

# Introduction to Programming

Wolfgang Schreiner  
Research Institute for Symbolic Computation (RISC)  
Johannes Kepler University, Linz, Austria

[Wolfgang.Schreiner@risc.jku.at](mailto:Wolfgang.Schreiner@risc.jku.at)

July 11, 2022

## **Abstract**

We introduce the basics of computer science, i.e., the science of problem solving with computers. Our presentation starts with an abstract view on programming and algorithm development and continues with the discussion of how real programs are executed on computers. We then describe the programming language Java beginning with basic constructs for writing small programs that solve simple problems and proceeding to more advanced features for developing larger programs that solve more complex problems. Throughout this process we touch fundamental topics of computer science such as data structures, algorithms, and systematic program development.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>The Basics of Programming</b>	<b>7</b>
2.1	Variables . . . . .	7
2.2	Assignments . . . . .	8
2.3	Sequences . . . . .	10
2.4	Conditionals . . . . .	11
2.5	Loops . . . . .	12
2.6	Example Algorithms . . . . .	13
2.6.1	Swapping Two Variables . . . . .	14
2.6.2	Maximum of Three Numbers . . . . .	16
2.6.3	Primality Test . . . . .	17
<b>3</b>	<b>Programming Computers</b>	<b>20</b>
3.1	Computer Systems . . . . .	20
3.1.1	The Main Memory . . . . .	21
3.1.2	The Processor . . . . .	23
3.2	Programming Languages . . . . .	27
3.2.1	Assembly Language . . . . .	27
3.2.2	High-level Languages . . . . .	28
3.2.3	Java . . . . .	30

---

<b>4</b>	<b>Java Light</b>	<b>33</b>
4.1	Java Applications	33
4.2	Source Code Conventions	35
4.3	Basic Text Output	37
4.4	Basic Graphical Output	39
4.5	Variables and Assignments	42
4.5.1	Variable Declarations	42
4.5.2	Assignments	43
4.5.3	Constant Declarations	44
4.5.4	Choosing Variable Names	45
4.6	Primitive Data Types and Expressions	46
4.6.1	Booleans	46
4.6.2	Integers	47
4.6.3	Characters	54
4.6.4	Floats	55
4.6.5	Strings	60
4.7	Basic Input	64
4.7.1	Error Handling	65
4.8	Control Structures	65
4.8.1	Assertions	66
4.8.2	The Block Statement	67
4.8.3	The if Conditional	69
4.8.4	The if-else Conditional	71
4.8.5	The switch Conditional	77
4.8.6	The while Loop	79
4.8.7	break and continue	82
4.8.8	The do Loop	86
4.8.9	The for Loop	89
4.9	Sample Programs	93
4.9.1	Printing Stars	93
4.9.2	Drawing Circles	95
4.9.3	Analyzing Exam Grades	99

---

<b>5</b>	<b>Stronger Java</b>	<b>108</b>
5.1	Methods . . . . .	108
5.1.1	Method Declarations and Calls . . . . .	108
5.1.2	The return Statement . . . . .	111
5.1.3	Local and Global Variables . . . . .	113
5.1.4	Parameters . . . . .	116
5.1.5	Functions . . . . .	119
5.1.6	Program Function versus Mathematical Functions . . . . .	121
5.1.7	Visibility of Variables . . . . .	124
5.1.8	Method Overloading . . . . .	125
5.1.9	Documenting Methods . . . . .	126
5.2	Objects and Classes . . . . .	129
5.2.1	Objects . . . . .	131
5.2.2	Classes . . . . .	134
5.2.3	Object Creation . . . . .	135
5.2.4	Constructors . . . . .	135
5.2.5	Transient Parameters . . . . .	139
5.2.6	Structuring Programs . . . . .	140
<b>A</b>	<b>Software Tools</b>	<b>145</b>
A.1	Oracle Java SE . . . . .	145
A.2	TextPad . . . . .	146
A.3	Eclipse IDE . . . . .	148

# Chapter 1

## Introduction

This course is about problem solving with computers. Computers are machines whose behavior, unlike that of other machines, is not determined by their physical construction (the *hardware*). Rather computers are controlled by *programs*, i.e., formal descriptions of the behavior that we want the computer to perform. Programs are *software*, i.e., intangible units like thoughts. Software and hardware together make up a *computer system*; today's computer systems are linked by *networks* for communication and cooperation within an organization (Intranet) and across the globe (Internet).

A *programming language* is a formal notation for writing programs; like a thought may be formulated in different human languages, a program may be written in different programming languages. An *algorithm* is a method for solving a given problem in a particular way; like an idea may be expressed by different words (or in different languages), an algorithm may be implemented by different programs or in different programming languages.

The process of solving problems with computers proceeds in various phases:

1. the analysis of the problem *requirements* yielding a
2. *specification* which describes in an unambiguous and concise way *what* problem is to be solved; based on this specification, we
3. *design* the problem solution by decomposing the problem into subproblems that we decompose further until we are able to
4. use known algorithms (or design new ones) that describe *how* the subproblems can be solved; we then

Software can be stored on a physical medium like a thought can be written on paper; however like a thought is not the paper on which it is written, software is not the the medium on which it is stored.

Errors detected during debugging or in the test phase may force us to rewrite the program, redesign the problem solution, rewrite the specification, even reanalyze the problem. Software development is an *iterative* process.

This description only *sketches* the development process; the art of *software engineering* is to manage large projects with teams of software developers solving complex problems.

5. analyse the design, consider alternatives, and *refine* it until we are convinced of its soundness such that we can go and
6. *implement* the solution as a well structured and clearly documented program in a particular programming language; we then
7. run the program and *debug* it, i.e., fix errors which let the program malfunction or crash, afterwards we
8. *test* the program in a systematic way to exhibit any less apparent errors of the problem solution which have to be corrected; finally we
9. *package* the program together with all the documentation that describes its implementation and use and ship it to the customer.

From above description, it should become clear that problem solving with computers is more than programming, i.e., learning a programming language and writing programs in this language. However, programming is still the *core activity* in this process. It is a *craft* which we must thoroughly trained such that we can free our minds from technical details and concentrate on the other aspects of the development process (like the writer of a book does not have to think any more about how to put letters on paper). It is also an *art* in which creativity and intuition combine to find elegant and efficient solutions; many people can program but there are only very few master programmers (like most people learn reading or writing but only few are good writers).

Computer programming also shapes our minds and strengthens our discipline. We cannot cheat by omitting unpleasant parts of a problem or by not refining a solution in sufficiently much detail; ultimately the computer is the referee that determines whether a result works or not. Writing programs is therefore a good training methodology for analyzing and solving (mostly but not only) technical problems. Even if your career goal is not to write programs but manage technical projects, analyze systems for their suitability for a particular purpose, design system solutions by using existing products, or perform corresponding consulting activities, you will therefore profit from a solid education in software development.

We will in this script focus on the craft of computer programming and use the programming language *Java* as a tool to demonstrate the fundamental concepts of modern software development. We will, however, not forget the other aspects sketched above and repeatedly demonstrate the systematic development process by concrete examples.

# Chapter 2

## The Basics of Programming

A *program* is the formal description of a method that solves a particular problem. Every program consists of individual *instructions* (also called *commands* or *statements*) that operate on *data* (values like numbers, characters, text files, images, ...). In this chapter, we explain in an abstract notation those elements of programming that are (essentially) *independent* of concrete programming languages and computer systems, i.e. valid for any of them. The subsequent chapters will describe the programming of real computers in a particular real language.

### 2.1 Variables

Data are stored in *variables*. One can think of a variable as a box that has a label on it (the *name* of the variable) and that holds a *value* (also called the *content* of the variable). For example, a variable named  $x$  holding the value 17 can be depicted as the box

$x$ : 

17
----

A program in general uses multiple variables that may hold values of various types. For instance, the picture

$x$ : 

17
----

  
 $ch$ : 

'+'
-----

  
 $str$ : 

"hello"
---------

depicts a variable  $x$  holding an integer number 17, a variable  $ch$  holding a character '+' and a variable  $str$  holding a string of characters "hello".

A program may *change* the values of its variables. E.g., after the execution of a program operating on above variables, they may have contents

```
x: 18
ch: '*'
str: "world"
```

While the value of a variable may thus change over time, it may not change arbitrarily. Every variable has an associated *type* that describes the set of values that the variable may hold during its whole life time, e.g. the type *integer* for variable *x* or *character* for *ch* or *string* for *str*. The type of a variable remains fixed during its whole life time; it may be also attached to the variable's visual representation such as in

```
x: 17 integer
ch: '+' character
str: "hello" string
```

Real programming languages are (for good reasons) very picky about the types of variables and which kinds of values they may hold. In this chapter, all variables will range over integer numbers such that explicit type annotations can be omitted.

## 2.2 Assignments

The simplest and most important instruction in almost every programming language is the *variable assignment* (short *assignment*) statement which has the form

$$x \leftarrow E$$

where *x* is a variable and *E* is a mathematical expression. The statement is read as “*x* becomes *E*”; it computes the value of *E* and stores it in *x* (overwriting the previous value of *x*).

For instance, if we have the variable contents

```
x: 5
y: 1
```

the execution of the command

$$y \leftarrow 2$$

yields the variable contents

$$\begin{array}{l} x: \boxed{5} \\ y: \boxed{2} \end{array}$$

If we then execute the command

$$y \leftarrow x + 1$$

(which takes the content of the variable  $x$ , adds 1 to it, and stores the result in variable  $y$ ), we get

$$\begin{array}{l} x: \boxed{5} \\ y: \boxed{6} \end{array}$$

where  $y$  has been updated but  $x$  remains unchanged. We may then execute the command

$$x \leftarrow x + 1$$

which *first* reads the value of  $x$ , adds 1 to it, and *then* updates the value of  $x$  by the result. The variable contents after the execution are consequently

$$\begin{array}{l} x: \boxed{6} \\ y: \boxed{6} \end{array}$$

If we now execute

$$x \leftarrow x * y$$

we get

$$\begin{array}{l} x: \boxed{36} \\ y: \boxed{6} \end{array}$$

The execution of a program essentially consists of a sequence of variable assignments. All other programming language constructs are essentially *control structures* that determine the “flow” of assignments executed by a program.

## 2.3 Sequences

A program may consist of a *command sequence* (short: *sequence*) of assignments that are executed one after another. For instance, if we have the variable contents

$x$ :  $\boxed{5}$   
 $y$ :  $\boxed{1}$

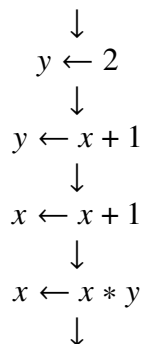
the execution of the program

$y \leftarrow 2$   
 $y \leftarrow x + 1$   
 $x \leftarrow x + 1$   
 $x \leftarrow x * y$

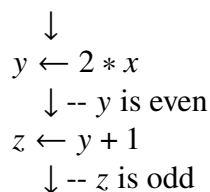
ultimately yields the contents

$x$ :  $\boxed{36}$   
 $y$ :  $\boxed{6}$

Above command sequence may also be depicted by a *control flow diagram*



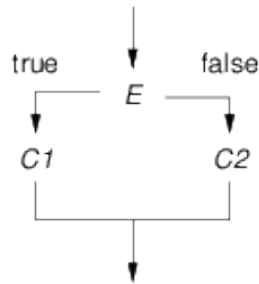
which uses arrows to show the order in which the individual commands are executed. These arrows may be also labelled by *assertions* i.e. propositions about the values of the variables at a certain state of the program, e.g.



(the dashed line -- shows the position in the program when the assertion holds).

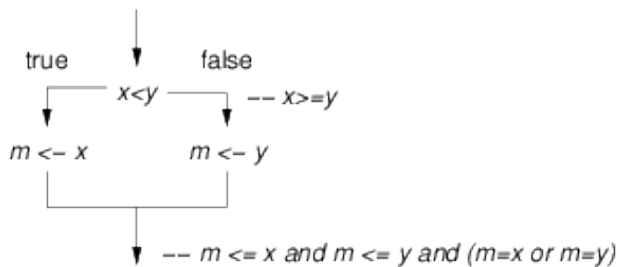
## 2.4 Conditionals

It is frequently the case that a command is only to be executed if a particular condition holds. This gives rise to a *conditional statement* (short *conditional*) with control flow diagram



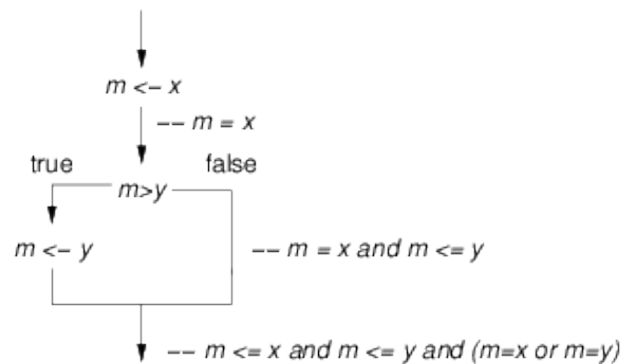
where  $E$  denotes a condition,  $C1$  denotes a command that is executed, if  $E$  is true, and  $C2$  denotes a command that is executed if  $E$  is false.

For example, the program



sets the variable  $m$  to the minimum of the variables  $x$  and  $y$  i.e. the smaller of both values. When the control flow reaches the condition  $x < y$ , it branches either to the left or to the right. If  $x < y$  is true, the left branch of the conditional, i.e. the assignment  $m \leftarrow x$ , is executed; if  $x < y$  is false (i.e.  $x \geq y$  is true), the right branch, i.e. the assignment  $m \leftarrow y$ , is executed. After the execution of the conditional (when both branches converge),  $m$  holds in either case the minimum of  $x$  and  $y$ . The assertion denoted by  $--$  describes the knowledge that we have in the corresponding branch of the conditional.

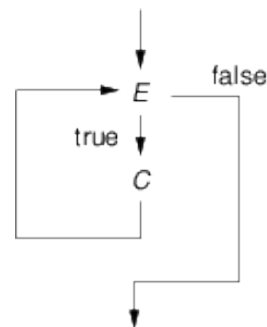
One branch of a conditional (but not both) may be empty; in this case, no command is executed if this particular branch is taken. Take for example the following program for computing the minimum of two values in a slightly different way:



This program first sets the variable  $m$  to the value of  $x$  and then tests whether  $m > y$ . If this is the case, then  $y$  is the smaller of the two values and  $m$  is set to  $y$ . Otherwise,  $m$  is already the smaller of both values such that nothing needs to be done any more.

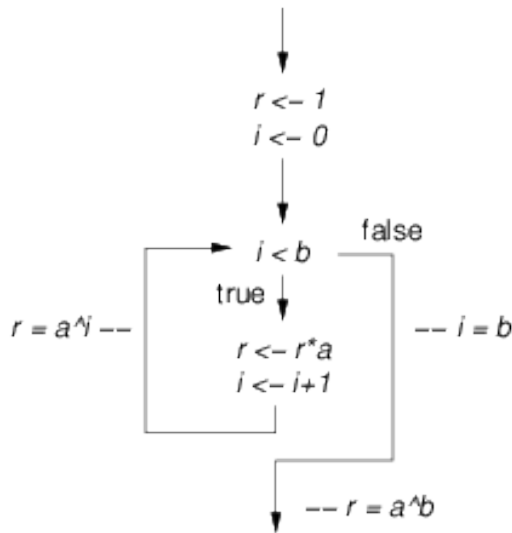
## 2.5 Loops

By a *loop statement* (short *loop*) one can express that a command  $C$  (the *loop body*) is *iterated* (repeatedly executed) as long as a condition  $E$  (the *loop condition*) remains true:



When the control flow reaches the loop,  $E$  is tested. If the condition is false, the loop is not executed at all. If, however, the condition is true, then the statement  $C$  is executed and the control flow returns to the beginning of the loop. Consequently,  $C$  is repeatedly executed as long as  $E$  remains true.

As an example take the following program



which operates on four natural number variables  $a$ ,  $b$ ,  $r$  and  $i$ . Initially,  $r$  is set to 1 and  $i$  to 0. As long as  $i$  is less than  $b$ , the program multiplies  $r$  by  $a$  and increments  $i$  by 1 (while  $a$  and  $b$  do not change).

To understand the behavior of this program, let us assume that  $a$  has initially the value 4 and  $b$  the value 3. Then the variables  $r$  and  $i$  have, before the test  $i < b$  is performed, the following values:

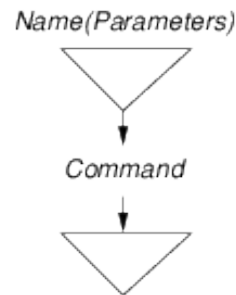
	$r$	$i$
Before first iteration	1	0
After first iteration	4	1
After second iteration	16	2
After third iteration	64	3

As one can see from this table, after  $i$  iterations,  $r$  has the value  $4^i$ . The loop is terminated after three iterations, thus  $r$  has finally the value  $4^3 = 64$ . For general values of  $a$  and  $b$ , this program thus sets  $r$  to  $a^b$  (please also note the corresponding assertions in the control flow diagram).

## 2.6 Example Algorithms

In the previous section, we have described the four kinds of statements (assignment, sequences, conditionals, loops) of which all programs are made of. These

statements may be also combined (e.g. one branch of a conditional may be a loop or the body of a loop may be a conditional) to all kinds of *algorithms*, i.e. precisely formulated problem solutions. In the following subsections, we will discuss the development of some simple algorithms. Every algorithm is described by a diagram of the form



where triangles initiate and terminate the control flow. The algorithm is given a name and a list of *parameters* i.e. variables that are provided by the user and on which the algorithm operates (in addition to auxiliary variables that are only used within the algorithm).

### 2.6.1 Swapping Two Variables

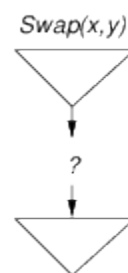
Our first task is to write an algorithm *Swap* that exchanges the values of two variables  $x$  and  $y$  provided by the user. For instance, starting with variable values

$x$ :   
 $y$ :

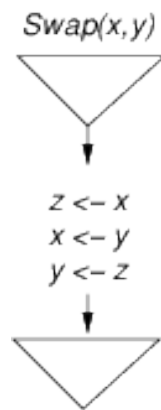
we would like to reach the state

$x$ :   
 $y$ :

We thus have to fill the body of the following algorithmic skeleton:



Thinking a bit about the problem, we realize that we cannot perform the assignment  $x \leftarrow y$  first, because this assignment erases the value of  $x$  provided by the user which is thus lost. Likewise, we cannot perform the assignment  $y \leftarrow x$  first, because this erases the value of  $y$ . The key insight is that we need a *third* variable  $z$  into which we can first save the value of e.g. variable  $x$ , such that we can then execute  $x \leftarrow y$  to set  $y$  to  $x$  and finally execute  $y \leftarrow z$  to set  $y$  to the (original) value of  $x$ . This gives the algorithm



We may simulate with paper and pencil the effect of this algorithm on the contents of the variables for particular initial settings. For instance, for the initial variable values  $x = 3$  and  $y = 5$ , we get the following table which lists the values of the variables at different times during the execution of the algorithm:

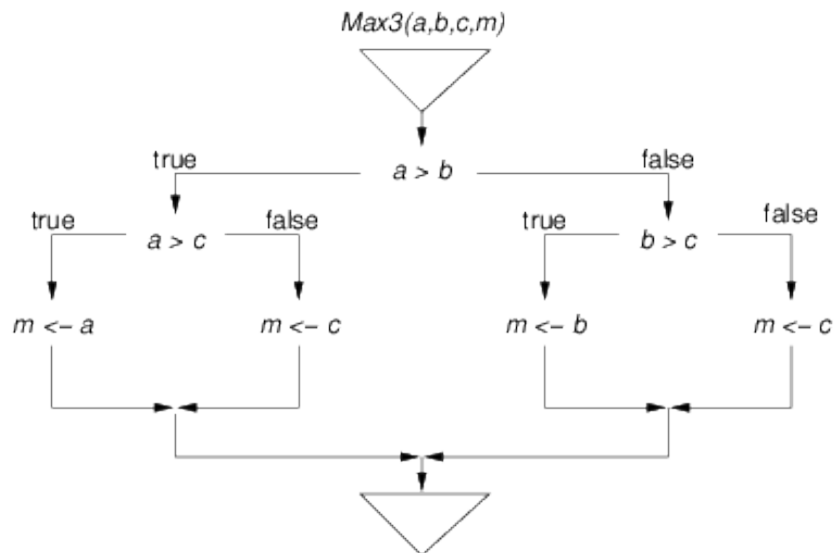
$x$	$y$	$z$	
3	5		Initial Situation
		3	After $z \leftarrow x$
5			After $x \leftarrow y$
	3		After $y \leftarrow z$

This confirms that the final value of  $x$  is 5 and the final value of  $y$  is 3.

By such paper and pencil simulations, we can “test” every algorithm i.e. we can simulate the effect of its application to concrete variable values. While this does *not* prove that the algorithm is correct for all possible initial variable contents, it at least helps to find errors and (if no more errors can be found) to increase the confidence in our solution.

## 2.6.2 Maximum of Three Numbers

We want to devise an algorithm *Max3* that takes three numbers  $a$ ,  $b$ , and  $c$  and sets a variable  $m$  to the largest of these values. The result is depicted below:



This algorithm has a more complex structure as those shown up to now. It consists of a conditional testing the expression  $a > b$ ; each branch of this conditional is again a conditional, one testing  $a > c$  and one testing  $b > c$ . Let us follow the line of reasoning that led to this structure.

- If  $a$  is greater than  $b$ , then the value of  $b$  is not a candidate for the maximum  $m$  any more, but  $a$  is. If now  $a$  is also greater than  $c$ , it is indeed the maximum, so we set  $m$  to  $a$ . Otherwise  $c$  is greater than or equal  $a$  (which as we remember is greater than  $b$ ), thus  $c$  is the maximum and we set  $m$  to  $c$ .
- If  $a$  is not greater than  $b$ , then  $b$  is greater than or equal  $a$ , so we can choose  $b$  as a candidate for  $m$ . If now  $b$  is also greater than  $c$ , then  $b$  is the maximum, so we set  $m$  to  $b$ . Otherwise,  $c$  is greater than or equal  $b$  (which as we remember is greater than or equal  $a$ ), thus  $c$  is the maximum and we set  $m$  to  $c$ .

This reasoning can be also expressed by assertions in the diagram:



Following the control flow that leads to each assignment and collecting the test conditions and assertions on this way allows one to quickly verify the correctness of this algorithm. Assertions represent a valuable aid for understanding the logic of a program; they should be regularly used in the development process.

### 2.6.3 Primality Test

We complete this chapter by the development of an algorithm *isPrime* that takes a natural number  $n$  and sets the variable  $p$  to “true”, if  $n$  is a prime number and to “false”, otherwise (recall that a prime number is a natural number greater than 1 that is only divisible by 1 and itself). In the following, we will discuss the solution depicted in Figure 2.1.

From the definition of primality, it immediately follows that a number less than 2 is not prime. We thus first test for  $n < 2$  and, if this is true, set  $p$  to “false”. Otherwise our basic assumption (unless proved otherwise) is that  $n$  is prime, so we initialize  $p$  with “true”.

Furthermore, we know that 2 is the only even prime number. In order to restrict our consideration to odd numbers, we treat this case specially and first test if  $n$  is 2. If yes, we are already done (remember that  $p$  is already set to “true”). Otherwise, if  $n$  is not 2 but nevertheless divisible by 2 (described by the predicate  $2|p$  to be read as “2 divides  $p$ ”), we have encountered a number that is surely not prime. We therefore set  $p$  to “false” and are also done.

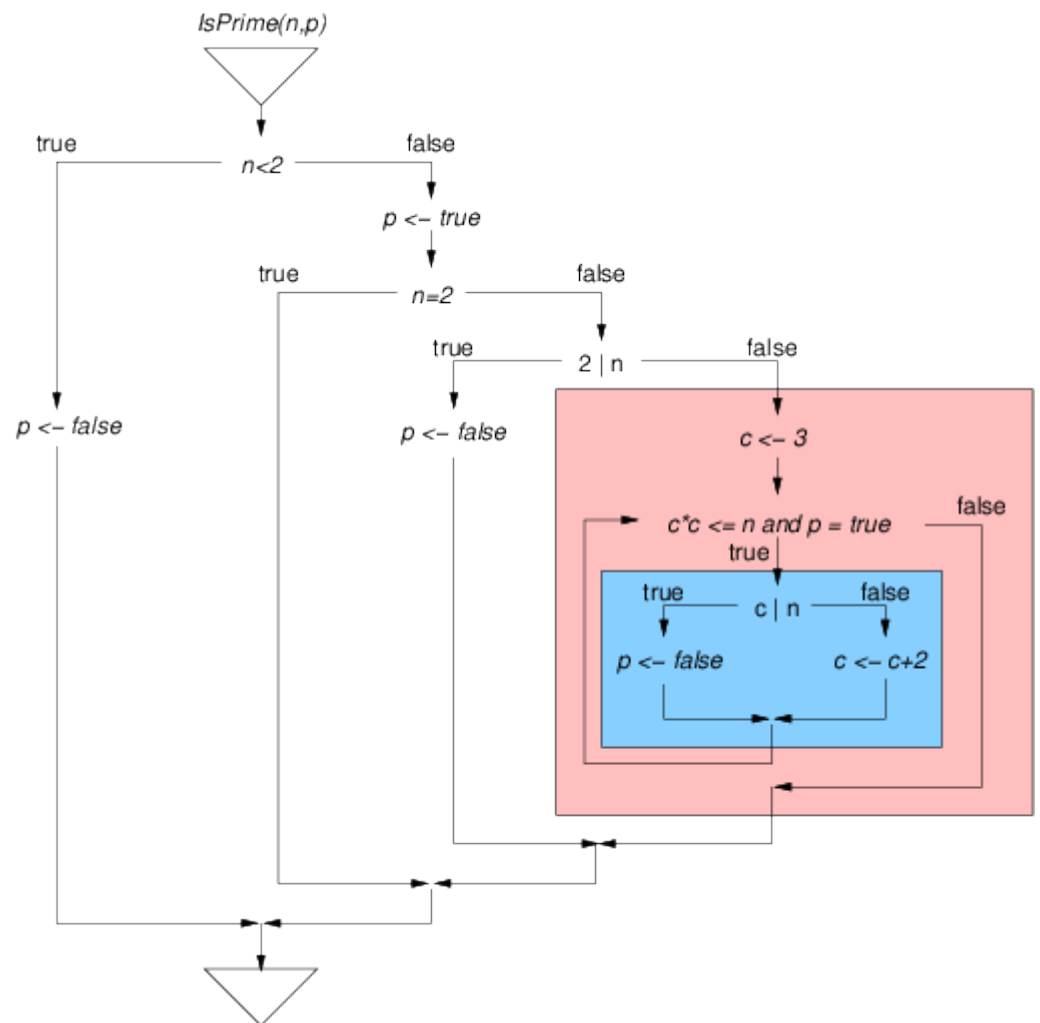


Figure 2.1: Primality Test

The only case left is that of an odd number  $n$  greater than one. To check for the primality of an odd number, we only have to check whether it is divisible by any other odd number greater than 2. All candidates for this “other odd number” are certainly in the range from 3 to  $n - 1$ . Even more, we may restrict our consideration to odd numbers from 3 to  $\sqrt{n}$  (because, if  $n$  is divisible by a number  $c$  greater than  $\sqrt{n}$ , then  $n$  equals  $c \cdot d$  for a number  $d$  less than  $\sqrt{n}$ , such that this  $d$  is also a divisor of  $n$ ).

Thus all we have to do is to check whether  $n$  is divisible by some number  $c$  such that  $c \geq 3$  and  $c \cdot c \leq n$ . The subalgorithm for this check is depicted by the big shaded box in Figure 2.1. We initially set the candidate  $c$  to 3 and check in a loop, as long as  $c \cdot c$  is less than or equal  $n$  and  $n$  can be still assumed prime, whether  $n$  is divisible by  $c$ . If yes, we can set  $p$  to “false” which terminates the loop. If no, we increment  $c$  by 2 (which gives the next odd number) and continue the test with the next candidate value.

The structure of this algorithm is much more complex than the ones illustrated up to now: it contains a loop that is part of a conditional statement (which itself is nested into two other conditionals) and the body of the loop is itself a conditional. Despite of their apparent complexity, such constructions are routine in software development. While this is clearly very difficult for the beginner, by a systematic top-down construction of algorithms using the sort of reasoning shown above as well as by regular exercising and the corresponding accumulation of experience, the development of (simple) algorithms will soon become familiar.

# Chapter 3

## Programming Computers

In the last chapter, we have considered programs from a rather abstract point of view as formal descriptions of methods for solving particular problems. In this chapter, we will become a bit more concrete and discuss how such a description can actually control the behavior of a computer and what exactly happens inside the computer during program execution. For this purpose, we will first describe the internal architecture of a computer, then describe the low-level language that is understood by a computer and in which executable programs are encoded, and finally discuss the relationship to those programs that are written in a high-level programming language by a human programmer. In the later chapters, we will learn the basics of one such high-level programming language called *Java* for writing computer programs.

### 3.1 Computer Systems

The key hardware components of a computer system are

- the central processing unit (CPU), short *processor*,
- the random access memory (RAM), short *main memory*,

The processor executes programs that are stored in the main memory together with the data on which the program operates.

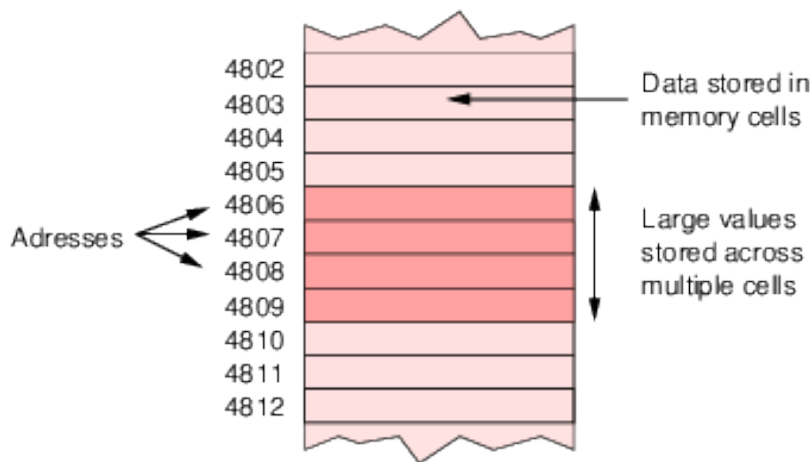


Figure 3.1: The Main Memory

### 3.1.1 The Main Memory

The main memory is made of a series of consecutive *memory cells* each of which stores a single natural number, the *memory content*, and is also addressed by a natural number, the *memory address* (see Figure 3.1). *Random Access Memory* (RAM) expresses the fact that each memory cell can be individually referenced by its address, i.e., a value can be read from or written into there. Figure 3.2 shows a line of memory chips to be plugged into a computer's main board with a single memory chip at the front.

The fact that a memory cell can only hold a natural number is not a limitation to the power of a computer: today's computers are *digital* i.e., all data (and programs) on all media are represented by discrete pieces which can be denoted by natural numbers.

**Example** *ASCII*<sup>1</sup> (American Standard Code for Information Interchange) is a code for the representation of 128 ( $=2^7$ ) *characters* (letters and other symbols) by the numbers 0–127:

<sup>1</sup><http://www.asciitable.com>

In *sequential* storage devices like hard disks, data can be only read or written in sequence.

The opposite of digital is *analog*: data are represented by continuous signals like for instance electromagnetic waves.

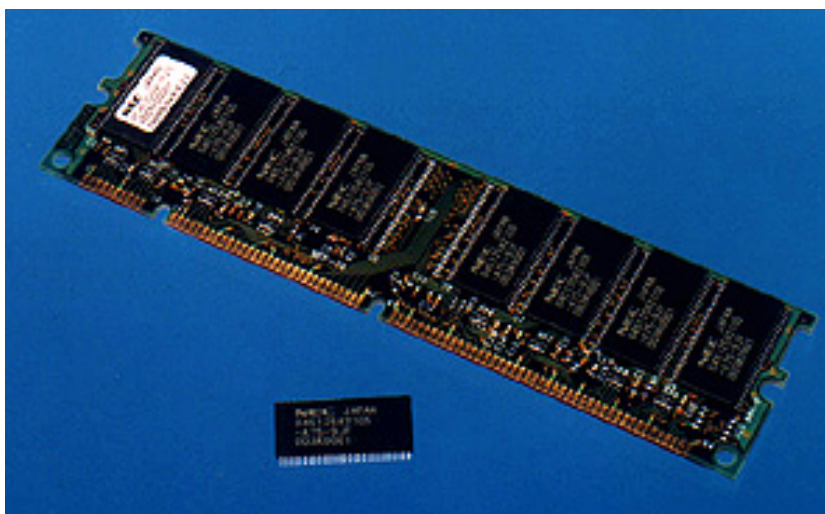


Figure 3.2: Memory Chips

ASCII code	Character
...	...
10	<i>LineFeed (LF)</i>
...	...
48–57	0–9
...	...
65–90	A–Z
...	...
97–122	a–z
...	...

The *LF* character is a *non-printable* character, it represents the shift from one line to the next.

Since *text* is a sequence of characters, it can be represented by a sequence of digits, e.g. the text “Hi, Heather.” can be represented in ASCII as

```

H   i   ,   H   e   a   t   h   e   r   .
72 105 44 32 72 101 97 116 104 101 114 46

```

The *ISO 8859-1*<sup>2</sup> character code represents 256 ( $=2^8$ ) characters of west European languages (Latin-1); the first 128 characters coincide with the ASCII standard.

<sup>2</sup><http://www.htmlhelp.com/reference/charset>



Figure 3.3: Main Board

The *Unicode*<sup>3</sup> character code represents 65534 ( $=2^{16} - 2$ ) characters (of almost all known languages); the first 256 characters coincide with the ISO 8859-1 standard.

### 3.1.2 The Processor

The processor is the most important (and the most expensive) part of a computer; it continuously fetches instructions from the main memory and executes them. Processor and main memory together represent the core of a computer; they sit together with several other components on the computer's *main board*. Figure 3.3 shows such a main board where the square black box to the right is the processor and the long lines on the left represent the memory bank that receives the memory chips.

The basic architecture of a computer and the processor is shown in Figure 3.4.

In detail, the processor consists of the following items:

<sup>3</sup><http://www.unicode.org/unicode/standard/WhatIsUnicode.html>

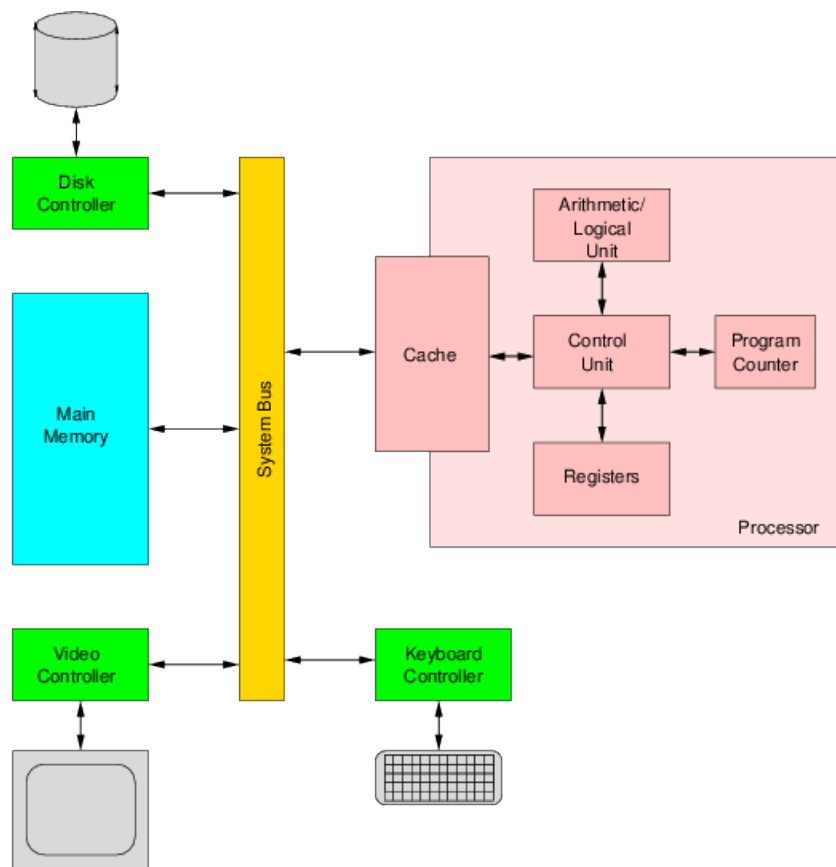


Figure 3.4: Basic Computer Architecture

- The *control unit* (CU) which coordinates the processing steps,
- the *arithmetic/logic unit* (ALU) which performs calculations and makes decisions,
- the *registers* which represents a small but very fast auxiliary memory within the processor,
- the *cache* which is a fast buffer memory between the processor and the actual main memory.

The program executed by the processor is (together with the data on which the program operates) stored in the main memory; it consists of a sequence of machine *instructions* each of which is encoded in a computer word. For instance, an instruction `load r1, 100` may be represented by a 4 byte word

17	1	0	100
----	---	---	-----

where 17 represents the operation code for `load`, 1 denotes register `r1` and the byte pair (0, 100) represents a 16 bit word containing the address 100. Typical instructions are

**load** *r, a*: Load the word at memory address *a* into register *r*.

**store** *a, r*: Store the word from register *r* into the memory at address *a*.

*op r1, r2*: perform an arithmetic operation *op* on registers *r1* and *r2* and store the result in *r1*.

**jump** *p*: Set the program counter to address *p*.

**cjmp** *r, c, p*: Compare the content of register *r* with constant *c*. If the comparison is successful, set the program counter to address *a*, otherwise do nothing.

The *program counter* is a special processor register which holds the address of the next instruction to be executed. The operation of the processor is now a continuous cycle of the following steps coordinated by the control unit:

1. fetch the instruction referenced by the program counter from the main memory,

2. decode the instruction and increment the program counter to reference the next instruction in sequence,
3. execute the instruction.

This operation principle is called the *von Neumann architecture* after John von Neumann who invented the concept in 1945.

---

**Example** Assume that we want to compute the value  $a^b$  where  $a$  and  $b$  are natural numbers stored in the memory words at addresses 100 and 104. A machine program to compute this value and store the result in address 108 basically looks as follows (the prefix “l:” denotes the address of the corresponding instruction in computer memory and the comment in *italics* is not part of the instruction):

```

0: cnst r0, 1    load constant 1 into register r0
4: load r1, 100 load memory word a into register r1
8: load r2, 104 load memory word b into register r2
12: cjmp r2, 0, 28 if r2 is 0, jump to instruction 28
16: mult r0, r1 multiply r0 with r1 and store result in r0
20: diff r2, 1 subtract 1 from r2 and store result in r2
24: jump 12 jump to instruction 12
28: stor 108, r0 store register r0 in memory word c

```

We look at the execution of the program which starts in a state where the program counter has value 0 and memory locations 100 and 104 have values 3 and 2, respectively. All other registers and memory location 108 have arbitrary values at the beginning.

step	pc	r0	r1	r2	100	104	108
0	0				3	2	
1	4	1			3	2	
2	8	1	3		3	2	
3	12	1	3	2	3	2	
4	16	1	3	2	3	2	
5	20	3	3	2	3	2	
6	24	3	3	1	3	2	
7	12	3	3	1	3	2	
8	16	3	3	1	3	2	
9	20	9	3	1	3	2	
10	24	9	3	0	3	2	
11	12	9	3	0	3	2	
12	28	9	3	0	3	2	9

We see that instructions 12 – 24 are iterated two times, in each iteration register r0 is multiplied with register r1 and register r2 is decreased by 1. After the second iteration, the result is stored at memory location 108.

---

## 3.2 Programming Languages

While the processor executes a machine program of binary-encoded instructions such as (see page 26)

17	1	0	100
----	---	---	-----

this is not how programs are written. Scientists and engineers invented already very early (in the 1950s) *programming languages* that allow to write programs in a much more comfortable way.

### 3.2.1 Assembly Language

A first step away from binary machine programs was the *assembler*, a program which allows you to use symbolic notation for the individual machine instructions. The human may write an assembly language program like

```

0: cnst r0, 1
4: load r1, 100
8: load r2, 104
12: cjmp r2, 0, 28
16: mult r0, r1
20: diff r2, 1
24: jump 12
28: stor 108, r1

```

and the assembler converts it into a sequence of machine instructions. However, the symbolic program still contains a description of each instruction of the corresponding machine program. Assemblers are used to write low-level code that has to access machine features and/or must be very efficient.

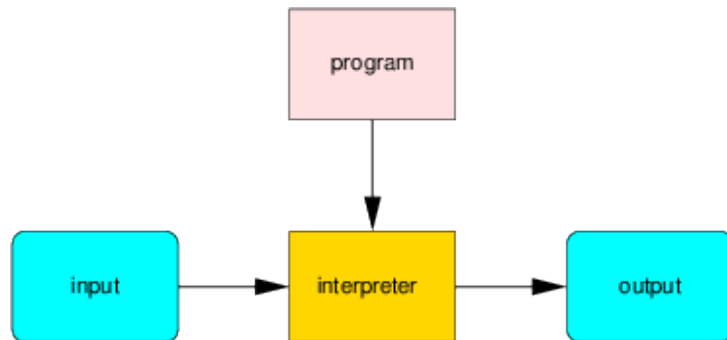


Figure 3.5: Interpretation

### 3.2.2 High-level Languages

In a high-level language, the machine program written in the previous section may be expressed as

```
int r = 1;
for (int i = 0; i < b; i++)
    r = r*a;
```

which is much shorter and easier to understand. High-level languages are (in roughly chronological order) like Fortran, Basic, Pascal, C, Modula-2, Simula, Ada, C++, Oberon, Java, etc.

However, now there is the problem how such a program can be executed, since the computer's processor only understands machine language. Basically, there exist two approaches.

#### Interpretation

We have an *interpreter*, i.e., a program that reads the high-level program and executes it. Like the processor interprets machine language instructions, this program interprets the instruction of the high-level language (see Figure 3.5).

The problem with this approach is that the interpretation overhead is very high, i.e., for each high-level instruction to be executed, the interpreter may have to perform say 100 machine instructions. Therefore the program runs 100 times slower than the corresponding machine program. The interpreter approach is therefore usually only used when performance is not a concern.

A processor is an interpreter for machine language.

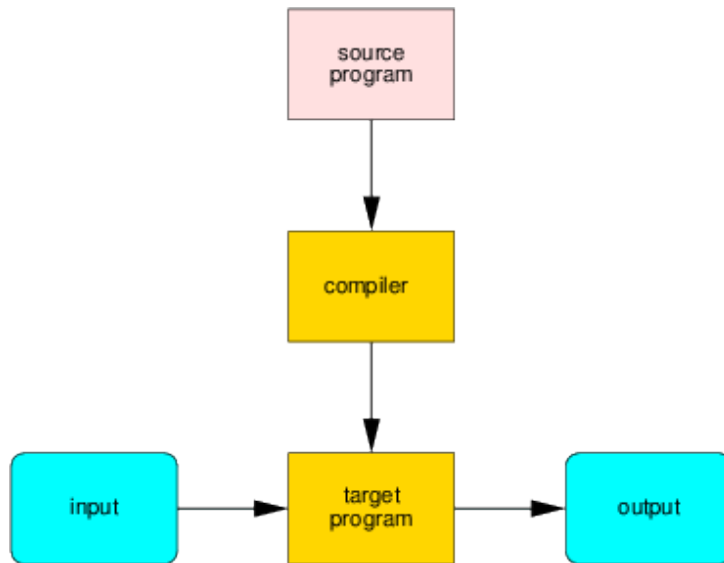


Figure 3.6: Compilation

---

## Compilation

We have a *compiler*, i.e., a program that reads the high-level program (the *source code*) and generates a corresponding machine program (the *target code*) from it (see Figure 3.6). Actually, the compiler usually generates an assembly language program from which an assembler creates the actual machine program.

This is the approach chosen by most programming languages. By highly sophisticated analysis and optimization techniques, the machine program generated by the compiler is usually *almost* as efficient (say not more than 30% slower) as if the user had written the machine program by hand.

Usually, a high-level program is not written as one monolithic piece of code but in multiple *modules*. Each module is an individual piece of the program that can be independently developed and compiled to an *object file* (a piece of code in machine language). The *linker* takes all object files and combines them to the overall executable program file. This two-step of compiling and linking is the basis of most high-level programming languages. Usually, the program uses pre-compiled object code *libraries* that contain functionality which is commonly used in many application programs. There exist standard libraries that are of general use but also application-specific libraries for e.g. writing programs for particular GUIs.

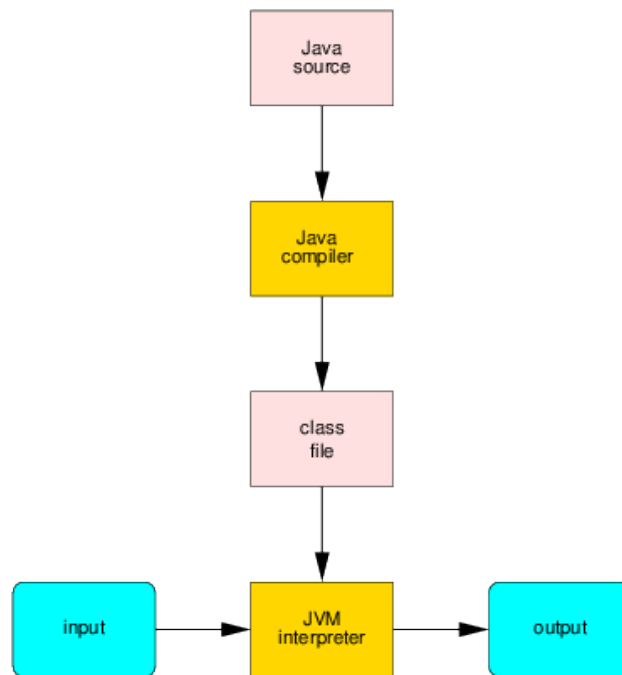


Figure 3.7: Java Compilation and Interpretation

Compiled programs have to be ported to different kinds of systems.

Since the generated machine program is specific to a particular brand of processor (such as e.g. the Intel Pentium line), the source code has to be re-compiled and re-linked for every platform where the program has to be executed. This process is cumbersome and error-prone because different systems may behave in subtly different ways (e.g. due to differences of the compiler or not quite compatible library versions). Nevertheless many programs run (almost) the same way on computers with different processors or even different operating systems. Programs that have this property are called *portable*, the process of adapting a program running on one machine to execution on another machine is called *porting*.

### 3.2.3 Java

Java is US slang for “coffee”.

Java<sup>4</sup> was developed in the 1990’s by a team of the computer company Sun and became very popular with the uprise of the World Wide Web and the use of the Internet as a global communication and collaboration platform. Java is further developed by the “Java Community Process” under the stewardship of the

<sup>4</sup><http://www.oracle.com/technetwork/java/index.html>

computer company Oracle (which bought Sun); while there are various commercial implementations, most of the Java platform has been released as open source software (OpenJDK<sup>5</sup>).

Java has inherited various concepts from predecessor languages such as C++, but it was developed with a specific idea in mind: *the target code should execute in the same way on all kinds of machines* (“write once, run everywhere”). To achieve this goal, the language developers introduced an abstraction layer between the Java source language and the computer system, the *Java Virtual Machine* (JVM).

The JVM is a hypothetical machine that executes as its machine language the JVM *byte-code* language. The byte-code language is sufficiently close to the machine language of real processors such that a JVM interpreter can execute this byte code with only little overhead (compared to interpreters for high-level languages). A Java compiler generates from Java source code a *class file*, which is an object file in the JVM byte-code language. All class files for an application are passed to the JVM that *dynamically* links the class files to the application program (i.e., no class file is generated for the whole program) and executes it (see Figure 3.7).

Actually, the JVM defines more than a processor architecture, it defines a whole execution environment including file input/output and a graphical user interface. The JVM byte code program does not directly access the operating system level (and thus the corresponding machine resources), it considers the JVM as its operating system to which it delegates all interaction with the outside world. Since this execution environment is the same for all implementations of the JVM on all kinds of machines, it is reasonable to expect that Java byte code programs will run the same way on all kinds of architectures.

To speed up the execution of JVM byte code programs, sophisticated optimization techniques were introduced, in particular the technique of *Just in Time (JIT)* compilation. A JIT-enabled interpreter dynamically compiles JVM byte code to machine code of the underlying processor and then executes this machine code rather than interpreting the byte code. With this and other developments, the performance overhead for the execution of Java programs has become very small compared to the execution of programs in other high-level languages (which are directly compiled to machine code). If your application is not extremely time-critical or system-oriented, Java therefore represents a reasonable implementation language.

While Java has been originally developed for providing active content within web pages via “applets”, it has fallen out of favor for that particular purpose, due to the fact that modern web browsers are themselves able to execute programs in

---

<sup>5</sup><http://openjdk.java.net>

the language “JavaScript” (syntactically similar but otherwise unrelated to Java) and due to numerous security holes in the Java browser plugin technology. Java’s main application is now the development of standalone programs, in particular client/server web applications, distributed systems, embedded systems, mobile phones, etc. It is one of the most popular general purpose programming languages used today.

# Chapter 4

## Java Light

We will now start with the basics of the Java programming language, version 18 (*Java 18*). These lecture notes are not a reference manual for Java; if you want to lookup details, you can browse the Java language specification respectively the Java Platform Standard Edition 18 API Specification.

Java programs can be developed with different programming environments, e.g. the Java Development Kit (JDK) (which is freely available<sup>1</sup> for various systems). For learning programming in Java, there is not much difference which environment you use. In these notes, we use the JDK commands when we give examples for compiling or running programs,

Historically, there have been two kinds of Java programs:

- *applications* which are standalone programs that can be independently executed, and
- *applets* which are embedded into Web pages and can be executed by viewing the page by a Java-enabled Web browser.

However, the applet technology is outdated and not any more supported by modern web browsers. Therefore we will deal in this course only with Java applications.

### 4.1 Java Applications

The source code of a Java application has the form

---

<sup>1</sup><https://www.oracle.com/technetwork/java/javase/downloads/index.html>

```
public class Name
{
    public static void main(String[] args)
    {
        ...
    }
}
```

We use teletype font to denote code which is to be taken verbatim and *italics* font to denote parts which can be substituted by the programmer.

The keyword `class` will be explained later, for the moment just use it as shown above.

This code implements an application called *Name* where *Name* is a Java *identifier*. Such an identifier is a word that may contain upper and lower case letters, decimal numbers and the underscore letter “\_” but must start with a letter or underscore. We have the convention to use program names with upper-case capital letters.

---

**Example** `Prog_2A` is a Java identifier, `2A_Prog` is not.

---

Every application must have a method `main`. A *method* is a piece of code (more details later), the `main` method is the method which is executed when the program is started. What the line `public static void main(String[] args)` exactly means will be explained later. The *body* of a class is everything between the curly braces `{ }` (the method `main` above), likewise we speak of the body of a method (the “...” above).

In JDK, we store the source code of an application *Name* in a file `Name.java` and call the Java compiler by typing on the command line

```
javac Name.java
```

This generates the class file `Name.class`. We can execute this file by invoking the JVM interpreter

```
java Name
```

---

**Example** As a first example, you may write in file `Example.java` the following program:

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

You can then compile the program by typing on the command line

```
javac Example.java
```

This generates a file `Example.class` which you can execute by typing

```
java Example
```

The execution of the program then prints to the screen

```
Hello, World!
```

---

---

## 4.2 Source Code Conventions

The format of the source code does not have any logical significance, we may write it as

```
public class Name {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

or as

```
public class  
Name  
{  
    public static void  
    main(String[] args)  
    {  
        ...  
    }  
}
```

or even as

```
public class Name { public static void main ( String [ ] args ) { ... } }
```

i.e., we may insert empty space and empty lines wherever we please. It is only the sequence of *tokens* (names and programming language constructs like `class` or `}`) that matters (therefore we must not insert space or break a line within a token). Ultimately, the style of source code formatting is therefore a matter of taste and subject to only one rule: whatever style you choose, you should stick to it consistently.

However, we must keep in mind that the source code of a program is intended for reading by not only ourselves; we should write it under the consideration that *other* people may have to understand our code. We therefore strongly recommend to use appropriate *indentation* to exhibit the logical structure of a program. Whenever a program element is contained within other elements (typically denoted by the opening brace `{`), we start a new line and indent the inner element by say 2 character positions. Whenever a line gets too long by indentation, we strive to break it into two lines.

Most editors automatically indent the source code.

Furthermore, to make understanding easier, a source file does not only contain code but also *comments* which are descriptions in natural language that are ignored by the compiler. In Java, comments start with the token `//` and extend to the end of line. We will stick to the convention that every class and every method has a comment header.

The program header essentially gives information about the development of the program; a method header gives information about the functionality of the method (we will discuss the documentation of methods later in more detail). The commented source code of an application might therefore look as follows:

```
// -----  
// Example.java  
// Show the general structure of a Java application  
//  
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@fhs-hagenberg.ac.at>  
// Created: August 18, 2001  
// Changed: August 19, 2001  
//   Changed the name of the class.  
// Changed: August 21, 2001  
//   Modified the comments.  
// -----  
public class Example  
{  
    // -----  
    // print a sample string to the standard output stream  
    // -----  
    public static void main(String[] args)  
    {  
        System.out.println("Hello, World!");  
    }  
}
```

```
}  
}
```

In order to make the source code readable for people with different editor settings, do not use more than 78 characters per line.

Typical editor settings are  $80 \pm 1$  characters.

Another issue is in which human language to write program identifiers and comments. In general it is better to assume that sometimes also foreigners may have to read your program. Therefore we recommend to write identifiers and comments not in your native language but in *English*.

## 4.3 Basic Text Output

The body of a method consists of a sequence of *statements* separated by the semicolon (;). Statements are executed in the sequence in which they occur in the method and have some effect which is either internal to the program or influences the external environment.

Two statements that have external effect are the output statements

```
System.out.print(string)  
System.out.println(string)
```

each of which prints a string (a sequence of characters) to the *standard output stream* (typically the terminal where you execute the program). The `println` (“print line”) statement starts after the string a new line while the `print` statement leaves the line unchanged. Thus the method

```
public static void main(String[] args)  
{  
    System.out.print("One, ");  
    System.out.print("two, ");  
    System.out.println("three.");  
    System.out.println("We are done.");  
}
```

prints the text

```
One, two, three.  
We are done.
```

We may not only print string literals enclosed by the double quote " but also, e.g. integer numbers as in

```
System.out.print("One plus one is ");
System.out.print(2);
System.out.println(".");
```

A literal is a constant which is textually included in the source code.

which gives the text

```
One plus one is 2.
```

The Java compiler here automatically converts the number 2 to the string "2". Java also allows to concatenate strings (append them together), such that we may write above code shorter as

```
System.out.println("One plus one is " + 2 + ".");
```

We will see later that other entities can be printed in the same way.

Since the character " is used to denote the limits of string literals, how can we print this character itself? We use for this purpose the *escape sequence* \" where the letter \ (the *escape character*) says that the next character " does not denote the end of the literal but just stands for itself. Therefore we may write

```
System.out.println("He said \"Hello\" and left.");
```

which prints

```
He said "Hello" and left.
```

Since the character \ is now special, we also have to quote it within string literals as in

```
System.out.println("C:\\windows\\system");
```

which prints

```
C:\windows\system
```

Other escape character are the \n character which starts a new line and the \t character which inserts a tabulator. Executing

```
System.out.println("He said:\n\tYes.\nShe said:\n\tNo.");
```

thus gives

```
He said:
    Yes.
She said:
    No.
```

## 4.4 Basic Graphical Output

For allowing simple graphical output, we provide a small framework whose operations are described in file `Drawing.README`. This framework is contained in a file `kwm.jar`; to use it just download this file from the course site and place it in the directory with the source code of your program. The framework can then be used by a program of the following form:

```
import kwm.*;
import java.awt.*;

public class Name
{
    public static void main(String[] args)
    {
        Drawing.begin("title", width, height);
        Drawing.graphics.operation(...);
        ...
        Drawing.end();
        ...
    }
}
```

The program has the usual structure except for the two `import` declarations at the beginning; these are not necessary to understand the following descriptions.

The `main` method executes an operation `Drawing.begin(...)` which opens a graphical window with the denoted *title* (a string), *width* and *height* (positive integer numbers); when the `Drawing.end()` operation is executed, a message

```
Press <ENTER> to continue or close window to quit...
```

appears on the output terminal. If you close the window, the program terminates; if you hit the “ENTER” key, the program continues with the statements after the `Drawing.end()`.

Between `Drawing.begin(...)` and `Drawing.end()` there may be arbitrarily many operations of form `Drawing.graphics.operation(...)` where the command `operation(...)` denotes one of the following graphical output operations (there are also others, a complete list can be found on the Java site<sup>2</sup>):

---

<sup>2</sup><http://docs.oracle.com/javase/8/docs/api/java/awt/Graphics.html>

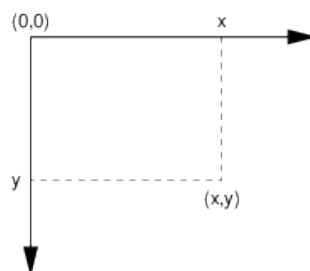


Figure 4.1: Java Coordinate System

Operation	Description
<code>drawRect(x, y, w, h)</code>	draw rectangle $[x \dots x + w, y \dots y + h]$
<code>drawOval(x, y, w, h)</code>	draw oval $[x \dots x + w, y \dots y + h]$
<code>drawLine(x1, y1, x2, y2)</code>	draw line from $(x1, y1)$ to $(x2, y2)$
<code>drawString(s, x, y)</code>	draw string $s$ at $(x, y)$
<code>fillRect(x, y, w, h)</code>	fill rectangle $[x \dots x + w, y \dots y + h]$
<code>fillOval(x, y, w, h)</code>	fill oval $[x \dots x + w, y \dots y + h]$
<code>setColor(c)</code>	set drawing color to $c$

The graphical operations take place in a coordinate system where the top-left corner of the drawing area has coordinate  $(0, 0)$ , the first coordinate grows towards the right and the second grows towards the bottom (see Figure 4.1).

Coordinates are given in non-negative integer numbers denoting screen pixels, for instance the operation

```
Drawing.graphics.fillRect (60, 80, 225, 30);
```

draws a rectangle whose left upper corner is 60 pixels to the right and 80 pixels down and that extends 225 pixels to the right and 30 pixels down (all coordinates relative to the left upper corner of the painting area). The colors used in `setColor` can be taken from a set of predefined colors like `Color.red`, `Color.blue`, ...

### Example

An example of the main method of a program called `Picture` demonstrates the use of these calls:

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

```
Drawing.begin("Picture", 300, 200);

Drawing.graphics.setColor(Color.green);          // filled shapes
Drawing.graphics.fillRect (60, 80, 225, 30);    // rectangle
Drawing.graphics.fillOval (150, 120, 200, 150); // oval

Drawing.graphics.setColor(Color.red);           // non-filled shapes
Drawing.graphics.drawRect (50, 50, 40, 40);    // square
Drawing.graphics.drawOval (75, 65, 20, 20);    // circle
Drawing.graphics.drawLine (35, 60, 100, 120);  // line

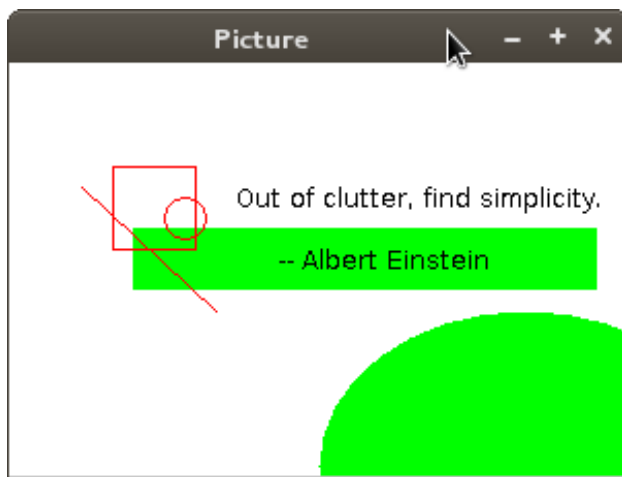
Drawing.graphics.setColor(Color.black);        // text
Drawing.graphics.drawString ("Out of clutter, find simplicity.", 110, 70);
Drawing.graphics.drawString ("-- Albert Einstein", 130, 100);

Drawing.end();
}
```

Compiling this program yields file `Drawing.class`. If we then start the program as

```
java Picture
```

the following window pops up.



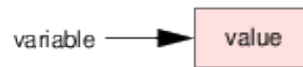


Figure 4.2: A Variable

## 4.5 Variables and Assignments

The data managed in a program are represented by program variables. A *variable* is a name for a memory address at which a data value is stored. In a more abstract way, can think of variables simply as the names of containers or boxes that contain values (see Figure 4.2).

### 4.5.1 Variable Declarations

A *variable declaration* is a program statement which instructs the compiler

1. to reserve a portion of memory space to hold a particular type of value,
2. to initialize this portion with a particular value,
3. and to refer further on to this portion by a particular name.

As an example, the program

```
String name = "Markus";  
int age = 21;  
System.out.println(name + " is " + age + " years old.");
```

We will see later why `int` starts with a lower-case letter while `String` starts with an upper-case letter.

declares two variables *name* and *age* of type `String` and `int` and initializes these variables with the values “Markus” and 21. The `println` statement may now refer to these variables and print

```
Markus is 21 years old.
```

The general format of a variable declaration is

```
type identifier = value;
```

where

- *type* denotes the kind of data that the variable may hold, e.g. `int` for integer numbers and `String` for character strings;
- *identifier* is the name of the variable which may be used from the point of the declaration to the end of the method;
- *value* is the initial value of the variable.

It is an error to declare a variable twice (in the same scope, see later).

### 4.5.2 Assignments

The value of a variable is not fixed but may be changed by an *assignment* statement of the form

```
identifier = value;
```

This statement instructs the compiler to write the value denoted by the right hand side of the assignment to the variable identified by the left hand side. It is read as “*identifier* becomes *value*”.

“Varying” means “changing”.

For instance, writing

```
String name = "Markus";  
int age = 21;  
System.out.println(name + " is " + age + " years old.");  
name = "Michaela";  
age = 23;  
System.out.println(name + " is " + age + " years old.");
```

prints

```
Markus is 21 years old.  
Michaela is 23 years old.
```

The assignment

```
x = x+1;
```

takes an integer variable  $x$  and overwrites its old value with the new value  $x + 1$ .

This is not saying that  $x$  equals  $x + 1$ !

It is an error to try to assign to a variable a value which is different from the variable type:

```
int age = 21;
age = "twentyone";
```

The compiler detects this error and reports

```
Main.java:6: incompatible types
found   : java.lang.String
required: int
    age = "twentyone";
        ^
```

(where 6 is the line number of the assignment in the source code).

If a variable is assigned a value before its first use, we may omit the initialization part of a variable declaration. For instance, the program

```
int age;
age = 21;
System.out.println(age);
```

prints

```
21
```

The compiler will report an error, if an uninitialized variable is used before it is assigned a value.

### 4.5.3 Constant Declarations

A declaration of the form

```
final type identifier = value;
```

declares *identifier* as a *constant*; the value of a constant can not be changed after the declaration any more (no assignment is possible).

“Constant” means “not changing”.

If you have any special values in your program, it is good to refer to them by constants, e.g.

```
final int MAX_VALUE = 127;
final String SYS_DIR = "c:\\windows";
```

To differentiate between variables and constants, constant names are sometimes written in upper case letters.

### 4.5.4 Choosing Variable Names

We use variable names with lower-case capital letters. The name of the variable should denote the object it holds; it is therefore typically a self-explaining noun like

```
int temperature = 17;
```

or a combination of words as in

```
int maximumTemperature = 17;  
String directoryName = "c:\\windows";  
final int maximumWindowHeight = 640;
```

where the upper-case letter indicates the beginning of the next word.

Variable names with more than say 12 letters get very cumbersome; most people prefer to abbreviate them like in

```
int maxTemp = 17;  
String dirName = "c:\\windows";  
final int maxWinHeight = 640;
```

However, do *not* use for variable names mysterious acronyms like

```
int mt = 17;  
String dn = "c:\\windows";  
final int MWH = 640;
```

which make programs very hard to read.

Usually it is a good idea to annotate a variable declaration by a comment that explains the intended use of the variable across all assignments

```
int temp = 0;    // temperature taken in last measurement  
int maxTemp = 0; // maximum temperature of all measurements  
int avgTemp = 0; // average temperature of all measurements
```

such that we can quickly get the general idea.

## 4.6 Primitive Data Types and Expressions

Java has various *primitive data types* which correspond to the datatypes built into the Java Virtual Machine. All primitive datatypes have names that start with a small initial letter; the datatype `String` is actually not primitive, but will be also explained in this section for the sake of convenience.

### 4.6.1 Booleans

The Java datatype `boolean` consists of two constants `true` and `false`. The program

```
boolean b = true;
System.out.println(b);
```

prints

```
true
```

We have the following boolean operations (see the accompanying mathematics course for details):

In mathematics, boolean operators are called logical *connectives*.

Operator	Description
!	“not” (negation)
&&	“and” (conjunction)
	“or” (disjunction)

For instance, the program

```
System.out.println(!true);
System.out.println(true && false);
System.out.println(true || false);
```

prints

```
false
false
true
```

The usual precedence rules hold, i.e., `!` binds stronger than `&&` which binds stronger than `||`. Therefore we can write

```
!a && b || c
```

to denote `((!a) && b) || c`. Nevertheless we recommend to use parentheses to make such structures more readable, e.g. as in

```
(!a && b) || c
```

The operations `&&` and `||` are *short-circuited*: whenever the first argument is sufficient to determine the result, the second argument will not be evaluated, i.e., `false && any` yields `false` and `true || any` yields `true`. The relevance of this will become clear later.

We have in Java on *all* datatypes the following *equality operations*:

Operator	Description
<code>==</code>	equal to
<code>!=</code>	not equal to

For instance, the program

```
boolean b = (1 == 0);
System.out.println(b);
```

prints

```
false
```

## 4.6.2 Integers

By “integers”, we denote various datatypes that hold integral numerical values in a certain range. In Java, we have four integer types

Type	Size	Minimum Value	Maximum Value
<code>byte</code>	8 bits	-128	127
<code>short</code>	16 bits	-32768	32767
<code>int</code>	32 bits	-2147483648	2147483647
<code>long</code>	64 bits	-9223372036854775808	9223372036854775807

where `int` is the integer type used for most purposes. We refer to the minimum and maximum values of integer datatypes also by the constants

Type	Minimum Value	Maximum Value
<code>byte</code>	<code>Byte.MIN_VALUE</code>	<code>Byte.MAX_VALUE</code>
<code>short</code>	<code>Short.MIN_VALUE</code>	<code>Short.MAX_VALUE</code>
<code>int</code>	<code>Integer.MIN_VALUE</code>	<code>Integer.MAX_VALUE</code>
<code>long</code>	<code>Long.MIN_VALUE</code>	<code>Long.MAX_VALUE</code>

No, I don't know why Integer and not Int.

## Arithmetic Operations

An *integer literal* is formed as a sequence of decimal digits like `0`, `9`, `47113414`. Literals larger than `Integer.MAX_VALUE` denote values of type `long` and need a suffix `L`, e.g.

```
long age2 = 49203281234L;
```

We have on the integer types the operations

Operator	Description
<code>+</code>	unary plus and addition
<code>-</code>	unary minus and subtraction
<code>*</code>	multiplication
<code>/</code>	truncated division
<code>%</code>	remainder

For example, we can write

```
int a = 2;
int b = 3;
a = ((a+b)*7)/a;
System.out.println(a);
```

and get the output

```
17
```

Division by zero results in a runtime error, for instance when executing

```
int a = 1/0;
```

the JVM interpreter aborts with the message

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Main.main(Main.java:5)
```

Division truncates towards zero ( $-17/2 = -8$ ), the remainder satisfies the equation  $a = (a/b) * b + (a\%b)$ .

So, what is  $-17\%2$ ?

The usual precedence rules hold for operators (`*` and `/` bind stronger than `+` and `-`), so we may write

$a*b+a/b$ 

rather than

 $(a*b)+(a/b)$ 

There are also rules that tell us whether

 $a*b/c$ 

is to be interpreted as

 $(a*b)/c$ 

or as

 $a*(b/c)$ 

which makes a difference, e.g.  $(2 * 3)/2 = 3$  but  $2 * (3/2) = 1$ . However, most people do not remember these rules; you better make extra parentheses to make clear what you want.

Integer arithmetic is tricky!

## Overflows

It is important to keep in mind that the Java integer types only represent a small subset of the set  $\mathbb{Z}$  of mathematical integers. While in mathematics the addition of two positive integers gives a positive result, this is not necessarily true for the bounded integers of Java.

---

**Example** Take the program

```
int a = 1500000000;  
int b = 1000000000;  
int c = a+b;  
System.out.println(c);
```

which prints

-1794967296

While the initial values of both  $a$  and  $b$  are less than `Integer.MAX_VALUE`, their sum 2500000000 is greater than this bound and can therefore not be represented by a positive `int` value any more. An *integer overflow* happens where the result “wraps” back into the negative range, i.e., we get as the result

$$\begin{aligned} & (1500000000 + 1000000000) - \text{MAX\_VALUE} + (\text{MIN\_VALUE} - 1) \\ & = 2500000000 - 2147483647 + (-2147483648 - 1) \\ & = 352516353 - 2147483649 \\ & = -1794967296. \end{aligned}$$

---

---

An integer overflow is not an error which is reported by the compiler or the interpreter. We have to make sure that these situations cannot happen by choosing appropriate ranges for the results of arithmetic operations.

### Widening Conversions

When a Java program computes with variables of types `byte` or `short`, their values are automatically converted to type `int` before the arithmetic operation is performed (*widening conversion*). For instance, the program

In arithmetic operations, `byte` and `short` are promoted to `int`.

```
byte a = 100;
byte b = 100;
System.out.println(a+b);
```

prints

```
200
```

rather than `-56`, because the addition is performed on `int` values rather than with `byte` values. Because of this *arithmetic promotion*, we can mix `byte`, `short`, and `int` values; all arithmetic will be performed on `int`. For instance,

```
byte a = 127;
short b = 255;
int c = a*b+1;
System.out.println(c);
```

prints

```
32386
```

Likewise, if an arithmetic operation is performed with an `int` value and a `long` value, the `int` value is first converted to `long` before the result is computed.

```
int a = Integer.MAX_VALUE;
long b = Integer.MAX_VALUE;
System.out.println(a+b);
```

prints

```
4294967294
```

In arithmetic operations with `long`, `int` is promoted to `long`.

Generally in any assignment from a smaller type to a larger type, the smaller type is promoted to the larger type. The program

```
byte a = 127;
short b = a;
int c = b;
long d = c;
System.out.println(d);
```

compiles without problem and prints

```
127
```

In assignments, the smaller value type is promoted to the larger variable type.

### Narrowing Conversions

While widening conversions are automatically performed, this is not true for the other direction. Given a program

```
int a = 127;
short b = a;
```

the compiler complains

```
Main.java:6: possible loss of precision
found   : int
required: short
    short b = a;
           ^
```

The compiler does not detect that *a* holds value 1 which actually could be represented as a `short`

because the `int` value of *a* *might* be too large to be stored in the `short` variable *b*. Even on the program

```
short a = 1;
short b = a+1;
```

the compiler complains

```
Main.java:6: possible loss of precision
found   : int
required: short
    short b = a+1;
           ^
```

because the computation of  $a + 1$  is performed on `int` values, i.e., the result is also an `int`.

In order to perform a *narrowing conversion* from a wider datatype to a shorter datatype, we have to explicitly write

Explicit type casts are needed for narrowing conversions.

```
int a = 127;
short b = (short)a;
```

The construct (*type*) is a *type cast* which converts the value of the expression (possibly a wider integer type) to the denoted type (possibly a more narrow type). The system will not check for integer overflows but wraps too large values back into the range of the result type. For instance, the program

```
short a = 128;
byte b = (byte)a;
System.out.println(b);
```

prints

```
-128
```

Please note the setting of parenthesis in the following type cast

```
short a = 1;
short b = (short)(a+1);
```

because writing `(short)a+1` would apply the typecast just to variable `a` which would be useless.

Why?

## Relational Operations

We have on all integer types (in addition to the equality operations) the following *relational operations*:

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

The same promotion rules apply as for arithmetic operations. For instance, the program

```
short i = -1;
int j = 1;
boolean b = (i <= j);
System.out.println(b);
```

compiles correctly and prints

```
true
```

Also the program

```
boolean b = (-2 <= 1) || (1/0 == 1);
System.out.println(b);
```

runs correctly and prints

```
true
```

This is because the operator `||` is short-circuited (see page 47): the first argument evaluates to true and therefore the second argument is not evaluated (which would result in a division by zero error, see page 48).

Compound boolean expressions are evaluated from left to right.

### 4.6.3 Characters

In Java, characters are provided by the datatype `char` in Unicode representation (see page 22), i.e., each character takes two bytes of memory. A *character literal* is enclosed in single quoted, for instance in

```
char letter = 'a';
char newline = '\n';
char backslash = '\\';
System.out.print(letter);
System.out.print(newline);
System.out.println(backslash);
```

which prints

```
a
\
```

In addition to the escape sequences already discussed (see page 38), we have the sequence `\'` to denote the single quote character. The escape sequence `\unnnn` denotes the Unicode code of a character in the hexadecimal system, e.g.

```
char letter = '\u0041';
System.out.println(letter);
```

prints

```
A
```

Actually, Java considers the datatype `char` just as an integer type smaller than `int` and allows to use values of this type in the same way as e.g. `short` values may be used. For instance, we may initialize a character as in

```
char letter = 65;
System.out.println(letter);
```

`char` is an integer type.

which prints

```
A
```

We may compare characters as in

```
boolean b = ('A' <= 'B')
System.out.println(b);
```

which yields

```
true
```

We have automatic promotion from `char` to `int` and may cast `int` values to `char` values:

```
int i = 'A'+1;
System.out.println((char)i);
```

which yields

```
B
```

Since the characters inherit their ordering properties from their Unicode values (see page 21), the view of characters as integers becomes handy in text processing.

#### 4.6.4 Floats

*Floating point* types or *floats* are used to represent fractional values. In contrast to integers, floating point values are only *approximated* and *rounding errors* may (and will!) occur in computations. The term “floating point” stems from the fact that numbers have not fixed precision (“fixed point”) but that the precision varies with the size of the number. In particular, fractions close to zero are approximated with higher accuracy than large fractional values.

Floating point types only approximate fractions.

In Java, we have the following floating point types:

Type	Size	Minimum Value	Maximum Value	Significant digits
float	32 bits	ca. $-3.4 * 10^{38}$	ca. $3.4 * 10^{38}$	7
float	64 bits	ca. $-1.7 * 10^{308}$	ca. $3.4 * 10^{308}$	15

Type `double` can therefore represent not only a much larger range, it also represents fractions with greater accuracy.

A floating point literal can be denoted by a sequence of decimal numbers with a dot to indicate the fractional part:

```
float f = 3.1415927F;
double g = 2.718281828459;
```

Literals of type `float` have to be marked by an appended `F`, otherwise the literal is assumed to be of type `double`. An optional appendix  $En$  (where  $n$  is an integer literal) indicates multiplication of the fraction with  $10^n$ :

```
float f = 3.1415927E3F; // = 3141.5927F
double g = 2.718281828459E-3; // = 0.002718281828459
```

A floating point number is represented in computer memory by a triple  $(s, m, e)$  where

- the *sign* bit  $s$  denotes  $+1$  or  $-1$ ,
- the *mantissa*  $m$  is a  $n$  bit binary number representing the value  $m/2^n * 2$  (which is a rational number less than 1),
- the *exponent*  $e$  is a binary number.

The value represented by this triple is  $s * m/2^n * 2^e$ .

---

**Example** Assume we have an eight bit floating point number `0|01011|10` where

- the first bit `0` represents the sign  $+1$ ,
- the five bit mantissa `01011` represents the fraction  $11/32$  (why?),
- the two bit exponent is  $2$ .

The value represented by this floating point number is  $+1 * 11/32 * 2^2 = 1.375$ .

---

## Arithmetic Operations

We can print floating point numbers as in

```
float f = 1.5F;
System.out.println(f);
```

which gives

1.5

We have on the floating point types the operations

Operator	Description
+	unary plus and addition
-	unary minus and subtraction
*	multiplication
/	truncated division
%	remainder

The remainder satisfies the equation

$$a = (\text{int})(a/b) * b + (a\%b).$$

In floating point computations, the result value may become too big to be represented by a normal floating point number; the result is then denoted by the special value `Infinity` as in

So, what is `2.8%1.5`?

```
double f = 2.718281828459E300;
double g = f*f;
System.out.println(g);
```

which prints

```
Infinity
```

(likewise a computation with negative values may yield `-Infinity`). Floating point division by zero does not trigger a runtime error but yields `Infinity` or `-Infinity` (depending on the sign of the first argument). The floating point division `0.0/0` gives the special value “Not a Number” printed as `NaN`.

## Conversions

In an arithmetic operation with a `float` and a `double` argument, the `float` argument is automatically promoted to `double`, likewise in an assignment of a `float` value to a `double` variable.

In operations with mixed integer and floating point values, the integer values are promoted to floating point values, for instance

```
float f = 1.5F;
long l = 1;
System.out.println(f+l);
```

prints

2.5

In arithmetic operations with double, float is promoted to double.

To narrow a double value to a float value or to truncate an floating point value to an integer value, we have to perform an explicit type cast. For instance, the program

```
double f = 2.718281828459;
float g = (float)f;
int h = (int)f;
System.out.println(f);
System.out.println(g);
System.out.println(h);
```

prints

2.718281828459  
2.7182817  
2

### Pitfalls

Floating point numbers are very tricky beasts as shown by the following example: we compute two times  $f^{13} * g$ , one time ending the multiplication with  $g$ , one time starting the multiplication with  $g$ .

```
float f = 3.141592F;
float g = 3.141593F;
System.out.println(f*f*f*f*f*f*f*f*f*f*f*f*f*f*f*g);
System.out.println(g*f*f*f*f*f*f*f*f*f*f*f*f*f*f*f);
```

Executing the program gives output

9122149.0  
9122147.0

i.e., two different results in the *integer* part. The reason is that due to rounding errors simple arithmetic laws like associativity ( $a * (b * c) = (a * b) * c$ ) do *not* hold for floating point numbers and small rounding errors can after a few steps can accumulate to large overall errors.

Since floating point computation is also much slower than integer computation, the morale of the story is: use floating point numbers only when you need them; if you need them, handle them with care.

### Relational Operations

Like for integer types, we have on the floating point types the following *relational operations* which return a boolean value:

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

with the same promotion rules as for arithmetic operations. For instance, the program

```
int i = -1;
double d = 1.5;
boolean b = (i <= d);
System.out.println(b);
```

compiles correctly prints

```
true
```

Please note that it almost never makes sense to compare two floating point numbers directly ( $f = g$ ). This test only succeeds if both numbers are *bit-wise* equal, i.e., even the smallest rounding error in the computation of one of the arguments makes this comparison fail. It is much better to test whether both numbers differ only by a small tolerance value ( $|f - g| < e$  for say  $e = 0.0001$ ). Even this is not sufficient for scientific programs because in the comparison of large numbers also the tolerance should be larger, i.e., not an absolute tolerance but a relative tolerance is used. However, we will not deal with this issue further in this course.

### 4.6.5 Strings

The datatype `String` provides sequences of characters which are internally represented by their Unicode numbers (see page 22), i.e., each character takes two bytes of memory. The default value of `String` is `null` which denotes “no string” (which is different from the “empty string”, see below).

A *string literal* is enclosed in double quotes (in contrast to the previously encountered character literals which are enclosed in single quotes). The program code

```
String letter = "a";
String empty = "";
String word = "hello";
String sentence = "The ball is red.";
System.out.println(letter);
System.out.println(empty);
System.out.println(word);
System.out.println(sentence);
```

thus prints

```
a

hello
The ball is red.
```

Since in a string literal enclosed by double quotes, no double quote may appear as a plain character between the quotes. For this reason, we have the escape sequences (see page 38) `\"` to denote the double quote character. We may thus write the program

```
var doubleQuote: String = "\"";
System.out.println(doubleQuote);
```

which prints

```
"
```

The escape sequence `\unnnn` denotes the Unicode code of a character in the hexadecimal system, e.g.

```
var letter: String = '\u0041';
System.out.println(letter);
```

prints

```
A
```

### String Operations

The term  $s1+s2$  denotes the concatenation of strings  $s1$  and  $s2$ , i.e. that string that consists of the characters of  $s1$  followed by the characters of  $s2$ . The program

```
String s1 = "abc";
String s2 = "def";
String s = s1+s2;
System.out.println(s);
```

thus prints

```
abcdef
```

The term  $s.length()$  (note the parentheses) denotes the number of characters in string  $s$ . Thus the program

```
String s = "abc";
int n = s.length();
System.out.println(n);
```

prints

```
3
```

The term  $s.charAt(i)$  denotes the character at position  $i$  of string  $s$  (position counting starts at 0). The program

```
String s = "abcd";
char ch = s.charAt(2);
System.out.println(ch);
```

thus prints

```
c
```

The term  $s.codePointAt(i)$  denotes the Unicode value of the character at position  $i$  of  $s$ . The program

```
String s = "abcd";
int c = s.codePointAt(2);
System.out.println(c);
```

thus prints

```
99
```

(the Unicode value of “c” is 99).

The term `s.substring(i, j)` denotes that substring of `s` that starts at position `i` and ends *before* position `j`. The program

```
String s = "abcdefg";
String t = s.substring(2,5);
System.out.println(t);
```

thus prints

```
cde
```

If the parameter `j` is omitted, then the substring starting at position `i` and running till the end of the string is denoted, for instance

```
String s = "abcdefg";
String t = s.substring(2);
System.out.println(t);
```

prints

```
cdefg
```

The terms `s1.indexOf(s2)` and `s1.lastIndexOf(s2)` denote the position of the first respectively last occurrence of string `s2` in `s1`. If `s2` does not occur in `s1`, the result is `-1`. The program

```
String s = "abcabc";
System.out.println(s.indexOf("bc"));
System.out.println(s.lastIndexOf("bc"));
```

thus prints

```
1
4
```

respectively.

Correspondingly, the terms `s1.indexOf(s2, i)` and `s1.lastIndexOf(s2, i)` search for the first/last occurrence of string `s2` in string `s1`; the additional parameter `i` specifies the position where to “start” the search, e.g. the program

```
String s = "abcabc";
System.out.println(s.indexOf("bc", 2));
System.out.println(s.lastIndexOf("bc", 3));
```

prints

```
4
1
```

### String Comparisons

Unlike other values, we must not compare strings with the builtin equality operations `==` and `!=` (this is technically possible but may give unexpected results, e.g. `"abcdef" == "abc" + "def"` may *not* hold).

The correct way to compare two strings for equality is with the operation `s1.equals(s2)` which returns `true` if and only if both strings `s1` and `s2` have the same length and the same character in every position. The program

```
String s = "abcdef";
System.out.println(s.equals("abc" + "def"));
```

thus prints

```
true
```

We compare strings lexicographically by the operations `s1.compareTo(s2)` and `s1.compareToIgnoreCase(s2)`, where the second one treats lower case and upper case characters as the same. The result of these operations is an integer value which is 0, if both strings are equal, negative, if `s1` comes lexicographically before `s2` and positive, if `s1` comes lexicographically after `s2`. “Lexicographical” ordering means that the Unicode values of all characters are compared from left to right until the first difference is found (which then determines the result) or if one string ends (in this case, the shorter string is considered as the smaller one). The program

```
String s = "AB";
System.out.println(s.compareTo("B") < 0);
System.out.println(s.compareTo("AC") < 0);
System.out.println(s.compareTo("ABC") < 0);
```

thus prints

```
true
true
true
```

When comparing strings of letters, the comparison thus corresponds to the well-known “phone book ordering”.

## 4.7 Basic Input

Java does not provide facilities for the input of basic datatypes that are so convenient to use as the method for `System.out.println()`. We therefore provide our own simple input methods whose description is stored in file `Input.README`. Like the previously explained graphical operations, these operations are also contained in the file `kwm.jar` that you can download from the course site and place into the directory of your own program.

We have the following input methods:

Method	Description
<code>Input.readBoolean()</code>	reads boolean value
<code>Input.readByte()</code>	reads byte value
<code>Input.readShort()</code>	reads short value
<code>Input.readInt()</code>	reads int value
<code>Input.readLong()</code>	reads long value
<code>Input.readChar()</code>	reads char value
<code>Input.readFloat()</code>	reads float value
<code>Input.readDouble()</code>	reads double value
<code>Input.readString()</code>	reads String value

Each of these operations reads one line of input from the *standard input stream* (typically the terminal where you execute the program), converts the content to the specified data, and returns the result. Thus the program

```

System.out.print("Enter first float: ");
float f1 = Input.readFloat();
System.out.print("Enter second float: ");
float f2 = Input.readFloat();
System.out.print("The product of " + f1 + " and " + f2 + " is ");
System.out.println(f1*f2);

```

gives rise to the following dialogue (where the input provided by the user is written in *italics*):

```

Enter first float: 2.7
Enter second float: -1.4
The product of 2.7 and -1.4 is -3.78

```

### 4.7.1 Error Handling

We have to consider the possibility that a line read from the input stream does not match the format expected for the datatype or that the input stream has been terminated (e.g., if the user has pressed the key combination CTRL+D). We therefore have four more methods:

Method	Description
<code>Input.isOkay()</code>	was last input okay?
<code>Input.hasEnded()</code>	has input stream ended?
<code>Input.getError()</code>	get message of last error
<code>Input.clearError()</code>	clear error flag and message

The idea is that after the use of an input operation the user calls the method `Input.isOkay()` to determine whether the value returned is indeed valid. If yes, we may further process this value. If not, we call `Input.hasEnded()` to determine whether the input stream has ended. If this function returns `false`, we have had an input error, probably the user has entered the data in an invalid format. A call of `Input.getError()` then returns a string describing the error; before we perform another read operation, we have to call `Input.clearError()` which clears the error and the associated message.

The use of these methods will be explained in the following sections.

## 4.8 Control Structures

The *control flow* of a program is the order in which its statements are executed. The default control flow is the sequential execution of statements from the first

statement of a method to the last. We will now discuss program constructs that modify the control flow. They basically fall in two categories:

- A *conditional statement* selects, depending on a condition determined at runtime, a statement for execution from a set of possible candidates. The control flow is therefore split into multiple alternative paths.

Java has the conditional statements `if`, `if-else`, and `switch`.

- An *iteration statement* repeats a statement until a particular condition (the *termination condition*) is satisfied. The control flow therefore forms a *loop* that may be traversed multiple times. It is the responsibility of the programmer to make sure that the termination condition eventually becomes true such that the loop terminates. Otherwise, we have a *infinite* loop, i.e., a non-terminating program.

Java has the iteration statements `while`, `do`, and `for`.

In addition, we have the *block statement* `{}` and the control flow statements `break` and `continue`. To describe the effects of control structures, we will make use of *assertions*.

### 4.8.1 Assertions

An *assertion* is a statement

```
assert boolean_expression;
```

which indicates our conviction that, whenever the control flow of the program reaches the position of the assertion, the boolean expression evaluates to true. A simple assertion is

```
var = exp;  
assert var == exp;
```

(provided that *exp* does not contain a reference to *var*).

---

#### Example

```
x = 1;   assert x == 1;  
x = a+b; assert x == a+b;  
x = x+1; // x == x+1 does not hold!
```

---

---

The use of assertions is an important technique to detect program errors at runtime. The idea is to insert at critical parts of the program checks for boolean conditions. If such a condition is true, execution continues in the normal way. If the condition is false (“the assertion fails”), we have found an error in the program (or our expectations of it); the execution of the program is therefore aborted with a corresponding message.

However, by default assertion checking is *disabled* at runtime, i.e., above statement is silently ignored. Only if you compile your program with the option `-ea` (short for `-enableassertions`)

```
javac -ea Program.java
```

assertion checking is switched on. If the check fails, the program aborts with an `AssertionError` message indicating the number of the source line where the assertion failed. If you want a more expressive failure message, you may use the expanded form of the assertion statement:

```
assert boolean_expression : any_expression;
```

In this case, the string representation of the value of the second expression is used as the error’s detail message.

## 4.8.2 The Block Statement

A *block statement* is the grouping of multiple statements into one by the use of curly braces:

```
{  
    statement;  
    statement;  
    ...  
}
```

A block statement can appear wherever a single statement can appear; this will get useful for the constructs introduced in the following sections.

If a variable declaration appears within a block statement, the declared variable can be only used from the point of the declaration to the end of the block. If the

same variable name is declared in different blocks, it denotes different variables that do not interfere with each other.

---

**Example** The program

```
{
  int i = 1;
  System.out.println(i);
}
{
  int i = 2;
  System.out.println(i);
}
```

compiles without errors and prints on execution

```
1
2
```

Given the program

```
{
  int i = 1;
  System.out.println(i);
}
System.out.println(i);
```

the compiler complains

```
Main.java:9: cannot resolve symbol
symbol   : variable i
location: class Main
  System.out.println(i);
                    ^
```

---

---

It is good program design to declare variables only in the blocks where they are actually used at those position where they are needed for the first time.

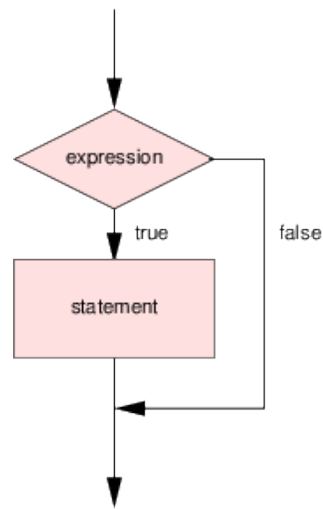


Figure 4.3: The `if` Statement.

---

### 4.8.3 The `if` Conditional

The *if* statement

```
if (boolean expression)
    statement
```

first evaluates the boolean expression and then, if the result is true, executes the statement. If the result is false, the statement is not executed. In both cases, execution continues with the statement after the conditional. The control flow of an `if` statement is illustrated in Figure 4.3.

---

**Example** Take the program

```
if (count < max)
    count = count+1;
```

If the variable *count* has value 5 and *max* has value 6, then the assignment is executed and *count* receives value 6. If both *count* and *max* have value 6, the assignment is not executed and the variable values remain unchanged.

---

---

In general, an `if` statement ranges over a block of statements.

---

**Example** Take the program

```
if (count < max)
{
    count = count+1;
    sum = sum+count;
}
```

If *count* is less than *max*, both assignments are executed. Otherwise, none of the assignments is executed.

---

---

### Assertions

When we enter the branch of an *if* statement, we can be sure that the condition we have just checked holds. We can therefore insert the assertion

```
if (expression)
{
    assert expression;
    ...
}
```

Furthermore, if an assertion holds before an assignment, we can restate the assertion after the assignment with the variables replaced by the values they had before the assignment.

---

**Example** We can annotate the program of the last example with the following assertions:

```
if (count < max)
{
    assert count < max;    // if branch
    count = count+1;
    assert count-1 < max; // count-1 = old count
    sum = sum+count;
}
```

---

---

If an assertion holds before an *if* statement, the same assertion holds also when we enter the branch of the statement. We can then combine multiple assertions with the **&&** operator.

---

**Example** We may annotate the nested *if* statement

```
if (sum < max)
{
    if (count < max)
    {
        count = count+1;
        sum = sum+count;
    }
}
```

as follows:

```
if (sum < max)
{
    assert sum < max;                // first if branch
    if (count < max)
    {
        assert sum < max && count < max;    // second branch with
        count = count+1;                // previous condition
        assert sum < max && count-1 < max;    // count-1 == old count
        sum = sum+count;
        assert sum-count<max && count-1<max; // sum-count == old sum
    }
}
```

---

### 4.8.4 The if-else Conditional

The *if-else statement*

```
if (boolean expression)
    statement1
else
    statement2
```

first evaluates the boolean expression. If the result is true, it then executes the first statement. Otherwise, i.e., if the result is false, it executes the second statement. The control flow of an if-else statement is illustrated in Figure 4.4.

Both branches in an if-else statement may be actually blocks of statements, i.e., the general form of the conditional is

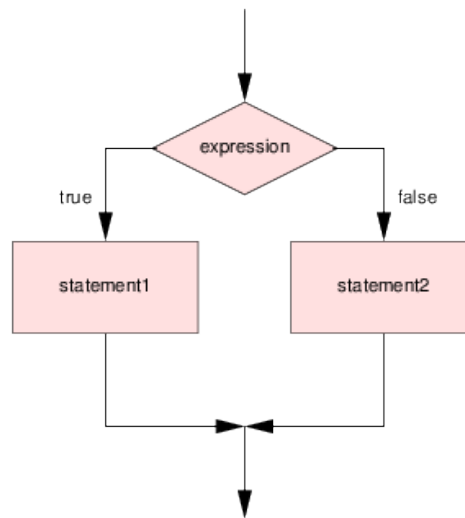


Figure 4.4: The if-else Statement.

```
if (boolean expression)
{
    statement;
    statement;
    ...
}
else
{
    statement;
    statement;
    ...
}
```

Depending on the result of the evaluation, either the statements of the first block or the statements of the second block are executed.

**Example** In the program

```
if (value % 2 == 0)
    value = value/2;
else
{
    value = value-1;
```

```
    count = count+1;
}
```

we divide *value* by 2, if is even, and we decrease *value* by 1 and increase *count* by 1, otherwise.

---

---

**Example** Take the program

```
if (a > b)
    if (a > 0) max = a;
else
    max = b;
```

The compiler actually reads this program as

```
if (a > b)
    if (a > 0)
        max = a;
else
    max = b;
```

i.e., the *else* branch belongs to the preceding *if*. This *dangling else* problem can be overcome by introducing block braces as in

```
if (a > b)
{
    if (a > 0) max = a;
}
else
    max = b;
```

In general, it is a good idea to use explicit braces for all *if* branches apart for the simplest ones (e.g., assignments).

---

---

## Assertions

In an `if-else` statement, we can add the following assertions:

```
if (expression)
{
    assert expression;
    statement;
    ...
}
else
{
    assert !expression;
    statement;
    ...
}
```

i.e., we know that in the second branch the boolean expression does *not* hold.

---

### Example (Minimum of Three Numbers)

The following code stores in variable `min` the minimum of three numbers `a`, `b`, and `c`:

```
if (a < b)
    if (a < c)
        min = a;
    else
        min = c;
else
    if (a < c)
        min = b;
    else
        min = c;
```

We can check this by deriving the assertions:

```
if (a < b)
{
    assert a < b;
    if (a < c)
    {
        assert a < b && a < c;
```

```
    min = a;
}
else
{
    assert a < b && c <= a;
    min = c;
}
}
else
{
    assert b <= a;
    if (b < c)
    {
        assert b <= a && b < c;
        min = b;
    }
    else
    {
        assert b <= a && c <= b;
        min = c;
    }
}
}
```

---

---

### Multi-Branch Conditionals

The else branch of a conditional may be a `if-else` statement itself such that we can construct *multi-branch conditionals* of the following form:

```
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
...
else
    statementN
```

In the execution of such a statement, one boolean expression after the other is evaluated and the first branch is executed, for which this evaluation returns true.

If none of the expressions yields true, the final `else` branch is executed (if such a branch exists).

---

**Example** In the program

```
if (value < 5)
    count = count+1;    // value < 5
else if (value < 10)
    count = count+2;    // 5 <= value < 10
else if (value < 20)
    count = count+3;    // 10 <= value < 20
else
    count = count+4;    // 20 <= value
```

we decompose the variable range of *value* into four parts and execute one of four branches.

---

---

---

**Example** The most typical use of the input operations explained in Section 4.7 is shown by the following program structure:

```
int i = Input.readInt();
if (Input.isOkay())
{
    handle input i
}
else if (Input.hasEnded())
{
    handle case that input stream has ended
}
else
{
    String error = Input.getError();
    handle error case
    Input.clearError();
}
```

---

---

In general, we may assume in each branch that none of the tests preceding the branch has been successful:

```
if (exp1)
{
    assert exp1;
    ...
}
else if (exp2)
{
    assert !exp1 && exp2;
    ...
}
else if (exp3)
{
    assert !exp1 && !exp2 && exp3;
    ...
}
...
else
{
    assert !exp1 && !exp2 && !exp3 && ... && !expN-1;
    ...
}
```

### 4.8.5 The switch Conditional

Frequently, we want the program flow to follow one of several possible cases determined by the value of an expression. Rather than writing

```
if (expression == const1)
    stat1;
else if (expression == const2)
    stat2;
else if (expression == const3)
    stat3;
...
else
    statN;
```

we can use the *switch statement*

```
switch (expression)
{
    case const1:
```

```
        stat1;
        break;
    case const2:
        stat2;
        break;
    case const3:
        stat3;
        break;
    ...
    default:
        statN;
}
```

where the expression must be of an integer type or of type `char` or of type `String` and all case tags must be literals or constants of this type; the same case tag must not appear in two different branches.

The `switch` statement evaluates the expression; if the result equals one of the case tags, the corresponding statement is selected for execution. If the expression value does not equal any case tag, the default branch is executed (if there is one). In any case, execution continues with the statement after the conditional.

The `break` statement should follow each case; if it is omitted, execution continues with the statement of the textually following case (which is usually not what we want).

This is bad language design inherited from C++.

The main advantage of the `switch` construct is its efficient implementation: rather than comparing the value of the expression with one case constant after the other, the system uses a precomputed table of jump addresses for each possible case constant. By looking up this table, the system determines in one step the right branch and switches execution to this branch.

A branch can also be prefixed by *multiple case tags*; the corresponding statement is executed, if the expression value equals one of the tags.

---

#### Example The program

```
switch (i%5) // value is in 0..4
{
    case 0:
        System.out.println("divided by 5");
        break;
    case 1: case 3:
        System.out.println("odd remainder");
}
```

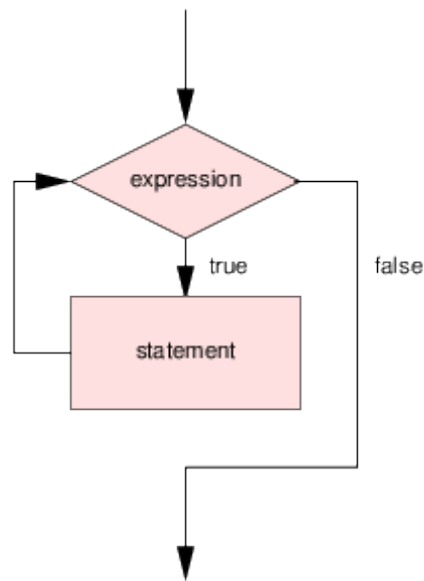


Figure 4.5: The while Statement.

```
    break;
default:
    System.out.println("even remainder");
}
```

executes one of the three branches.

### 4.8.6 The while Loop

A *while loop* has the form

```
while (boolean expression)
    statement
```

The construct evaluates the boolean expression (the *loop condition*) and, if the result is true, executes the statement (the *loop body*). However, unlike the `if` statement, this process is *repeated* until the loop condition is false; execution then proceeds with the next statement after the loop. The control flow of the statement is depicted in Figure 4.5.

**Example** The loop in the program

```
int limit = 3;
int i = 0;
int s = 0;
while (i < limit)
{
    s = s+i;
    i = i+1;
}
```

is executed three times; the effect of the loop is therefore the same as that of the block

```
{
    s = s+i; i = i+1; assert i == 1;
    s = s+i; i = i+1; assert i == 2;
    s = s+i; i = i+1; assert i == 3;
}
```

The loop in the program

```
int n = 8;
int i = 0;
while (n % 2 == 0)
{
    n = n/2;
    i = i+1;
}
```

is also executed three times; the effect is the same as that of

```
{
    n = n/2; i = i+1; assert n == 4;
    n = n/2; i = i+1; assert n == 2;
    n = n/2; i = i+1; assert n == 1;
}
```

The loop in the program

```
int n = 7;
int i = 0;
while (n % 2 == 0)
{
    n = n/2;
    i = i+1;
}
```

is *never* executed.

The loop in program

```
int n = 0;
int i = 0;
while (n % 2 == 0)
{
    n = n/2;
    i = i+1;
}
```

runs *infinitely*, i.e., the program does not terminate (why?).

---

---

### Assertions

When we have entered a loop, we can be sure that the loop condition holds; when we continue execution after a loop, we can be sure that the loop condition does not hold. We can therefore always annotate a loop with the following assertions

```
while (expression)
{
    assert expression;
    ...
}
assert !expression;
```

When designing a loop, it is always good to find the corresponding *loop invariant*, a non-trivial condition that characterizes the loop because it holds at the beginning of every iteration and also after the loop has terminated. We can then annotate the loop as follows:

```
while (expression)
{
    assert expression && invariant;
    ...
}
assert !expression && invariant;
```

---

### Example (Exponentiation)

The following program computes  $r := a^b$ :

```
int r = 1;
int i = 0;
while (i < b)
{
    r = r*a;
    i = i+1;
}
```

The idea for this loop is that, in every iteration  $i$ , we have  $r = a^i$  and  $i \leq b$ . Since after the termination of the loop  $i \leq b$  holds, this implies  $i = b$  and  $r = a^b$ .

We can express this idea by the following assertions (where  $\text{power}(a, i)$  is a pseudo-expression that shall denote  $a^i$ ):

```
int r = 1;
int i = 0;
while (i < b)
{
    assert i < b && r == power(a,i) && i <= b;
    r = r*a;
    i = i+1;
}
assert i >= b && r == power(a,i) && i <= b;
```

---

### 4.8.7 break and continue

There are two statements that are frequently used to control loops in addition to the loop condition. The statement

`break`

immediately terminates the enclosing loop. The statement

`continue`

skips the rest of the loop body, i.e., the program continues with the evaluation of the loop condition and possibly the next loop iteration.

---

**Example** Take the following program to compute the product of  $n$  non-negative numbers read from the input stream

```
int i = 0;
int r = 1;
while (i < n)
{
    int a = Input.readInt();
    if (!Input.isOkay()) break;
    if (a < 0) continue;
    r = r*a;
    i = i+1;
}
```

The loop terminates prematurely if the input stream ends or an input error occurs. If a negative number is read, the rest of the body is skipped and the loop continues with reading the next number.

---

If we have *nested loops* (loops within loops), we can use *labels* to explicitly say which loop is to be terminated respectively continued. For instance, in the program

```
l: while (...)
{
    while (...)
    {
        if (...) break l;
        if (...) continue l;
    }
}
```

the outer loop is labeled with identifier *l*. The `break` in the inner loop uses this label to terminate the outer loop and continue with the first statement after the outer loop. The `continue` in the inner loop uses this label to continue execution with the check of the termination condition of the outer loop. In general, this feature is rarely used.

The statements `break` and `continue` are not absolutely necessary; they can be always replaced by more complex loop conditions, possibly with the help of auxiliary variables.

---

**Example** Above program can be written without `break` and `continue` as follows:

```
int i = 0;
int r = 1;
```

```
boolean okay = true;
while (i < n && okay)
{
    int a = Input.readInt();
    okay = Input.isOkay();
    if (okay && a >= 0)
    {
        r = r*a;
        i = i+1;
    }
}
```

We use the auxiliary boolean variable `okay` to hold the status of the last read operation such that we can exit the loop if an error has occurred. In order to enter the loop for the first time, we have to set `okay` to `true`.

---

Since `break` and `continue` are not absolutely necessary and may make loops difficult to understand, some people suggest to avoid them at all. However, as we see from above example, avoiding these statements can make loops harder to read. We therefore recommend to formulate loops in that way that makes it simpler to understand; it is secondary, whether this involves `break` or `continue` or not.

---

### Example (Stream Processing)

For the iterative processing of a stream of inputs, we typically have the following program structure:

```
while (true)
{
    int i = Input.readInt();
    if (Input.hasEnded()) break;
    if (!Input.isOkay())
    {
        String error = Input.getError();
        handle error case
        Input.clearError();
        continue;
    }
    handle input i
}
handle case that input stream has ended
```

In every loop iteration we read another value; if the input stream has ended, we leave the loop with a `break` statement and then perform the termination code. Otherwise, we check whether an error has occurred and, if yes, deal with the error, clear the error, and continue by the `continue` statement with another loop iteration. Otherwise, we handle the input read.

In this example, the loop is only exit by `break` and not by the loop condition (which is always `true`).

---

---

### Example (Interactive Input)

When input values are entered interactively by a human, we have to take into account that the reason for an input error is that the human mistyped the input. In this case, we want to give him another chance by printing a notification and repeating the input.

An abstract action *read input i* is therefore often concretely implemented as:

```
int i;
while(true)
{
    System.out.print("Enter input: ");
    i = Input.readInt();
    if (Input.isOkay() && other condition on i) break;
    if (Input.hasEnded())
    {
        System.out.println("End of input!");
        System.exit(0);
    }
    System.out.println("Invalid input!");
    Input.clearError();
}
handle input i
```

This program uses a new call

```
System.exit(integer expression);
```

which terminates program execution with the denoted exit code. By convention, an exit code 0 signals normal program termination. Any other code signals exceptional termination; a negative exit code signals an error.

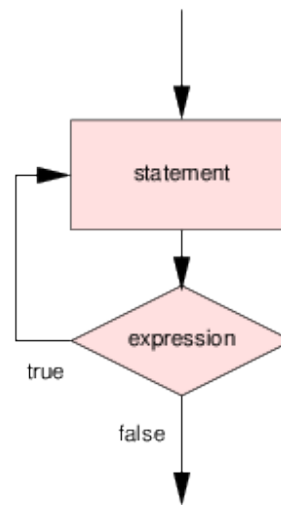


Figure 4.6: The do Statement.

---

The *other condition on input* expresses any other check that we would like to perform on the input. The loop only terminates when some *i* has been entered for which this check succeeds. The program aborts execution if the input stream ends.

---

---

### 4.8.8 The do Loop

A *do loop* has the form

```
do
    statement
while (boolean expression)
```

This construct first executes the statement (the loop body) and then evaluates the boolean expression (the *loop condition*). If the result is true, the process is iterated. The control flow of the statement is depicted in Figure 4.6.

---

**Example** The following programs reads one number after another and prints it until (an error occurs or) a zero is read:

```
do
{
    int i = Input.readInt();
    if (!Input.isOkay()) break;
    System.out.println(i);
}
while (i != 0)
```

---

Both loop constructs, `do` and `while` have equal power; the effect of a `do` loop can be expressed as

```
statement
while (expression)
    statement
```

while the effect of a `while` loop can be simulated by

```
if (expression)
do
    statement
while (expression)
```

Use that form that is more convenient for your particular situation.

### Assertions

A `do` loop can be annotated with assertions as follows:

```
do
{
    assert invariant;
    statement;
}
while (expression);
assert !expression && invariant;
```

---

**Example** The program

```

int number = i;
String reverse = "";
do
{
    int digit = number%10;
    reverse = reverse + digit;
    number = number/10;
}
while (number > 0);
System.out.println(reverse);

```

takes a non-negative number  $i$  (say 2846) and prints its reverse (6482).

At the beginning of every loop iteration, we have the following variable values (the first is an integer, the second one is a string):

number	reverse
2846	
284	6
28	64
2	648
0	6482

We can therefore annotate the loop as follows:

```

do
{
    assert str(i).equals(str(number)+rev(reverse));
    int digit = number%10;
    reverse = (reverse*10) + digit;
    number = number/10;
}
while (number > 0);
assert !(number > 0) && str(i).equals(str(number)+rev(reverse));

```

where the pseudo-function `str` returns the string representation of a number (the empty string for 0) and the pseudo-function `rev` returns the reverse of a string. From the condition after the loop it follows that `number` is 0 and that `reverse` is therefore the reverse of the string representation of `i`.

### 4.8.9 The for Loop

The *for loop* has the general form

```
for (init; boolean expression; update)  
    statement
```

where *init* may be a variable declaration or a statement and *update* is a statement. Its behavior is identical to the following program:

```
{  
    init;  
    while (boolean expression)  
    {  
        statement;  
        update;  
    }  
}
```

This construct therefore first executes the statement (or variable declaration) *init*, then checks the boolean expression (the loop condition) and if the result is true, executes the loop body *statement* followed by the *update* statement. This process is iterated until the loop condition is false. The control flow of the statement is depicted in Figure 4.7.

The most relevant form of the *for* loop is

```
for (int var = val; var < exp; var++)  
    statement
```

where

```
var++
```

is a Java statement that uses the *increment operator* to increase the content of the integer variable *var* by 1, i.e., it is equivalent to

```
var = var+1
```

A more general form of the increment operator is

```
var += exp
```

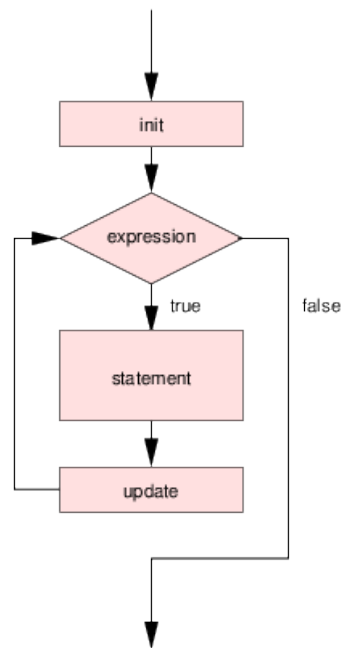


Figure 4.7: The for Statement.

---

which is equivalent to

$$var = var + exp$$

To use a declaration `int var = val` in the initialization part of a for loop means the declaration of a new variable; this variable is only visible *within* the loop. We recommend to use only such locally declared variables as loop variables. However, one can also use an initialization part `var = val` which refers to a previously declared variable. When the loop terminates, such a variable will retain its value (including the update after the last iteration).

---

**Example** The program

```
for (int i = 0; i < n; i++)
    System.out.println("i: " + i);
```

prints  $n$  values

```
i: 0
i: 1
i: 2
...
i: n-1
```

If  $n$  is 0, no value is printed. The program

```
for (int i = 0; i < n; i++)
    System.out.println("i: " + i);
for (int i = 0; i < n; i++)
    System.out.println("i: " + (-i));
```

prints once the sequence above and once the sequence

```
i: 0
i: -1
i: -2
...
i: -(n-1)
```

---

### Decreasing Loops

Corresponding to the increment operator, we have the *decrement operator* in two variants

```
var--
var -= exp
```

which are equivalent to

```
var = var-1
var = var-exp
```

The corresponding typical loop structure is

```
for (int var = val; var >= exp; var--)
    statement
```

---

**Example** The program

```
for (int i = n-1; i >= 0; i--)
    System.out.println("i: " + i);
```

prints  $n$  values

```
n-1
...
2
1
0
```

If  $n$  is 0, no value is printed.

---

### Assertions

We can annotate a for loop

```
for (int var = init; var < max; var++)
    statement
```

by the following assertions

```
for (int var = init; var < max; var++)
{
    assert init <= var && var < max;
    statement
}
```

One should strive to design a loop invariant such that the following program is well annotated:

```
for (int var = init; var < max; var++)
{
    assert init <= var && var < max && invariant;
    statement
}
assert invariant[var ← max];
```

where  $\text{invariant}[var \leftarrow max]$  denotes a version of  $\text{invariant}$  where every occurrence of  $var$  has been replaced by  $max$ .

---

### Example (Exponentiation)

The program

```
int r = 1;
for(int i = 0; i < b; i++)
    r = r*a;
```

for the computation of  $r := a^b$  (see page 81) can be annotated as follows:

```
int r = 1;
for(int i = 0; i < b; i++)
{
    assert 0 <= i && i < b && r = exp(a, i);
    r = r*a;
}
assert r = exp(a, b);
```

---

## 4.9 Sample Programs

We conclude this section by illustrating the development of some sample programs which demonstrate the use of the constructs introduced in the previous sections.

### 4.9.1 Printing Stars

We are given the problem to print  $n$  lines of output of form

```
*
**
***
****
*****
...
```

i.e., the  $n$ -th line shall contain  $n$  stars. The number  $n$  is provided by the user as text input.

The problem apparently is of an iterated nature: we shall print one line after the other. The corresponding program structure is

```
// print n lines
for (int i = 1; i <= n; i++)
    print line with i stars
```

Printing a line with  $i$  stars is again an iterative problem: we have to print one star after; the line is finally terminated by a newline character. The corresponding program structure is:

```
// print line with i stars
for (int j = 1; j <= i; j++)
    System.out.print("*");
System.out.println("");
```

Therefore the core program has the following structure:

```
// print n lines
for (int i = 1; i <= n; i++)
{
    // print line with i stars
    for (int j = 1; j <= i; j++)
        System.out.print("*");
    System.out.println("");
}
```

The complete program including input and error check can be therefore given as shown below.

```
public class Stars
{
    public static void main(String[] args)
    {
        int n = 0; // number of lines

        // read n, exits on empty input, retries on invalid input
        while (true)
        {
```

```
System.out.print("Enter number of lines: ");
n = Input.readInt();
if (Input.isOkay()) break;
if (Input.hasEnded())
{
    System.out.println("No input given!");
    System.exit(-1);
}
System.out.println("Invalid input!");
Input.clearError();
}

// print n lines
for (int i = 1; i <= n; i++)
{
    // print line with i stars
    for (int j = 1; j <= i; j++)
        System.out.print("*");
    System.out.println("");
}
}
```

You can download the source code of the application to compile and run it.

### 4.9.2 Drawing Circles

We are given the task to fill a disk with alternating circles of black and white color each of which has a particular width. The innermost circle (the one which has radius less than or equal to this width) is to be painted in red. In short, we want to paint a “bull’s eye” as shown in Figure 4.8.

Given the graphical primitives of page 39, we first have to design a strategy by which the corresponding picture can be drawn. The only primitive of use seems to be `fillOval` by which we can paint a filled ellipse and therefore also a circular disk. The effect of a “bull’s eye” can be achieved by drawing first a black disk of maximum radius, then painting on top of the black disk a white disk with a somewhat smaller radius, then again painting a black disk with somewhat smaller radius and so on, until the radius becomes small enough to paint the red disk.

The general program structure therefore is something like

```
// draw alternating circles of black and white
```

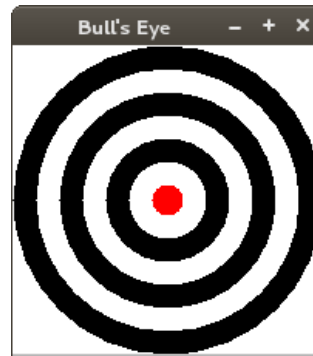


Figure 4.8: Bull's Eye

```
// starting with outermost circle with radius = RADIUS
// iteratively shrink radius by WIDTH until radius <= WIDTH
for (int radius = RADIUS; radius > WIDTH; radius -= WIDTH)
{
    switch color from black to white or from white to black
    draw circle with denoted radius and denoted color
}
draw innermost circle in red
```

To represent the color, we need a flag which indicates whether the next disk is to be drawn in black or in white. By checking the state variable, we can set the drawing color and switch to the other color for the next iteration. The refined program therefore has the following structure:

```
boolean black = true;
for (int radius = RADIUS; radius > WIDTH; radius -= WIDTH)
{
    // set color, switch to other color
    if (black)
    {
        Drawing.graphics.setColor(Color.black);
        black = false;
    }
    else
    {
        Drawing.graphics.setColor(Color.white);
        black = true;
    }
}
```

```

    draw circle with denoted radius and denoted color
}
draw the innermost circle in red

```

The requirement “draw circle with denoted radius and denoted color” is met by the statement

```

// draw circle of denoted radius with center (MID_X, MID_Y)
Drawing.graphics.fillOval(MID_X-radius, MID_Y-radius, 2*radius, 2*radius);

```

where MID\_X and MID\_Y represent the coordinates of the center of the disk.

What remains to be discussed is how we can “draw the innermost circle in red”, in particular what is the radius of this circle? Assume that WIDTH=15 and in the last iteration of the loop *radius*=25. The last update operation decreases *radius* to 10 which is smaller than WIDTH and the loop terminates. Therefore the final red circle has radius 10.

However, the problem is that the last value of *radius* is not visible outside the loop because of the local declaration of the loop variable:

```

for (int radius = ...; ...; ...)

```

We basically have three choices to overcome the problem:

1. Recompute the radius of the red circle from RADIUS and WIDTH.
2. Place the code for *draw the innermost circle in red* into the loop as

How can you do this without a loop?

```

for (int radius = RADIUS; radius > WIDTH; radius -= WIDTH)
{
    ...
    if (radius-WIDTH <= WIDTH)
    {
        draw circle of radius "radius-WIDTH" in red
    }
}

```

3. Retain the value of the iteration variable by changing the declaration of the loop variable to

```
int radius = 0;
for (radius = RADIUS; radius > WIDTH; radius -= WIDTH)
{
    ...
}
draw circle of denoted radius in red
```

The last solution is the simplest one, so we go for it.

The complete code of the program's main method can be therefore written as follow (we use constants for the disk radius and the circle width rather than asking for user input):

```
public static void main(String[] args)
{
    final int RADIUS = 100;    // disk radius
    final int WIDTH = 15;     // width of each circle drawn
    final int MID_X = RADIUS; // x coordinate of center
    final int MID_Y = RADIUS; // y coordinate of center

    Drawing.begin("Bull's Eye", 2*RADIUS, 2*RADIUS);

    boolean black = true;    // next color drawn

    int radius = 0;         // we retain radius for last circle

    // draw alternating circles of black and white
    // starting with outermost circle with radius = RADIUS
    // iteratively shrink radius by WIDTH until radius <= WIDTH
    for (radius = RADIUS; radius > WIDTH; radius -= WIDTH)
    {
        // set color, switch to other color
        if (black)
        {
            Drawing.graphics.setColor(Color.black);
            black = false;
        }
        else
        {
            Drawing.graphics.setColor(Color.white);
            black = true;
        }
    }
}
```

```
// draw circle of denoted radius with center (MID_X, MID_Y)
Drawing.graphics.fillOval(MID_X-radius, MID_Y-radius, 2*radius, 2*radius);
}

// draw red center circle
Drawing.graphics.setColor(Color.red);
Drawing.graphics.fillOval(MID_X-radius, MID_Y-radius, 2*radius, 2*radius);

Drawing.end();
}
```

Figure 4.8 is actually a screenshot of this program.

You can download the source code of the application to compile and run it.

### 4.9.3 Analyzing Exam Grades

We are given the problem to analyze exam grades (1–5). The grades given for an exam are to be read from the standard input stream; the analysis is to determine the best grade, the worst grade, the average grade, and to print these values to the standard output stream. The program shall be able to process the grades of multiple exams.

The problem is apparently of iterated nature; in each iteration, we analyze an exam and print the result. To determine whether to continue, we ask the user; if the answer is not 'y', we are done. The problem to *analyze exam grades* can therefore be solved by a program with the following structure:

```
// analyze exam grades
char answer = 0;
do
{
    analyze exam;
    print result;
    ask user whether to continue;
}
while (answer == 'y');
```

We start with the refinement of *ask user whether to continue* which follows our general structure for processing inputs. we should only take answers into account that are 'y' or 'n', because any other input may be a typing error. We therefore get the code fragment

```

// ask user whether to continue
while(true)
{
    System.out.print("Another exam (y/n)? ");
    answer = Input.readChar();
    if (Input.isOkay() && (answer == 'y' || answer == 'n')) break;
    if (Input.hasEnded())
    {
        System.out.println("Unexpected end of input!");
        System.exit(-1);
    }
    System.out.println("Invalid input!");
    Input.clearError();
}

```

For analyzing an exam, we have to determine the number of grades, the minimum (“best”) grade, the maximum (“worst”) grade, and the sum of all grades. When we have all grades, we can then determine the average grade as the quotient of the sum and the number of grades. Thus we refine *analyze exam* to

```

// analyze exam
int count = 0;           // number of grades processed so far
int sum = 0;             // sum of grades processed so far
int min = Integer.MAX_VALUE; // minimum grade processed so far
int max = Integer.MIN_VALUE; // maximum grade processed so far
read and process exam

```

where we initialize *min* with a value that is larger than any possible grade (such that the first grade read will become the first candidate for the minimum, see below) and *max* with a value that is smaller than any possible grade (such that the first grade read will become the candidate for the maximum). We then also refine *print result* to

```

// print result
System.out.println("\nNumber of grades: " + count);
System.out.println("Best grade: " + min);
System.out.println("Worst grade: " + max);
System.out.println("Average grade: " + sum/count);

```

Do you see any problem with this refinement?

What remains to be done, is therefore to refine *read and process exam*. This problem is of iterative nature where in every iteration we read and process a new grade. However, we have to determine when this processing ends, i.e., when all

grades of an exam have been read. We choose a solution where the user enters a grade 0 as a “sentinel” that denotes the end of the input. Our refinement therefore looks as follows:

```
// read and process exam
while (true)
{
    int grade = 0;
    read grade
    if (grade == 0) break;
    process grade
}
```

The task *read grade* can be refined according to our usual strategy; a valid input is an integer in the range 0..5:

```
// read grade
while(true)
{
    System.out.print("Enter grade " + (count+1) + " (0 when done): ");
    grade = Input.readInt();
    if (Input.isOkay() && 0 <= grade && grade <= 5) break;
    if (Input.hasEnded())
    {
        System.out.println("Unexpected end of input!");
        System.exit(-1);
    }
    System.out.println("Invalid input!");
    Input.clearError();
}
```

The final (and core) task is then to refine *process grade* which must update the variables *count*, *sum*, *min*, and *max*. We can perform this simply as follows:

```
// process grade
count++;
sum += grade;
if (grade < min) min = grade;
if (grade > max) max = grade;
```

So we are done and have developed a program with the following abstraction levels

```
analyze exam grades  
analyze exam  
read and process exam  
read grade  
process grade  
print result  
ask user whether to continue
```

resulting in the following program structure:

```
// analyze exam grades  
char answer = 0;  
do  
{  
    // analyze exam  
    int count = 0;           // number of grades processed so far  
    int sum = 0;            // sum of grades processed so far  
    int min = Integer.MAX_VALUE; // minimum grade processed so far  
    int max = Integer.MIN_VALUE; // maximum grade processed so far  
  
    // read and process exam  
    while (true)  
    {  
        int grade = 0;  
        read grade  
        if (grade == 0) break;  
        process grade  
    }  
  
    print result  
  
    ask user whether to continue  
}  
while (answer == 'y');
```

We insert the code fragments for the parts in *italics* to build the complete program (see the end of this section), compile it and execute it.

We now test the program with the following dialogue:

```
Enter grade 1 (0 when done): 3  
Enter grade 2 (0 when done): 4  
Enter grade 3 (0 when done): 5
```

```
Enter grade 4 (0 when done): 0
```

```
Number of grades: 3
Best grade: 3
Worst grade: 5
Average grade: 4
```

Looks fine, so we are happy and continue

```
Another exam (y/n)? y
Enter grade 1 (0 when done): 2
Enter grade 2 (0 when done): 4
Enter grade 3 (0 when done): 4
Enter grade 4 (0 when done): 0
```

```
Number of grades: 3
Best grade: 2
Worst grade: 4
Average grade: 3
```

We get more and more confident and continue testing until we happen to type

```
Another exam (y/n)? y
Enter grade 1 (0 when done): 0

Number of grades: 0
Best grade: 2147483647
Worst grade: -2147483648
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Exams.main(Exams.java:51)
```

Uuups! We realize that when no grade is entered, the variables *min* and *max* have their initial values and *count* and *sum* are 0. Therefore the division gives an error and the program aborts. We therefore better rewrite *print result* to

```
// print result
System.out.println("\nNumber of grades: " + count);
if (count > 0)
{
    System.out.println("Best grade: " + min);
    System.out.println("Worst grade: " + max);
    System.out.println("Average grade: " + sum/count);
}
```

since these values only make sense when at least one grade is entered. We now can safely type

```
Enter grade 1 (0 when done): 0

Number of grades: 0

Another exam (y/n)?
```

We are happy to have detected and fixed this *bug* (program error). So we are even more confident than before and run

```
Another exam (y/n)? y
Enter grade 1 (0 when done): 2
Enter grade 2 (0 when done): 4
Enter grade 3 (0 when done): 4
Enter grade 4 (0 when done): 5
Enter grade 5 (0 when done): 0

Number of grades: 4
Best grade: 2
Worst grade: 5
Average grade: 3
```

Hmm, this looks a bit strange. Our feeling says that the average grade should be rather 4 than 3, so we look into the program where we compute the average

```
System.out.println("Average grade: " + sum/count);
```

Uuups, again! We realize that *sum* and *count* are integer variables, therefore we have computed the average by truncated integer division. We change this to floating point division by inserting the type casts

```
System.out.println("Average grade: " + (float)sum/(float)count);
```

Actually, because of arithmetic promotion (see page 57), it would suffice, to cast just one of the variables to `float`, the other would then be automatically promoted. We run the program again and get

```
Enter grade 1 (0 when done): 2
Enter grade 2 (0 when done): 4
Enter grade 3 (0 when done): 4
```

The first computers malfunctioned because of real bugs in their interiors.

```
Enter grade 4 (0 when done): 5
Enter grade 5 (0 when done): 0

Number of grades: 4
Best grade: 2
Worst grade: 5
Average grade: 3.75
```

which is what we actually want.

More thorough testing does not reveal any more problems, thus we are happy to ship the following code for the main method of our application:

```
// analyze exam grades
char answer = 0;
do
{
    // analyze exam
    int count = 0;           // number of grades processed so far
    int sum = 0;            // sum of grades processed so far
    int min = Integer.MAX_VALUE; // minimum grade processed so far
    int max = Integer.MIN_VALUE; // maximum grade processed so far

    // read and process exam
    while (true)
    {
        // read grade
        int grade = 0;
        while(true)
        {
            System.out.print("Enter grade "+(count+1)+" (0 when done): ");
            grade = Input.readInt();
            if (Input.isOkay() && 0 <= grade && grade <= 5) break;
            if (Input.hasEnded())
            {
                System.out.println("Unexpected end of input!");
                System.exit(-1);
            }
            System.out.println("Invalid input!");
            Input.clearError();
        }

        // check for end of data set
        if (grade == 0) break;
    }
}
```

```
// process grade
count++;
sum += grade;
if (grade < min) min = grade;
if (grade > max) max = grade;
}

// print results for data set
System.out.println("\nNumber of grades: " + count);

// these values are only valid, if there was at least one grade
if (count > 0)
{
    System.out.println("Best grade: " + min);
    System.out.println("Worst grade: " + max);
    System.out.println("Average grade: " + (float)sum/(float)count);
}
System.out.println("");

// ask whether to continue
while(true)
{
    System.out.print("Another exam (y/n)? ");
    answer = Input.readChar();
    if (Input.isOkay() && (answer == 'y' || answer == 'n')) break;
    if (Input.hasEnded())
    {
        System.out.println("Unexpected end of input!");
        System.exit(-1);
    }
    System.out.println("Invalid input!");
    Input.clearError();
}
while (answer == 'y');
```

You can download the source code of the application to compile and run it.

Above development illustrated the process of *stepwise refinement* where a problem is repeatedly broken into simpler problems until the problems can be directly solved. You should follow this process whenever you encounter a problem which you cannot solve immediately (i.e., for any problem apart from the most trivial ones).

However, in general, problems like the one above are not any more written as monolithic pieces of code. They are decomposed into separate methods as will be discussed in the next chapter.

# Chapter 5

## Stronger Java

In this chapter, we will introduce those concepts of Java that enable us to write larger programs that solve more complex problems. These constructs mainly deal with the structuring of code (*methods*) and of data (*classes*). Their combined application already provides a good deal of the expressiveness of the programming language. Furthermore, we will introduce *arrays* as one of the most important data structures of Java. Having studied this section, one should be able to develop large programs that consist of multiple classes and call various methods that process collections of values.

What will be not yet discussed in this chapter is how to separate the internal implementation of a class from the public specification (*information hiding*) and how new classes can be constructed by extending existing ones (*inheritance*). These issues will be presented in a later chapter; they will complete the *object-oriented* features of Java.

### 5.1 Methods

A real-world program is too big to be written as a single piece of code. Rather it is decomposed into smaller pieces called *methods* each of which performs a certain subtask. Each method has a name such that one can call the method arbitrarily often whenever the corresponding subtask is to be performed.

Methods structure program code.

#### 5.1.1 Method Declarations and Calls

The simplest *method declaration* has the form

```
public static void name()
{
    ...
}
```

where *name* is a Java identifier (the *method name*) and the *method body* within { } contains the instructions performed when the method is called.

A corresponding *method call* (or *invokation*) is a statement of the form

```
name();
```

This statement executes the body of method *name* and then continues execution with the next statement after the call.

---

**Example** The program

```
public class Main
{
    public static void main(String[] args)
    {
        printHeader();
        System.out.println(1);
        printHeader();
        System.out.println(2);
    }
    public static void printHeader()
    {
        System.out.println("Number:");
        System.out.println("-----");
    }
}
```

prints on execution

```
Number:
-----
1
Number:
-----
2
```

---

A method performs an activity; the name of the method should reflect this activity in such a way that the method call reads as a command in natural language (`printHeader` reads as “print header!”). We usually write method names with a lower-case first letter.

Methods may be declared in any order. However, a good strategy is to order the method declarations in such a way that method calls either refer only to methods that are declared *after* the call (forward calls) or the other way round (backward calls). Source code organized in either way is easier to understand than if methods are randomly ordered.

### Calling Methods from Other Classes

Methods may be declared in separate classes stored in different files. Methods from different classes may call each other as

```
Name.name();
```

where *Name* identifies the class where the method is invoked.

---

**Example** The program of the previous section can be also written as

```
public class Main
{
    public static void main(String[] args)
    {
        Print.printHeader();
        System.out.println(1);
        Print.printHeader();
        System.out.println(2);
    }
}
public class Print
{
    public static void printHeader()
    {
        System.out.println("Number:");
        System.out.println("-----");
    }
}
```

The declaration of class `Print` is written into a separate file `Print.java`. When we call

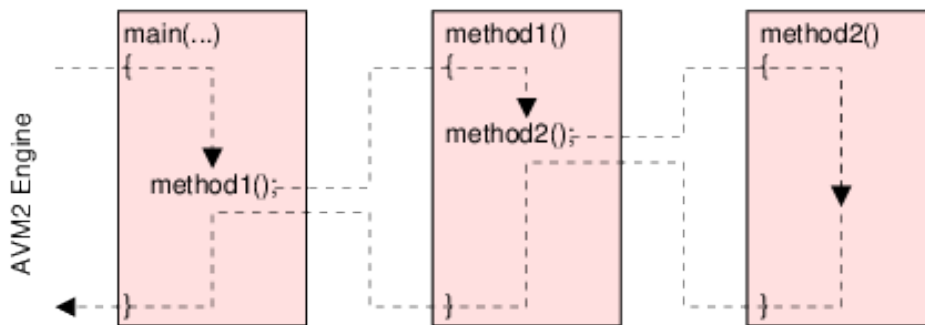


Figure 5.1: Method Calls

```
javac Main.java
```

the compiler also compiles `Print.java` and generates a class file `Print.class`. When we invoke

```
java Main
```

the JVM interpreter loads both class files and gives the same output as in the previous example.

### Chains of Method Calls

A called method may itself call other methods; in this way we get a chain of method calls as shown in Figure 5.1. The JVM interpreter invokes the program by calling the method `main`. When the execution of `main` reaches the call of `method1`, the program passes control to the first instruction of `method1`. When the execution of `method1` reaches the call of `method2`, control is passed to the first instruction of `method2`. When the execution of `method2` is completed, control is returned to `method1` at the first instruction after the call of `method2`. When `method1` is completed, control is returned to `main` at the first instruction after the call of `method1`. When `main` is completed, control is returned to the JVM interpreter.

We will see later that methods can also call *themselves*.

### 5.1.2 The return Statement

A method may decide to return control to the caller before the method has ended. The *return statement*

```
return;
```

immediately terminates the current method; if invoked in the main method, it has the same effect as `System.exit(0)`.

---

**Example** Take the methods

```
public static void main(String[] args)
{
    for (int i = 0; i < 3; i++)
        readPrintInt();
    System.out.println("Done.");
}

public static void readPrintInt()
{
    System.out.print("Enter integer: ");
    int i = Input.readInt();
    if (!Input.isOkay())
    {
        System.out.println("Invalid input!");
        Input.clearError();
        return;
    }
    System.out.println(i);
}
```

The execution of `main` results in the following dialogue:

```
Enter integer: 9
9
Enter integer: f
Invalid input!
Enter integer: -1
-1
Done.
```

---

---

### 5.1.3 Local and Global Variables

The body of a method may contain variable declarations; these variables are *local* to the method, i.e., they are not visible to other methods.

---

**Example** Take the methods

```
public static void main(String[] args)
{
    int i = 1;
    method1();
    System.out.println(i);
}
public static void method1()
{
    int i = 2;
    System.out.println(i);
}
```

The execution of main prints

```
2
1
```

---

If the same variable name is used in different methods, they denote different variables.

---

**Example** The compilation of a class with methods

```
public static void main(String[] args)
{
    int i = 1;
    method1();
    System.out.println(i);
}
public static void method1()
{
    System.out.println(i);
}
```

results in the compiler error

```
Main.java:11: cannot resolve symbol
symbol   : variable i
location: class Main
    System.out.println(i);
                        ^
1 error
```

---

Now, what can we do if we want to use the same constant in different methods? Of course, we could redeclare the constant in every method that uses it, but this is cumbersome and counteracts the main purpose of constant declarations. A better solution is to draw the declaration from the method body into the class body and add the keyword `public static` (as for methods); the constant thus becomes *global* to all methods within the class.

---

**Example** Take the program

```
public class Main
{
    public static final int LENGTH = 10;

    public static void main(String[] args)
    {
        print2TimesN();
        print3TimesN();
    }

    public static void print2TimesN()
    {
        System.out.println(2*LENGTH);
    }

    public static void print3TimesN()
    {
        System.out.println(3*LENGTH);
    }
}
```

The output of this program is

```
20
30
```

---

In the same way, we can declare global variables which may be not only read but also modified by the individual methods.

---

**Example** Take the program

```
public class Main
{
    public static int sum = 0;

    public static void main(String[] args)
    {
        add1(); multiply2(); add1();
        System.out.println(sum);
    }

    public static void add1()
    {
        sum = sum+1;
    }

    public static void multiply2()
    {
        sum = sum*2;
    }
}
```

The output of this program is

3

---

Global variables need not be initialized; if the initializer part is missing, the compiler assigns the variable a *default value*, for instance 0 for all integer types (including char), and 0.0 for the floating point types.

---

**Example** The program

```
public class Main
{
    public static int i;
    public static void main(String[] args)
```

```

    {
        System.out.println(i);
    }
}

prints

0

```

---

Global variables may pass information from one method to another but we will see in the next section that there are better ways to achieve this. However, there is a usage for global variables that cannot be achieved otherwise, namely to preserve state from one method call to another. We will show an example later.

### 5.1.4 Parameters

The parameters in a method head are also called *formal parameters*.

A *parameter* is a value that is passed to a method when the method is invoked. A method declaration may specify in the *method head* a list of parameters

```

public static void name(parameter1, parameter2, ...)
{
    ...
}

```

where *parameter1*, *parameter2*, ... are the *parameter declarations*. If a parameter declaration has the form

*type name*

then the parameter *name* may be used in the method body like a local variable. A parameter *name* declared as

*final type name*

may be used in the method body like a local constant, i.e., its value cannot be modified by an assignment.

The call of a method with parameters looks like

```

name(expression1, expression2, ...)

```

where *expression1*, *expression2*, ... are the *arguments* that are passed to the method. A method call is executed by

1. evaluating the arguments,
2. assigning the argument values to the parameters of the method, and
3. executing the body of the method.

The arguments in a method call are also called *actual parameters*.

---

**Example** Take the program

```
public static void main(String[] args)
{
    int x = 3;
    printSquare("this", x);
    printSquare("that", x+1);
}

public static void printSquare(String key, int val)
{
    System.out.println(key + ": " + val*val);
}
```

When the method is called for the first time, *key* is assigned “this” and *val* is assigned the value 3. In the second call, *key* is assigned “that” and *val* is assigned the value 4. The output of this program is

```
this: 9
that: 16
```

---

Please note that, even if the argument of a method call is a variable, the corresponding parameter is a separate variable which receives a *copy* of the argument value. Changing the value of the parameter in the method body does therefore *not* change the value of the argument!

---

**Example** Take the program

```
public static void main(String[] args)
{
    int i=1;
    increment(i, 2);
    System.out.println("caller: " + i);
}

public static void increment(int i, int n)
{
    i += n;
    System.out.println("method: " + i);
}
```

The parameter *i* in *increment* receives the value of the argument variable *i*. Since the modification of parameter is not propagated to the argument, the output of the program is

```
method: 3
caller: 1
```

---

---

By the use of parameters, we can reuse the same code in different contexts.

---

**Example** Take the program

```
public static void main(String[] args)
{
    int x = 3; int y = 4;
    System.out.print("x = " + x + ", y = " + y);
    System.out.print(", (x+y)*(x+y) = ");
    System.out.println((x+y)*(x+y));
    int a = 5; int b = 6;
    System.out.print("a = " + a + ", b = " + b);
    System.out.print(", (a+b)*(a+b) = ");
    System.out.println((a+b)*(a+b));
}
```

which gives output

```
x = 3, y = 4, (x+y)*(x+y) = 49
a = 5, b = 6, (a+b)*(a+b) = 121
```

We can write this program in a better way as

```
public static void main(String[] args)
{
    printSumSquare("x", 3, "y", 4);
    printSumSquare("a", 5, "b", 6);
}

public static void
printSumSquare(String var1, int val1, String var2, int val2)
{
    System.out.print(var1 + " = " + val1 + ", ");
    System.out.print(var2 + " = " + val2 + ", ");
    System.out.print("(" + var1 + "+" + var2 + ")");
    System.out.print("(" + var1 + "+" + var2 + ") = ");
    System.out.println((val1+val2)*(val1+val2));
}
```

---

Whenever one has some piece of code which is (or may be) required a second time in a similar way, it is a good idea to extract this code into a separate method. Those values that vary from one application to the next become parameters of the method.

Use methods to make your code *more general!*

### 5.1.5 Functions

A *function* is a method that returns a value of a particular type. A function declaration looks like

```
public static type name(parameters)
{
    ...
}
```

where *type* is (the name of) the *return type* of the function. The keyword `void` instead of a type name indicates that the method is a *procedure*, i.e., does not return a value (we have only seen procedure declarations up to now).

A function executes the statement

```
return expression;
```

to return the value of *expression* to the caller; the type of this value must correspond to the return type of the function. The last statement executed by a function must be such a statement (i.e., a function must not terminate by just reaching the end of the body).

A *function application* is an expression of form

*name(arguments)*

and may occur in any place where an expression of the function's return type may occur. An application is evaluated by invoking the corresponding function; the value returned by the function is the value of the application.

---

**Example** Take the program methods

```
public static void main(String[] args)
{
    int n = 3;
    int r = exp(n, 2);
    System.out.println(r);
    System.out.println(exp(n, 3));
    if (exp(n, 4) > 80)
        System.out.println("Yes.");
}

public static int exp(int a, int b)
{
    int r = 1;
    for (int i = 0; i < b; i++)
        r = r*a;
    return r;
}
```

The execution of `main` generates output

```
9
27
Yes.
```

---

Since the application of a function returns a value, its name should describe the computed result, e.g., the application `exp(a, b)` reads as the “result of the exponentiation of *a* by *b*”.

### 5.1.6 Program Function versus Mathematical Functions

There are two kinds of program functions which are characterized by the following conditions:

1. The function result is determined only by the function arguments (and possibly global constants); the execution of the function does not cause any *side effect* like modifying a global variable or reading input or generating output.

Such a program function behaves like a *mathematical function*.

2. The function result depends on the value of a global variable or a global variable is modified by the function or the function reads input or generates output.

Such a program function is a *procedure with return value*.

We will from now on differentiate between these two kinds of program functions. The function `exp` given in the last section was of the first kind. However, there are also examples of the second kind.

---

**Example** Take the methods

```
public static void main(String[] args)
{
    while (true)
    {
        int n = askForNat();
        if (n < 0) break;
        System.out.println(n);
    }
}

public static int askForNat()
{
    System.out.print("Enter natural: ");
    int i = Input.readInt();
    if (Input.isOkay() && i >= 0)
        return i;
    return -1;
}
```

The function `askForNat` is a procedure with return value that reads a natural number from the standard input stream; the function returns the number or `-1`, if an error has occurred.

---

An application *name(arguments)* of a procedure with return value should only occur in declarations or assignments as in

```
type var = name(arguments);  
var = name(arguments);
```

i.e., as separate statements that assign the result to a variable or constant. If a function application occurs in the context of another expression as in

```
... name(arguments) + expression ...
```

we expect *name* to behave like a mathematical function that could be applied twice with identical results.

Since the main aspect of a procedure with return value is to perform an activity (resulting in a value), its name should express the activity rather than the result. For instance, the call

```
int i = Input.readInt();
```

reads as “*i* is the result of reading an integer from the input stream”.

---

### Example (Random Number Generation)

Take the classes

```
public class Main  
{  
    public static void main(String[] args)  
    {  
        for (int i=0; i < 1000; i++)  
        {  
            int r = nextRandom(128);  
            System.out.println(r);  
        }  
    }  
}
```

```
public class Random
{
    public static int old = 314152;

    public static int nextRandom(int n)
    {
        int x = old;
        int high = x/127773;
        int low = x%127773;
        int val = 16807*low - 2836*high;
        if (val <= 0) val += Integer.MAX_VALUE;
        old = val;
        return val%n;
    }
}
```

On execution, `main` prints the seemingly random sequence

```
26
18
38
101
...
```

Every application of `nextRandom(n)` returns a new non-negative number less than *n* whose value cannot be predicted without knowledge of the body of the function. Such a function is called a *random number generator*; it does definitely not behave like a mathematical function which would return always the same result for same input. We should therefore not write

```
System.out.println(nextRandom(128));
```

because this looks as if always the same value were printed.

The behavior of `nextRandom` is possible because the function records in a global variable *old* the previously returned value which, in addition, to the function argument *n*, is used to compute the function result. Such a function with *hidden state* is always a procedure with return value.

---

---

### 5.1.7 Visibility of Variables

We have learned up to now the following kinds of variables/constants:

- Variables/constants declared in the body of a *class*;
- Parameters declared in a head of a *method*;
- Variables/constants declared in the body of a *method*.
- Variables/constants declared in a *block*.

We now have to clarify at which program position which variable is *visible*, i.e., can be used in program statements. There are two simple rules:

1. The visibility range of a variable/constant is from the point of its declaration to end of the entity (class or method or block) in which it was declared *unless*
2. there is an inner declaration of a variable/constant of the same name; in this case, the inner declaration *shadows* the outer declaration, i.e., the variable of the outer declaration is not visible any more.

We demonstrate these rules by the example in Figure 5.2: Variable  $x$  is declared in the class body; its visibility therefore ranges to the end of the class body. However, the declaration of  $x$  in the method body shadows the outer declaration; until the end of the method body it is the method variable, not the class variable, that is visible.

Also variable  $y$  declared in the class body is visible until the end of the class body; however, the declaration of  $y$  in the loop body shadows the outer declaration; until the end of the loop body it is this variable that is visible and not the class variable. The right picture shows the visibility range of  $y$  and how the inner declaration of  $y$  shadows the outer declaration.

The visibility of parameter  $p$  ranges over the whole method body. Actually, this is always the case because Java does not permit to give a variable in a method body the same name as a parameter.

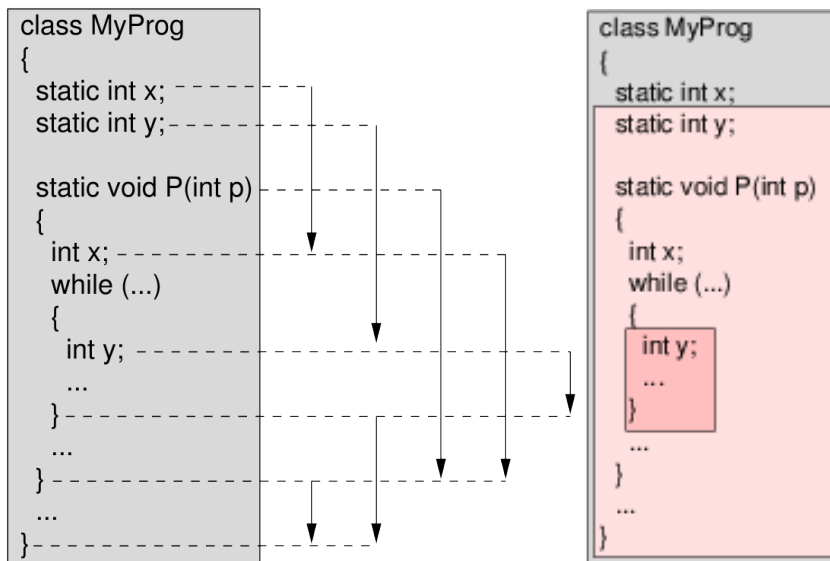


Figure 5.2: Visibility of Variables

### 5.1.8 Method Overloading

Generally all methods declared in the same class must have different names. However, there is an exception to this rule: two methods may have the same name if they have different parameter lists, i.e., if the number of parameters is different or the types of the parameters are different. We call the method name *overloaded* with different methods.

**Example** Take the class

```
public class Main
{
    public static void main(String[] args)
    {
        printLine(3);
        printLine("hi");
    }

    public static void printLine(int i)
    {
        System.out.println("integer: " + i);
    }
}
```

```

    public static void printLine(String s)
    {
        System.out.println("string: " + s);
    }
}

```

This class has two methods called `printLine`: one for printing integer values, one for print string values. The output of method `main` is

```

integer: 3
string: hi

```

---

It is good practice to give methods the same name if they provide similar functionality for different parameter types. However, it is *not* possible to declare two functions that differ just by their return type. In particular, we cannot have two methods

```

int    read();
String read();

```

and therefore must call one method `readInt` and the other one `readString`.

### 5.1.9 Documenting Methods

It is more important to write comment headers than to place comments into method bodies.

The comment header of a method should document the behavior of the method in such a way that the method can be used *without* looking into the body of the method. For this purpose, the method header should express the following pieces of information:

**Input Condition** What condition must the method parameters (and the relevant global variables) satisfy such that it is legal to call the method? It is the responsibility of the *caller* to make sure that this condition holds; the method it is not required to check this condition (but it may do so to be safe).

An input condition is also called a *precondition*; “pre” means “before” (the call).

This is a statement that may mention the method parameters (and relevant global variables).

**Output Condition** What condition do the method's return value and the values of the transient parameters (i.e., any objects referenced by the parameters) and the global variables modified satisfy after the call?

This is a statement that may mention the method parameters (and relevant global variables) and the return value and the “old” value of the transient parameters (i.e., the value before the call).

If a method performs input and output, it is in general necessary to document its behavior also by describing the *external effect* of the method.

The format in which this documentation is given is secondary; one should however stick consistently to a particular style. A more formal way of writing a document header is

```
// -----
// r = method(input, transient)
// One sentence that explains how above statement is to be read.
//
// Input condition:
//   A statement involving the 'input' parameters and the
//   'transient' parameters and the used global variables.
// Output condition:
//   A statement involving the return value 'r' and the
//   'transient' parameters and the used global variables.
// Effect:
//   A statement describing the external effect of the method.
// -----
public static method(type input, type transient)
```

Of course, also other documentation styles are possible; however they should contain the information specified above.

---

**Example** The following header describes division on natural numbers:

```
// -----
// q = div(m, n)
// 'q' becomes the truncated quotient of the natural numbers
// 'm' and 'n'.
//
// Input condition:
//   'm' is not negative, 'n' is positive.
```

An output condition is also called a *postcondition*; “post” means “after” (the call).

```
// Output condition:
//   there exists a remainder 'r' less than 'n' such that
//   'm = n*q + r'.
// -----
public static int div(int m, int n)
```

---

**Example** The following header describes the search for a particular character in a string:

```
// -----
// p = position(s, c)
// 'p' is the index of the first occurrence of character 'c'
// in string 's'.
//
// Input condition:
//   's' is not null.
// Output condition:
//   'p' is greater than or equal to -1 and
//   less than the length of 's'
//   If 'p' is equal to -1, then 'c' does not occur in 's'.
//   If 'p' is greater than -1, then
//   * 's' has at position 'p' character 'c' and
//   * 's' does not have at any position less than 'p'
//   character 'c'.
// -----
public static int position(String s, char c)
```

---

**Example** The following header describes the cutting of a string at a particular position.

```
// -----
// cut(s, n)
// cut string 's' at position 'n'
//
// Input condition:
//   's' is not null.
// Output condition:
//   let 'l' be the length of 's'.
```

```
//  if 'n' is greater than or equal 'l', then 's' is unchanged.
//  otherwise, 's' has length 'n' and has in all positions
//  the same characters as it had before the call.
//  -----
public static void cut(String s, int n)
```

---

**Example** A procedure that performs input/output can be better described by the effects performed than by input/output conditions.

```
//  -----
//  n = readNat()
//  read natural number 'n' from the input stream.
//
//  Output condition:
//  n is greater than or equal to -1.
//  Effect:
//  Asks the user for a non-negative integer number.
//  If such a number is read, it is returned as 'n'.
//  If the input stream has ended, 'n' is -1.
//  Otherwise, the process is repeated.
//  -----
public static int readNat()
```

---

## 5.2 Objects and Classes

A function may have multiple parameters but it may only return one result. Therefore we cannot write a function that say reads a date from the input stream and returns the day, month, and year of this date. However, we can overcome this restriction by packing day, month, and year into a single *object* of type “Date” and returning this object as the function result.

The type of such a compound object is called a *class*. We therefore use the name “class” for two purposes:

1. for a *module*, i.e., a collection of methods (this is how we have always used “class” up to now);

2. for the *type of an object*, i.e., a collection of values (this is a new use of “class”).

We will deal in this section with the second use of the name “class”; later we will learn why it makes sense to use the same name for two seemingly unrelated concepts.

To construct an object, we first have to write a *class declaration* like

```
public class Date
{
    public int day;
    public String month;
    public int year;
}
```

which contains variables also called *fields* or *attributes*. We can then create an object as an *instance* of this class:

```
Date date = new Date();
```

We can write into the fields of the object as in

```
date.day = 24;
date.month = "December";
date.year = 2001;
```

and also read the values in the same way. For instance, the statement

```
System.out.println(date.day);
```

prints

```
25
```

We neglect error checking in this example.

Thus we can write a function that returns multiple results:

```
public static void main(String[] args)
{
    Date d = readDate();
    System.out.println("date: "+d.day+ ". "+d.month+ " "+d.year);
}
```

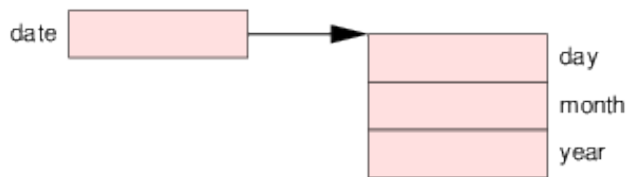


Figure 5.3: An Object

```
public static Date readDate()
{
    Date date = new Date();
    date.day = Input.readInt();
    date.month = Input.readString();
    date.year = Input.readInt();
    return date;
}
```

We may then have a dialogue

```
24
December
2001
date: 24. December 2001
```

The following sections will explain the details of objects and their use.

### 5.2.1 Objects

An *object* is represented in computer memory as a *pointer*, i.e., as the address of a memory area that holds the actual object data (see Figure 5.3). A Java variable that is of an object type therefore *references* the object data but it does *not* hold the data itself.

Correspondingly, if we have an object (referenced by) *name1* and execute a declaration

```
Class name2 = name1;
```

or an assignment

Objects are represented by pointers.

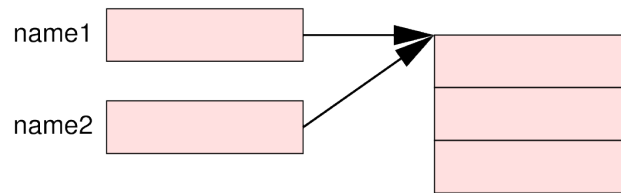


Figure 5.4: Assigning Object Variables

```
name2 = name1;
```

the object referenced by *name1* is not duplicated but referenced by a second name or *alias name2* (see Figure 5.4). Any modification of the object via *name1* is also visible to *name2* and vice versa.

---

**Example** Take the program

```
Date date1 = new Date();  
date1.day = 21;  
Date date2 = date1;  
System.out.println(date2.day);  
date1.day = 17;  
System.out.println(date2.day);
```

After the declaration of *date2*, both *date1* and *date2* refer to the same `Date` object. The later modification of the object via *date1* is also visible in *date2*; therefore the output is

```
21  
17
```

---

If we compare two object variables, we compare, whether they contain the same pointer, not whether they point to objects with identical contents.

---

**Example** Take the program

```
Date d1 = new Date();  
d1.day = 24; d1.month = "December"; d1.year = 2001;  
Date d2 = new Date();
```

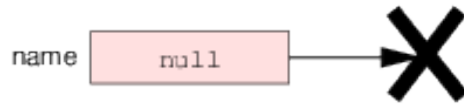


Figure 5.5: The null Reference

---

```
d2.day = 24; d2.month = "December"; d2.year = 2001;
System.out.println(d1 == d2);
d2 = d1;
System.out.println(d1 == d2);
```

Initially, the object variables `d1` and `d2` point to different objects with identical contents. After the assignment of `d1` to `d2`, they point to the same object. The program output therefore is

```
false
true
```

---

---

The special object reference

```
null
```

denotes “no object”, i.e., after an assignment

```
name = null
```

`name` does not refer to any object (see Figure 5.5). Any attempt to access an object field via `name` results in a runtime error.

---

**Example** The execution of the program

```
Date date = null;
System.out.println(date.year);
```

yields the runtime error

```
Exception in thread "main" java.lang.NullPointerException
    at Main.main(Main.java:6)
```

---

---

The value `null` is the default value for object variables (see page 115).

## 5.2.2 Classes

A *class* defines the memory layout for all objects that are instances of this class. For this purpose, the class declaration lists all the fields and their corresponding types as variable/constant declarations:

```
public class Name
{
    public type name = expression;
    public final type name = expression;
    ...
}
```

Please note that we do *not* use the keyword `static` for the declaration of object fields. Names with upper-case capital letters typically denote classes.

For variable declarations, the initialization part may be omitted; in this case, the corresponding object field receives a default value (see page 115) when the object is created. A constant declaration must include the initialization part; after the object has been created, the corresponding object field cannot be changed any more.

If a class has declared a field *name*, we can access the corresponding field value for an *object* of this class by *qualified names*:

```
... = ... object.name ...;
```

Likewise, we can use a qualified name on the left hand side of an assignment to set the value of a field:

```
object.name = ...
```

However, this is only possible if the class has not declared the field as `final`.

The declaration of *Class* is typically written into a separate file *Class.java*. However, we may be also embed it into the source code of a module (please note the keyword `static`):

```
public class Module
{
    public static class Class
    {
        ...
    }
    ...
}
```

In this case, the class is only visible from other modules as *Module.Class*.

### 5.2.3 Object Creation

An object is created by an expression

```
new Class()
```

An object declaration itself does not create the object.

where *Class* denotes the name of the object type. This expression

1. allocates the memory for an object of the denoted class,
2. initializes all object fields with the values specified in the class (respectively with default values, if there were no values specified), and
3. returns a pointer to this object.

It is typically used to initialize an object variable in a declaration

```
Class name = new Class();
```

or to set the value of an object variable by an assignment

```
name = new Class();
```

The expression may also in other places where object references are expected, but we recommend to stick to above cases (see our discussion on program functions versus mathematical functions on page [121](#)).

### 5.2.4 Constructors

It is rather inconvenient to have an object first created and its fields initialized with default values, and then use individual assignments to fill the fields with the actually desired values as in

```
Date date = new Date();  
date.day = 24;  
date.month = "December";  
date.year = 2001;
```

It is possible to achieve the same effect in a shorter way by passing the field values as arguments to the object creation call:

```
Date date = new Date(24, "December", 2001);
```

For this purpose, the declaration of class `Date` must contain a *constructor*:

```
public class Date
{
    public int day;
    public String month;
    public int year;

    public Date(int d, String m, int y)
    {
        day = d;
        month = m;
        year = y;
    }
}
```

When an object is created, a constructor is called.

A constructor is a special method that has the same name as the class in which it is declared (please note that we do not use the keywords `static void` for constructors). This method is bound to an object of this class in the sense that it can read and write the fields of the object. The system automatically provides a *default constructor* which initializes the object variables to the default values (up to now, we have only used the default constructor). If a constructor is declared by the user, the user constructor is always called *after* the default constructor; thus the field values have been already initialized with the default values when the constructor body is executed. If the user constructor is declared with parameters, it may be invoked in object creation with corresponding arguments (analogously to procedure calls).

A constructor may also be used to initialize a constant field.

---

**Example** Take the class

```
public class Date
{
    public int day;
    public String month;
    public final int year;
```

```
public Date(int d, String m, int y)
{
    day = d;
    month = m;
    year = y;
}
}
```

where the constant *year* receives its value from the constructor. It is not possible to modify the constant after it has received a value by the constructor. For instance, the statement

```
Date date = new Date(24, "December", 2001);
date.year = 2002;
```

gives the compiler error

```
Main.java:17: cannot assign a value to final variable year
    date.year = 2002;
            ^
```

---

A constructor may read the special variable *this* which refers to the current object. This becomes handy if parameter names shadow field names.

---

**Example** We can write above program as

```
public class Date
{
    public int day;
    public String month;
    public int year;

    public Date(int day, String month, int year)
    {
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
```

We recommend this style rather than inventing artificial parameter names.

---

Because of overloading (see page 125), we may have multiple constructors in a class provided that they differ in their parameter lists.

---

**Example** In above program, we may have another constructor

```
public class Date
{
    ...
    public final int THIS_YEAR = 2001;

    public Date(int day, String month)
    {
        this.day = day;
        this.month = month;
        this.year = THIS_YEAR;
    }
}
```

We then have two kinds of constructor calls:

```
Date date1 = new Date(31, 12);
Date date2 = new Date(31, 12, 1999);
```

---

If there is a user-defined constructor, the default constructor cannot be used any more in object creation calls; if the user still wants a call `new Class()`, she has to provide a corresponding constructor (whose body may be empty).

A constructor may call another constructor by using *as its first statement* a statement of the form

```
this(expressions);
```

This statement invokes the constructor `Class(expressions)` rather than the default constructor before the remaining body of the constructor is executed.

---

**Example** Above program may be also written as

```
public class Date
{
    ...
    public final int THIS_YEAR = 2001;

    public Date(int day, String month)
    {
        this(day, month, THIS_YEAR);
    }
}
```

---

### 5.2.5 Transient Parameters

We have seen that objects can be used to simulate multiple return values. They can be also used to simulate a kind of parameters for which there is no direct support in Java, namely transient parameters. A *transient parameter* is an argument variable whose value is to be updated by a method call, i.e., the variable has a value before the method is called and it shall have a different value after the call. We have already seen in the example on page 117 that an assignment to a parameter in the method body has no effect on the argument variable, therefore it is not possible to implement a method

```
public static void increment(int i, int n);
```

that increases the value of integer *i* by *n*.

However, we can encapsulate a variable into an *object* and pass as an argument a pointer to the object. The method parameter therefore becomes an alias (see page 132) of the argument variable; any modification of the object by the alias is also visible to the argument.

---

**Example** Take the class

```
public class Integer
{
    public int value;
    public Integer(int value)
    {
        this.value = value;
    }
}
```

“Transient” means “passing by”.

An instance of this class encapsulates an `int` value. We define the method

```
public static void increment(Integer i, int n)
{
    i.value = i.value + n;
}
```

The execution of

```
Integer i = new Integer(5);
increment(i, 3);
System.out.println(i.value);
```

gives output

```
8
```

---

---

## 5.2.6 Structuring Programs

As stated in Section 4.9.3, most programs are not written as monolithic pieces of code; they are composed from multiple methods that provide part of the functionality to achieve the overall goal. These methods typically correspond to the gradual refinement steps that we perform in the development process; rather than inserting into the main program the pieces of code that correspond to these refinements, we put them into separate methods such that the final program is an image of the refinement process.

We will now show how methods and objects can be used to give a better structure to the program *analyze exams* presented in Section 4.9.3. The overall functionality of the program is provided the following method (please compare with the original code):

```
public static void analyzeExamGrades()
{
    char answer = 0;
    do
    {
        Result result = analyzeExam();
        printResult(result);
        answer = askToContinue();
    }
    while (answer == 'y');
}
```

We introduce a local variable *result* of the (to be declared) object type *Result*; this variable will hold the result of the analysis which is to be printed.

Asking the user whether to continue becomes the following method:

```
public static char askToContinue()
{
    while(true)
    {
        System.out.print("Another exam (y/n)? ");
        char answer = Input.readChar();
        if (Input.isOkay() && (answer == 'y' || answer == 'n'))
            return answer;
        if (Input.hasEnded())
        {
            System.out.println("Unexpected end of input!");
            System.exit(-1);
        }
        System.out.println("Invalid input!");
        Input.clearError();
    }
}
```

We use a block variable *answer* to hold the answer of this user; if the answer is valid, we use a return statement to return this answer to the caller.

The analysis of the exam becomes

```
public static Result analyzeExam()
{
    Result result = new Result();

    // read and process exam
    while (true)
    {
        int grade = readGrade(result.count);
        if (grade == 0) return result;
        processGrade(grade, result);
    }
}
```

Since this method has not much else to do, we keep the functionality of *read and process exam* within *analyzeExamGrades* rather than delegating it to a separate method. Furthermore, rather than declaring individual variables *count*, *sum*, *min*, and *max*, we pack these values into an object of class *Result* declared as follows:

```
public class Result
{
    public int count = 0;           // number of grades
    public int sum = 0;            // sum of grades processed so far
    public int min = Integer.MAX_VALUE; // minimum grade processed so far
    public int max = Integer.MIN_VALUE; // maximum grade processed so far
}
```

The reference *result* to this object is passed to the following method which updates the object to incorporate a new grade.

```
public static void processGrade(int grade, Result result)
{
    result.count++;
    result.sum += grade;
    if (grade < result.min) result.min = grade;
    if (grade > result.max) result.max = grade;
}
```

Please note that *result* is here a transient parameter (see page 139) which references the object that holds the actual data. The method `analyzeExam` references this object by its local variable *result* and updates this object iteratively by calls of `processGrade`. After the last grade has been processed, `analyzeExamGrades` returns the object to the caller.

Reading a grade is achieved by the following method which takes the running number of the grade read as a parameter:

```
public static int readGrade(int count)
{
    while(true)
    {
        System.out.print("Enter grade "+(count+1)+" (0 when done): ");
        int grade = Input.readInt();
        if (Input.isOkay() && 0 <= grade && grade <= 5)
            return grade;
        if (Input.hasEnded())
        {
            System.out.println("Unexpected end of input!");
            System.exit(-1);
        }
        System.out.println("Invalid input!");
        Input.clearError();
    }
}
```

Again, we use a return statement to return the validated grade to the caller.

Finally, printing the results of the analysis is achieved by

```
public static void printResult(Result result)
{
    System.out.println("\nNumber of grades: " + result.count);

    // these values are only valid, if there was at least one grade
    if (result.count > 0)
    {
        System.out.println("Best grade: " + result.min);
        System.out.println("Worst grade: " + result.max);
        System.out.println("Average grade: " +
            (float)result.sum/(float)result.count);
    }

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

The complete program then has the following structure

```
// -----
// Exams2.java
// analyzing the grades of exams
//
// Author: Wolfgang Schreiner <Wolfgang.Schreiner@fhs-hagenberg.ac.at>
// Created: August 23, 2001
// -----
public class Exams2
{
    // -----
    // analyze the grades of multiple exams; read the grades from the
    // std input stream, writes the result to the std output stream.
    // for details, see 'analyzeExamGrades'.
    // -----
    public static void main(String[] args)
    {
        analyzeExamGrades();
    }

    // -----
    // the (intermediate) result of analyzing the grades of an exam
    // -----
    public static class Result { ... }
```

```
// -----  
// user enters grades from 1 to 5 or 0 to denote the end of exam.  
// system prints number of grades, best, worst and average grade and  
// asks whether to proceed with another exam.  
// -----  
public static void analyzeExamGrades() { ... }  
  
// -----  
// iteratively reads and analyzes the grades of an exam.  
// when a grade 0 is read, the result of the analysis is returned.  
// -----  
public static Result analyzeExam() { ... }  
  
// -----  
// asks the user for grade number 'count' (starting with 0).  
// if the user enters an integer from 0 to 5, this is returned as  
// the result. otherwise, the user is asked again.  
// aborts execution, if the input stream has terminated.  
// -----  
public static int readGrade(int count) { ... }  
  
// -----  
// processes 'grade' and updates the intermediate 'result'  
// -----  
public static void processGrade(int grade, Result result) { ... }  
  
// -----  
// prints the result of an analysis  
// -----  
public static void printResult(Result result) { ... }  
  
// -----  
// asks user whether to continue. if answer is 'y' or 'n', this  
// is returned as the result. otherwise, the user is asked again.  
// aborts execution, if the input stream has terminated.  
// -----  
public static char askToContinue() { ... }  
}
```

In this way, we have decomposed the problem solution into multiple methods which perform independent functionalities. You can download the source code of the application to compile and run it.

# Appendix A

## Software Tools

In this appendix, we describe the basic installation and configuration of the software tools used in this course (for computers running the Microsoft Windows operating system; the installation on Mac OS X computers should proceed in a similar way). We will

1. first use the Oracle Java SE and the TextPad editor as basic tools, and
2. then switch to the Eclipse IDE for Java Developers as an integrated development environment.

### A.1 Oracle Java SE

To install the Oracle Java SE (Standard Edition) Platform, go to

<https://www.oracle.com/technetwork/java/javase/downloads>

and select the entry “JDK Download”. Select the option “Accept License Agreement” and download the version of the JDK appropriate for your system (for Windows x64, this is currently the file `jdk-18_windows-x64_bin.exe`). After downloading, click on this file to start the setup. When prompted, press “Next”, and finally “Close”, then the installation is completed.

Start from the search field in the Windows “Start” area the command `cmd` to get a command line shell. You should then be able to type `java -version` respectively `javac -version` and get an output message similar to

```
java version "18.0.1.1" 2022-04-22
Java(TM) SE Runtime Environment (build 18.0.1.1+2-6)
Java HotSpot(TM) 64-Bit Server VM (build 18.0.1.1+2-6, mixed mode, sharing)
```

If this is the case, you are done. However, if you get an error message that the command could not be found, you have to set the environment variable `Path` to point to the executables `java` and `javac`. For this, type in the Windows search field `env`, then select from the search results the entry “Set Environment Variables”, then click in the tab “Extended” the button “Environment Variables”. Select the variable “Path” and press the button “Edit” which will open a new tab “Edit environment variable” to enter the path of the directory containing the executables.

You can find this path by selecting in the Windows Explorer the corresponding directory, for example “C:”, “Program Files”, “Java”, “jdk-18.0.1.1”, “bin”. Click on the right of the location bar on top of the Windows Explorer and you will see the whole path as a single string:

```
C:\Program Files\Java\jdk-18.0.1.1\bin
```

Copy this string with `Ctrl+C`. Press in the “Edit environment variable” tab the button “New”, paste into the new line with `Ctrl+V` the copied string, and then press “Okay”. Open a new command line shell and try again to enter `java -version` respectively `javac -version`, which should now work as described above.

## A.2 TextPad

To install the free programming editor TextPad, go to

```
https://www.textpad.com
```

and select the entry “Download”. Follow the instructions to download and install TextPad.

In “Configure/Preferences/Document Classes/Java/Tabulation” select the options “Convert new tabs to spaces” and “Convert existing tabs to spaces when saving files” and set both the “Default tab spacing” and “Default indent size” values to 2. In this way, portable source files will be created whose visual appearance does not depend on the tabulator width set by a particular editor.

When using TextPad, do not use non-ASCII characters in your source code (if you need such characters in string literals, use the Java Unicode escape character

\udddd instead). It is extremely complicated to have such characters treated correctly when using console input/output (apart from making sure, that the source file is indeed saved in UTF-8, one has to make sure that the console uses an UTF-8 capable font and that the character encoding of the console is configured correctly). This restriction does not apply any more, as soon as we use Eclipse as shown in the next section.

For compiling Java source files within TextPad, select “Configure/ Preferences/ Tools/ Compile Java” and enter as “Parameters”

```
-cp ".;U:\KWM101\kwm.jar" $File
```

Here the path U:\... has to be updated to point to the location where you have installed the kwm.jar file. Make sure that the option “Capture output” is selected (otherwise no compiler output will be shown within TextPad). When editing an .java file, it is now possible to start the compiler by selecting the menu entry “Tools/Compile Java” (respectively by the corresponding key shortcut Ctrl+1).

For also starting Java programs within TextPad, select “Configure/ Preferences/ Tools/ Run Java Application”, and enter as “Parameters”

```
-ea -cp ".;U:\KWM101\kwm.jar" $BaseName
```

Again the path U:\... has to be updated to point to the location where you have installed the kwm.jar file. Make sure that the option “Capture output” is *not* selected (otherwise command line input will not work). Having compiled a .java file, it is now possible to execute the generated .class file by selecting the menu entry “Tools/Run Java Application” (respectively the key shortcut Ctrl+2).

If the “Tools” entries “Compile Java” and “Run Java Application” should not exist, you can create corresponding entries on your own. For this, select “Configure/ Preferences/ Tools/ Add/ Program”, choose the path of the javac program, e.g.

```
C:\Program Files\Java\jdk-18.0.1.1\bin\javac.exe
```

and press “Apply”. Then you have under the header “Tools” the new entry “javac” that can be configured and used like the entry “Compile Java” explained above. Furthermore, select “Configure/ Preferences/ Tools/ Add/ Program”, choose the location of the java program, e.g.

```
C:\Program Files\Java\jdk-18.0.1.1\bin\java.exe
```

and press “Apply”. Then you have under the header “Tools” the entry “java” that can be configured and used like the entry “Run Java Application” explained above.

## A.3 Eclipse IDE

The Eclipse IDE is an integrated environment for developing Java applications. Go to

<https://www.eclipse.org>

press “Download” and then the “Download” button under the “Get Eclipse” header and then again the “Download” button. Download and start the installation program and select in the popup menu “Eclipse IDE for Java Developers”. This will create a desktop link to Eclipse as well as an entry in the start menu.

When you have started Eclipse, you may (after clicking on the “Workbench” icon of the “Welcome Page”) configure your editing preferences. For this, select the menu entry “Window/Preferences” and then select the entry “Java/Codestyle/Formatter”. Select “Edit” to edit the standard code formatting profile; select the tab “Indentation” and then set the “Tab policy” to “Spaces only” and the values of “indentation size” and “tab size” to both “2”. You have to save the profile under a new name.

To create a new project for building a Java application, select “File/New/Java Project” and *unselect* the option “Create module-info.java file”. After the creation of the project, you must configure it appropriately:

**Compiling and Linking** Go to the “Properties” dialog of the project (right-click the project symbol), then select “Java Build Path”, “Libraries” and “Class-path”. Use the button “Add External JARs” to select the file `kwm.jar` which provides the input/output interface for the Java applications developed in this course (and should be installed in some location of your home folder independently of the project).

**Executing** After you have created a Java file with a `main` method, save the file, click on the editor tab with the file and press the green “Run” arrow to execute the program. Pressing this arrow for the first time automatically creates a runtime configuration for your program. You may later edit this configuration by the menu entry “Run/Run Configurations...”. Select the runtime configuration for your project, pick the “Arguments” tab and set the “VM arguments” options to `-ea` (this enables assertion checking).

Afterwards, you can always compile your program just by saving the file and then start it by clicking the green “Run” arrow.

# Literature

The following books were used in the preparation of this course:

**Hanspeter Mössenböck** *Sprechen Sie Java? — Eine Einführung in das systematische Programmieren*. 5. Auflage, dpunkt.verlag, Heidelberg, Germany, 2014.

Based on the lecture notes of a course on Java programming for computer science students at the Johannes Kepler University of Linz (330 pages). This book is less a book about Java than about programming in general (still all of the basics of Java are introduced). Does not focus on object-oriented programming (most of the programs are written in an imperative style) but more on the systematics of program design. Compact and easy to read.

**John Lewis and William Loftus** *Java Software Solutions — Foundations of Program Design*. 7th edition, Addison-Wesley, Reading, Massachusetts, 2011.

A good introduction to learning programming in Java (800 pages with CD-ROM) supplemented by a Web site with examples and sources. Starts earlier with true object-oriented programming than we do in our course, focuses a bit more on technical details of Java, and discusses a bit less basic programming principles. A lot of material, easy to read.

# Index

actual parameters, 117  
alias, 132  
arguments, 117  
arithmetic promotion, 50  
arithmetic/logic unit, 25  
ascii, 21  
assembler, 27  
assertion, 66  
assignment, 43  
attributes, 130

block statement, 67  
body, 34  
bug, 104  
byte-code, 31

cache, 25  
character literal, 54  
characters, 21  
class, 129, 134  
class file, 31  
comments, 36  
compiler, 29  
computed conditionals, 77  
conditional statement, 66  
connectives, 46  
constant, 44  
constructor, 136  
control flow, 65  
control unit, 25

dangling else, 73  
decrement operator, 91  
default value, 115

digital, 21  
division by zero, 48  
do loop, 86

english, 37  
equality operations, 47  
escape character, 38  
escape sequence, 38, 54  
exponent, 56  
exponentiation, 81, 93

fields, 130  
floating point, 55  
for loop, 89  
formal parameters, 116  
function, 119  
function application, 120

global, 114

hardware, 5  
hidden state, 123

identifier, 34  
if, 70  
if statement, 69  
if-else statement, 71  
increment operator, 89  
indentation, 36  
infinite, 66  
input condition, 126  
instance, 130  
instructions, 25  
integer literal, 48  
integer overflow, 50

- interpreter, 28
- invokation, 109
- iso 8859-1, 22
- iteration statement, 66
  
- java virtual machine, 31
- jit, 31
- just in time, 31
  
- labels, 83
- linker, 29
- local, 113
- loop, 66
- loop body, 79
- loop condition, 79
- loop invariant, 81
  
- main memory, 20
- mantissa, 56
- memory address, 21
- memory cells, 21
- memory content, 21
- method, 34
- method body, 109
- method call, 109, 116
- method declaration, 108
- method head, 116
- method name, 109
- methods, 108
- module, 129
- modules, 29
- multi-branch conditional, 75
- multi-branch conditionals, 75
  
- narrowing conversion, 52
- nested loops, 83
- non-printable, 22
  
- object, 131
- object file, 29
- one-branch conditional, 69
- output condition, 127
  
- overloaded, 125
  
- parameter, 116
- parameter declarations, 116
- pointer, 131
- portable, 30
- porting, 30
- postcondition, 127
- precondition, 126
- primitive data types, 46
- procedure, 119
- processor, 20
- program counter, 25
- programming language, 5
  
- qualified names, 134
  
- random access memory, 21
- random number generator, 123
- references, 131
- registers, 25
- relational operations, 53
- result, 141
- return, 119
- return statement, 111
- return type, 119
  
- shadows, 124
- short-circuited, 47
- side effect, 121
- sign, 56
- software, 5
- software engineering, 6
- source code, 29
- specification, 5
- standard input stream, 64
- standard output stream, 37
- statements, 37
- stepwise refinement, 106
- string literal, 60
- switch statement, 77
  
- target code, 29

termination condition, [66](#)  
tokens, [36](#)  
transient parameter, [139](#)  
two-branch conditional, [71](#)  
type cast, [52](#)  
  
unicode, [22](#)  
  
variable, [42](#)  
variable declaration, [42](#)  
visible, [124](#)  
void, [119](#)  
von neumann architecture, [26](#)  
  
while loop, [79](#)  
widening conversion, [50](#)