusString
2  {
3      public static void main(String[] args)
4      {
5          System.out.println(1+2);                  //3
6          System.out.println("1"+2+3);              //123
7          System.out.println(1+2+"3");              //33
8          System.out.println(1+2+"3"+4+5);          //3345
9          System.out.println(1+2+"3"+(4+5));        //339
10         System.out.println(1+2+"3"+(4+5)+6);      //3396
11     }
12 }
```

Nur eine Zeichenkette in doppelten Anführungszeichen ist ein String, und der Plus-Operator entfaltet seine besondere Wirkung. Ein einzelnes Zeichen in einfachen Hochkommata konvertiert Java nach den Regeln der Typumwandlung bei Berechnungen in ein `int` und Additionen sind Ganzzahl-Additionen.

```
1  public class PlusZeichen
2  {
3      public static void main(String[] args)
4      {
5          System.out.println('0' + 2);     //50, '0'=48
6          System.out.println('A' + 'a');   //162, 'A'=65, 'a'=97
7      }
8  }
```

2.5 Bedingte Anweisungen

2.5.1 Verzweigung mit der `if`-Anweisung

Die `if`-Anweisung besteht aus dem Schlüsselwort `if`, dem zwingend ein Ausdruck mit dem Typ `boolean` in Klammern folgt. Es folgt eine Anweisung, die oft eine Blockanweisung ist.

```
1  public class WhatsYourNumber
2  {
3      public static void main(String[] args)
4      {
5          int number = (int) (Math.random() * 5 + 1);
6          System.out.println("Welche Zahl denke ich mir zwischen 1
               und 5?");
7          int guess = new java.util.Scanner( System.in ).nextInt()
               ;
8
9          if(number == guess)
10         {
11             System.out.println("Super getippt!");
12         }
13         else
14         {
15             System.out.printf("Stimmt nicht, habe mir %s gedacht
                   !", number);
16         }
```

```
    }
}
```

Ist das Ergebnis in der Bedingung `true`, so werden die Anweisungen in der Fallunterscheidung ausgeführt, sonst werden die else-Anweisungen ausgeführt. Eine Fallunterscheidung hat kein Semikolon. `if` und `if-else`-Anweisungen werden geschachtelt (kaskadiert).

```
public class IsLeapYear
{
    public static void main(String[] args)
    {
        int month = 2;
        boolean isLeapYear = false;
        int days;

        if(month == 4)
            days = 30;
        else if(month == 6)
            days = 30;
        else if(month == 9)
            days = 30;
        else if(month == 11)
            days = 30;
        else if(month == 2)
            if(isLeapYear)
                month = 29;
            else
                days = 28;
        else
            days = 31;
    }
}
```

Die eingerückten Verzweigungen nennen sich auch angehäufte `if`-Anweisungen oder `if`-Kaskade.

2.5.2 Der Bedingungsoperator

Der Bedingungsoperator, auch Konditionaloperator, erlaubt es, den Wert eines Ausdrucks von einer Bedingung abhängig zu machen, ohne dass dazu eine `if`-Anweisung verwendet werden muss. Die Operanden sind durch `?` und `:` voneinander getrennt.

```
public class BedingungsOperator
{
    public static void main(String[] args)
    {
        int max;
        int a = 10, b = 5;
        max = (a > b) ?  a: b; //if(a>b) max = a; else max = b;
        System.out.println("Die groesste Zahl ist " + max);
    }
}
```

Drei Ausdrücke kommen in Bedingunsoperator vor. Der erste Ausdruck muss vom Typ `boolean` sein. Der Bedingungsoperator kann eingesetzt werden, wenn der zweite und dritte Operand ein numerischer Typ, boolescher Typ oder Referenztyp sind.

2.5.3 Die `switch`-Anweisung

Eine Kurzform für speziell gebaute, angehäufte `if`-Anweisungen bietet `switch`. Es gibt eine Reihe von unterschiedlichen Sprungzeilen, die mit `case` markiert sind. Die `switch`-Anweisung erlaubt die Auswahl vin Ganzzahlen, Wrapper-Typen, Aufzählungen und Strings.

```
1  public class Calculator
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Zahl 1:");
6          double x = new java.util.Scanner(System.in).nextDouble()
               ;
7          System.out.println("+, -, * oder /:");
8          char operator = new java.util.Scanner(System.in).
               nextLine().charAt(0);
9          System.out.println("Zahl 2:");
10         double y = new java.util.Scanner(System.in).nextDouble()
               ;
11
12         switch(operator)
13         {
14             case '+':
15                 System.out.println(x+y);
16             break;
17             case '-':
18                 System.out.println(x-y);
19             break;
20             case '*':
21                 System.out.println(x*y);
22             break;
23             case '/':
24                 System.out.println(x/y);
25             break;
26         }
27     }
28 }
```