Object Oriented Programming – with Java

Kees Huizing Ruurd Kuiper

September 10, 2009

Preface

Computers are used almost everywhere. Their role is processing data. A computer can perform any data processing concern we desire if we provide it with an appropriate description, a program. Executing the program on a computer then results in performing the desired data processing concern.

Programs and their execution can easily become very complex. Therefore, structuring is necessary to make it feasible for humans to develop programs. Object Oriented Programming (OOP) is a powerful and widely used programming paradigm that supports structuring. Java is a representative, modern, OOP language.

This book explains OOP and Java.

The approach

No previous knowledge about programming is assumed.

One OOP methodology and style are explained rather than going into alternatives; especially for the novice programmer it is more helpful to be shown one consistent approach than to have to make uninformed choices. Java is explained and motivated as an OOP language. Knowledge is build up incrementally, without branching or making de-tours, chapters generally depending on previous ones. The emphasis is on concepts rather than on details. This "narrow path" approach provides a fast route to understanding OOP and Java for the conceptually inclined traveler.

The larger structure of the book is in parts, corresponding to programming paradigm issues.

The parts contain many small chapters that each consider one programming concept.

The chapters have a fixed structure.

A section **Aim** makes the programming aim explicit. A (learner) programmer is driven by programming aims, and has to (learn to) find solutions corresponding to the aims.

A section **Means** provides the corresponding programming concept. An example illustrates the concept that is being introduced.

A section **Execution model** incorporates the new concept in the execution model. This model is more detailed and more explicit than is usually provided. In OOP one's thinking often moves between the static, fixed, description in classes and the dynamic objects, processing data (significantly, the paradigm is named Object Oriented Programming rather than Class Oriented Programming). Thinking at the static level is supported by the language facilities. Thinking at the dynamic level is less directly and explicitly supported. The execution model remedies this, as it represents the dynamic behavior of objects that results from executing the static code of the classes. Such a model is useful to aid one's thinking about programming, e.g., to explain programming concepts; it also is helpful in discussions about programming issues if the participants have an explicit, shared, model in mind. A basic model is provided from the outset, which is extended when new concepts require so.

In some cases, the treatment of a concept involves further subdivision; then subsections are used, maintaining the structure as just outlined.

Where needed, a **Remarks** section is added.

Exercises are provided, separately, for each chapter. To learn OOP, it is necessary to do a substantial amount of the exercises, preferably after reading the corresponding chapter, before moving on to the next one. This, and explorations by yourself, will make you an amateur programmer; professional programming requires further education and training.

The book and the exercises are available on **??**. Also, an execution model viewer, called *CoffeeDregs*, a piece of software that visualizes the execution model on screen for concrete programs, is available there.

Additional web sources

The "narrow path" approach may require consulting other sources for further detail. Web sources for additional information are the following.

- For explanations about Java programming in general, see the SUN Java tutorials: `http://java.sun.com/docs/books/tutorial/`
- For a complete description of the Java language, see The Java Language Specification, third edition: `http://java.sun.com/docs/books/jls/`
- For information about predefined Java code, see the predefined classes and interfaces in the J2SE version of the Application Programmers Interface (API): `http://java.sun.com/javase/6/docs/api/`
- A good general book on Java is *Learning Java*, by Niemeyer and Knudsen, O'Reilly, 2005.

Preliminaries

To write, execute and, in the approach of this book, visualize Java programs, the appropriate software needs to be present on the computer. It is assumed that the reader has some basic knowledge about downloading and installing software and about using an editor. The software mentioned below is available for free. Version for Windows, Unix, Linux, and Macintosh platforms are available.

- To conveniently write Java programs an Integrated Development Environment (IDE) needs to be present. Various IDEs are available, the book does not depend on a particular one, all provide the facilities needed. A suitable IDE is *NetBeans*, available from http://www.netbeans.org/ .
- To execute Java programs, a run-time environment needs to be present. The Java Development Kit, JDK6, provides this; it is available from http://java.sun.com/ .
- To visualize the execution of Java programs, the model viewer, *CoffeeDregs*, needs to be present. *CoffeeDregs* is available from JavaBookWebsite .

Contents

Part I

Basic programming – One class, one object

Chapter 1

Program and execution – Class and object

A *computer* is a machine that stores and manipulates *data values*, like numbers or text, it performs a *data processing concern*. A computer can perform any data processing concern we desire, if we provide it with an appropriate description.

1.1 Aim

We want the computer to perform different data processing concerns, of our choice and design.

1.2 Means

We develop in the programming language Java a *program* that contains a *class* with the description of the data processing concern. We *execute* the program on a computer, which results in an *object* in the computer that performs the data processing concern.

Java is a language for *Object Oriented Programming* (OOP), a programming paradigm that is motivated and explained during the exposition in this book. Java is a "real world" language, not primarily designed for teaching purposes, so sometimes features show up at times that they cannot yet be appreciated; in such cases explanation is deferred till later. "Class" and "object" are like that: they make an immediate appearance and are essential OOP concepts, but in Part I are only exploited in a limited fashion.

1.2.1 Writing the source code

Program text is called *source code*, or simply *code* – often such somewhat less specific terms are used. To be executable, code must follow the Java *syntax*, a concise set of words and symbols, and grammatical rules. Also *coding conventions* are introduced that, unlike the syntax rules, are not binding but are strong suggestions that improve the readability and other quality aspects of the code. The conventions reflect generally accepted practice.

In programming, the computer plays a double role: apart from its main purpose, performing the execution of a program, it also supports writing the program. Such software is called an *IDE* (Integrated Development Environment). Examples of IDEs are: NetBeans, Eclipse (supported by IBM), JCreator. We assume that an IDE has been installed. Amongst other things, an IDE provides an *editor* which we use to write the program.

As an example, we write a small program to put a text message on the screen.

The desired output is:

```
Hello world, this a Java program.
Good luck, programmer!
```

We write the code stepwise, introducing and explaining the syntax. The program, in this simple case, contains just one class, which we call `HelloWorld`. It is the container for the code for the data processing concern.

We store the code in a file with, by rule, the same name as the class and with the extension .java; in our case HelloWorld.java.

```
public class HelloWorld { // class header (class declaration)
                          // here the class body will be written
}
```

The code of a class consists of a class *header* and a class *body*. The header states that the description following is a class and provides the class name; this is called the *declaration* of the class. By convention, the first letter of a class name is in uppercase. Java is case sensitive, e.g., names that differ in just the use of an upper- or lowercase will refer to different entities. `public` in the class header has to be there – explanation deferred.

Curly brackets { and } mark the place where we will write the class body. The text following //, on the same line, is *comment* and ignored by the computer. Comments are added to code to help humans understand that code. In the examples comments are also used didactically, e.g., to explain newly introduced programming features. Hence in the examples sometimes more comment may be present than is usual practice.

In this simple case only code for data manipulation is present. Later a class will be seen to contain more than just such code; this code for manipulation therefore is placed in a sub-container, a *method*. We call the method `show`.

```
public class HelloWorld {

    void show() { // method header (method declaration)
                  // here the method body will be written
    }
}
```

The text that describes a method also consists of a header and a body. By convention, method names start in lowercase. `void` before the method name and the parentheses `()` in the header have to be there – explanation deferred.

The new bracket pair marks the place where we will write the method body.

Finally, we write the code for the method body, that describes putting the message on the screen.

The code to achieve this will consist of *command statements*. A statement describes an instruction for the computer to perform. Statements end in a `;` (semicolon). The order in which such statements occur in the method is the order in which they will be executed.

```
public class HelloWorld {

    void show() {
        System.out.println("Hello world,this is a Java program."); // output and newline statement
        System.out.println("Good luck, programmer!"); // output and newline statement
     }
}
```

In this example we use only one type of output statement, `System.out.println...`. It describes to output a line of text in a fixed text window on the screen and to move the output-cursor to the next line on output, i.e., possible further output is put on the next line. The window is called the *console*; this kind of output is called *console output*. Between the parentheses is data provided to the output command, i.e., the text to be output. Text in Java must be placed between double quotes, `"` and `"`. So we use the command to print, `System.out.println...`, twice, with different output text. We use indentation for readability, it does not influence the workings of the program. We now have written in the class the description of the data processing concern that we aimed for.

However, this code cannot be executed directly. We also have to describe explicitly to *create* an *object* and to *call*, i.e., start, its method. This we do, in a standard manner and a standard form, by adding the following special piece of start-up code to the class.

```
public static void main(String[] args) { // marks start−up code
    new HelloWorld().show(); // create object, start method
}
```

For now, the relevant part of the start-up code is the statement `new HelloWorld().show();`, which in fact describes two things. The `new HelloWorld()` part describes to create, in the computer, an object according to the description of the data processing concern in the class `HelloWorld`. This object contains an executable form of the method, `show`. The `.show()` part, a *method call*, describes to start the execution of the method `show` of the object.

`new HelloWorld().show();` has to be executed as the first activity of the program, before the object exists: its very purpose namely is to get things off the ground, i.e., to create the object and to start the method. To make this happen, `new HelloWorld().show();` is contained in a special piece of code, the so-called `main`, that signals that its contents should be executed first. Why this code looks as it does will be explained (much) later.

If we write another class for another data processing concern, the name of that class and the code in the body will be completely different, but the shape of the code in the start-up part remains the same, with only the name of class and method being different.

By convention, we put the start-up code at the end of the class. Also by convention, we include a comment about the purpose of the class right before the class code and a comment about the effect of the method just before the method; we also indicate the start-up code by a comment. The complete class, with the didactic comments omitted, is then as follows.

```
// demonstrates console output
public class HelloWorld {

    // puts message on console
    void show() {
        System.out.println(”Hello world, this is a Java program.”);
        System.out.println(”Good luck, programmer!”);
    }

    // start−up code
    public static void main(String[] args) {
        new HelloWorld().show();
    }
}
```

1.2.2 Checking the source code for syntactic correctness

To be executable, a program has to be *syntactically correct*, i.e., follow the Java syntax. Apart from an editor, the IDE provides a *compiler* to check for syntax errors. Having the file containing the program code

open in the editor, we *compile* the code (available via a menu option or button with a name such as Compile File. As a result, information about syntactic errors in the code appears in a special window. After we have corrected the errors, we compile the code again. Compilation and correction have to be repeated until no errors are reported anymore.

1.2.3 Translating the source code to byte code

To be executable, the human-readable Java code that we have written needs to be *translated* into machine-executable code, so-called *byte code*. The compiler also provides this translation: when no syntactic errors occur in the code, compilation automatically produces the machine-executable code. The byte code is, automatically, stored in a file with the same name as the class but with extension .class; in our example HelloWorld.class. Usually, this file is, again automatically, placed in the same folder as the file with the Java code. When we work with an IDE, we hardly notice the machine-executable code issue; it is only mentioned here to explain the occurrence of the .class file.

1.2.4 Executing the byte code

Software that enables to execute programs is called a *run-time environment*. We assume that a run-time environment has been installed.

We start the actual execution by choosing the command (menu option, button, etc.) Execute (or a similar name).

Now the object is created and the method `show` is called and starts executing the command statements in the order that they occur in the code.

Output is:

```
Hello world, this is a Java program.
Good luck, programmer!
```

1.3 Execution model

In the previous section, two issues about programming showed up. First, that a program is a description of how a data processing concern should be performed. The program is syntactic, static, it does not change during the execution. Second, that a program can be executed, which makes things happen inside the computer. In particular, an object is made from the class. The goings-on inside the computer are the meaning or *semantics* of the program. The semantics of a program is dynamic, it captures the changes inside the computer over time.

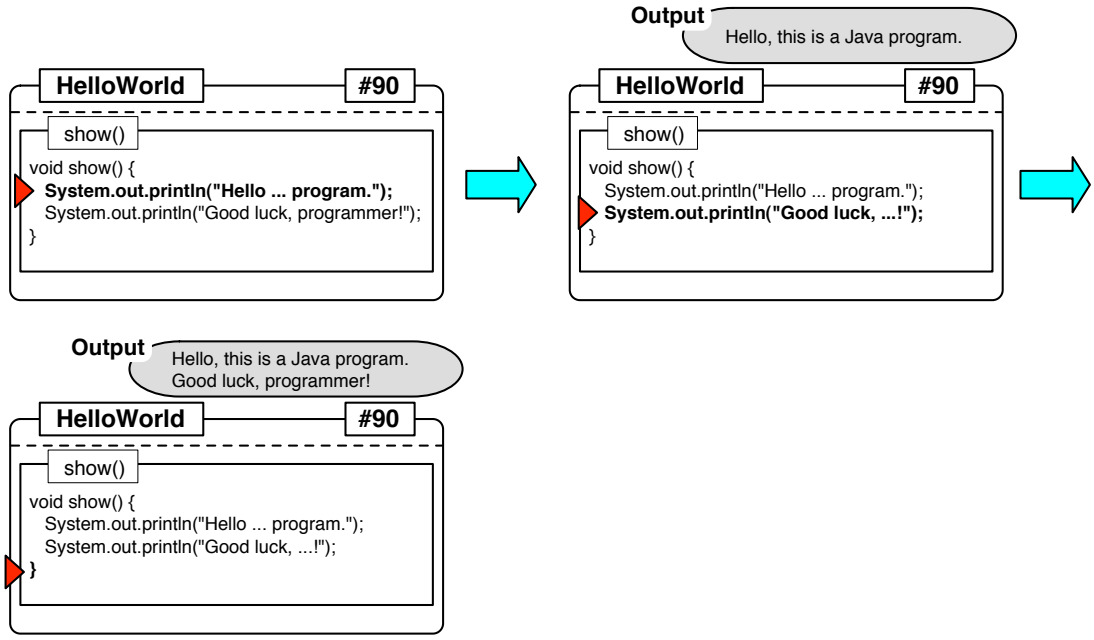
To write good programs, it is important to have a clear understanding of the semantics of a program. However, it is not needed or even helpful to describe in detail, say electronically, what goes on inside the computer. Therefore, a conceptual, pictorial representation of the semantics is given, called the *execution model*.

The execution model pictures the execution as a sequence of snapshots, called *states*. A state represents the relevant information about what is present inside the computer at one moment in time.

Figure 1.1 pictures the execution of `HelloWorld`.

Modeling the execution of the start-up part is deferred till later. Thus, the execution sequence starts with the result of the start-up phase, an object that is labeled with the class name, `HelloWorld`, and a number, #90 in this case. (This number identifies the object, when we have several objects, each object will have a different number. The actual value is chosen by the Java Machine and can not be used or changed by the

Figure 1.1: Execution of `HelloWorld`



program.) The object contains the method `show`. There is a dotted line above the method; later an object will contain other things than methods as well, that will be placed above the dotted line.

The computer executes one instruction at the time, namely the one where the so-called *control* is at. This information is also part of the state. The statement to be executed next is highlighted and marked with a cursor symbol. The main difference between the code in the class and the text in the dynamic object is the presence of control. Note that the start-up code is not present at all in the object.

Output is not part of the state, it appears on the console. For clarity, it is pictured in a separate box.

The pictures shows a sequence of execution states, beginning immediately when the start-up code is finished. For `HelloWorld`, this is when control is just before the first line of the body of `show`. In the next state, the effect of executing the instruction that was indicated by control is shown. In this case, `Hello world, this is a Java program.` is put on the output console. Furthermore, control has moved to the next, second, line of the body of `show`. This movement of control is called the *control flow*.

In the subsequent state, similarly `Good luck, programmer!` is put on the console. Control then moves to the last line of the object.

1.4 Remarks

1.4.1 The Java Language Specification

The book provides the essential syntactic and semantic concepts of Java. This entails that not all detailed language features will always be treated in full.

For example, there is another output command, `System.out.print` rather than the `System.out.println` that we used, that does not advance the cursor on the console output window to a new line. This statement can be used to split one line of output over two output statements, for instance to avoid having lines in the

source code that are longer than the editor screen is wide.

Furthermore, when `"` occurs in the output text itself, i.e., is desired to be output, it has to be indicated that it is text, and not to be interpreted by the computer as marking the beginning or ending of the text. This is achieved by placing a \ in front of the `"`.

Such details are not always considered in the book. A complete description of both the Java syntax and semantics can be found in The Java Language Specification [1], provided by and made available on-line by SUN Microsystems, the developers of Java. It can be consulted when additional information is needed. The descriptions given there can only be fully understood by someone who has knowledge about programming, i.e., the information in [1] should become more appreciable when progressing through this book.

1.4.2 Byte code and the Java Virtual Machine

For most languages, the executable code that the compiler produces is so-called *machine-code* code that is directly executed by the computer. For different types of computers, different machine-code has to be produced.

In the case of Java, the compiler produces not machine-code but *byte-code*, which is an intermediate level of code that is not executed by the computer directly, but by an intermediate program, the *Java Virtual Machine* (JVM). The byte-code is the same for all types of computer.

This standardized intermediate ground between source code and computer has advantages for *portability*, the extend to which programs can be executed by different machines, and for *security*, the protection of the machine against, e.g., viruses. Both issues are important for web use of programs. A price to be paid for these advantages is that the JVM approach makes programs less efficient.

As stated above, in section 1.2.3, we hardly notice the machine-executable code issue; this additional information is merely provided because one sometimes may easily encounter the terms byte code and JVM elsewhere.

1.4.3 Programming without a specific IDE

Rather than using an IDE to support programming, it is also possible to directly program. A program can be written using any editor. The Java JDK runtime environment then provides a compiler and enables to run the byte code. The command to compile source code in, say, a file `program.java` to byte code is `javac program.java` which does the syntax check and yields the file `program.class` containing the source code. The command to execute the byte code stored in `program.class` is `java.program`.

Chapter 2

Data operations – Built-in data types

In the previous chapter it was shown how to output fixed textual data values, namely strings of characters. An essential part of data processing is to calculate new data values from given ones. The basic language elements to do this are introduced now.

2.1 Aim

We want to calculate new data values from given ones.

See chapter 1.

2.2 Means

We use the Java *data types*. A data type provides syntax (notation) to describe calculations and semantics to let the computer perform these calculations. There are several such data types because there are several kinds of data values, e.g., numbers and text, with corresponding operations. With the syntax of data types we can build *expressions*, formulas the computer has to calculate.

The syntax of a data type is as follows.

1. *Literals*, a notation for data values, like `7` and `17.3`; literals are *simple expressions*.
2. *Operators*, like `+`, that take *arguments* to form *compound expressions*, like `7 + 17`. The arguments of an operator can be literals, but also compound expressions, possibly using parentheses; thus, expressions can be complex formulas, like `-(7+5*(-2+17)-8/4)`.

Expressions describe data manipulations, and hence are written in the body of the method of a class.

The semantics of a data type is as follows.

1. The semantics of a literal is a *value* in the data type, e.g. a number; inside the computer it is represented as some sequence of bits.
2. The semantics of an operator is an *operation* that takes values as arguments and produces a new value; inside the computer this is represented as producing a new sequence of bits. The semantics of an expression is the resulting value produced by successively applying the operations on (intermediate) values, so called *evaluation* of the expression; e.g., the expression `7 + 2 * 17` evaluates to the value `41`.

In general, expression evaluation follows the rules that we know from arithmetic etc. Like in arithmetic, parentheses may be used to influence the order of evaluation. The precise evaluation is explained by means of the execution model, in section 2.3.

Java has three categories of built-in data types: various *numeric* types, two *textual* types and a *logic* type. The essentials are explained below, complete descriptions of the types can be found in [1].

The numeric types can be grouped as dealing with integral numbers or with decimal decimal numbers, usually called *floating point numbers*. Of each of these groups the most relevant type, `int` `double`, respectively, is presented. They are used as the first choice, as so-called `default` type. The other numeric types are very similar to these and differ only in precision and range.

2.2.1 The integral type `int`

`int` is a so-called *integral* type, used to calculate with integers, i.e., whole numbers.

Syntax

1. literals: `..., -1, 0, 1, ...`
2. operators: + addition, – subtraction, $\star$ multiplication, / integer division, % modulo (remainder).

Semantics

The literals denote the corresponding values. For performance reasons, the value range of a type is fixed. The values in `int` range from -2^{31} to $2^{31} - 1$. Note that this exponent notation is not part of Java. It is important to be aware of these bounds, because if even during calculations values occur that are outside the range of the data type, the outcome may be quite unexpected.

The semantics of the operators is as in usual integer arithmetic, except for the last two operators. / denotes *integer division*, producing the integer part of a division. % denotes *modulo*, producing the remainder after integer division.

In the following example, we let the computer evaluate an expression that computes interest using integers. To show the result, we simply apply an output command to such an expression, i.e., we put the expression between the parentheses of an output command.

```
// handles account
public class InterestInt {


  // computes interest in integers
  void computeInterest() {

    System.out.print("For a balance of ");
    System.out.println(60);


    System.out.print("At rate 3% interest rounded down to a whole number is ");
    System.out.println((3 * 60)/100); // integer division
  }

// start-up code
  public static void main(String[] args) {
    new InterestInt().computeInterest();
  }
}
```

Output is:

```
For a balance of 60
At rate 3% interest rounded down to a whole number is 1
```

2.2.2 The floating point type `double`

`double` is a so-called *floating point* type, providing a notation for decimal numbers with a flexible positioning of the decimal point.

Syntax

1. literals: floating point notation, e.g., `123.45` or so-called "scientific" notation, with powers of 10, e.g., `1.2345E2`, which stands for $1.2345 \cdot 10^2$.
2. operators: `+` addition, `-` subtraction, `*` multiplication, `/` division.

The notation determines the type: `7.0` denotes a value of type `double`, whereas `7` denotes a value of type `int`.

Semantics

The literals denote the corresponding values. Note that `123.45` and `1.2345E` denote the same value. The values in `double` range from about $-1.79769313486231570 \cdot 10^{308}$ to $-1.79769313486231570 \cdot 10^{308}$ and smallest positive value about $4.9 \cdot 10^{-324}$.

The operations behave approximately as in decimal number arithmetic.

The idea of a "floating point" is that the round-off errors that are inevitable because of the finite representation of values in a computer can be minimized by placing the decimal point at the optimal place rather than at a fixed place. This is a complex issue; to provide some intuition about what precision can be expected in calculations, a sketch is given of how floating point calculations proceed. Roughly and intuitively, eighteen digits are available for the number part. Good precision is achieved by putting the decimal point immediately to the right of the first non-zero digit. For example, imagining instead of floating point numbers a 3-digit (three rather than eighteen for ease of writing the examples) fixed point number, i.e., with the point at a fixed place three digits from the right, 0.001234 can only be written as rounded to 0.001. In the 3-digit floating point case this can be written with greater precision as 1.23E-3.

An expression is evaluated by stepwise computing each (intermediate) value with the decimal point placed in the optimal position. For example, in a 3-digit fixed point computation, i.e., with the point at a fixed place three digits from the right, the computation 0.120 x 0.120 results in 0.014. Using 3-digit floating point notation, with the floating point placed immediately to the right of the first non-zero digit, the computation 1.20E-1 x 1.20E-1 results in 1.44E-2. As we see, the floating point representation achieves one digit more precision than the fixed point representation in this example.

Roughly, the precision of `double` calculations is determined by a representation using eighteen digits. Roughly, because the computer does not calculate with decimal fractions, it uses binary numbers. This difference affects the precision. E.g., the decimal number 0.5 is a recurring fraction when written as a binary number. Hence, it cannot be exactly represented in a Java calculation.

The division operator `/` applied to operands of the type `double` means ordinary division. This is different from its meaning when it is applied to two `int`s. Using the same symbol to denote different operations is called *overloading* of that symbol. In an expression with an overloaded symbol, the type of the operands determines the operation that will be used.

We change the previous example to a program that computes and prints interest with `double` precision.

```
// handles account
public class InterestDouble {
```

```
  // computes interest precise and rounded
  void computeInterest() {

    System.out.print("For a balance of ");
    System.out.println(60.00);


    System.out.print("At rate 3% interest is exactly ");
    System.out.println((3 * 60.00)/100); // double division
  }

// start-up code
  public static void main(String[] args) {
    new InterestDouble().computeInterest();
  }
}
```

```
For a balance of 60.0
At rate 3% interest is exactly 1.8
```

Java has several more numerical types, providing for different requirements as regards range of values, precision and amount of storage space used. The numerical types are, ordered according to increasing range, `byte, short, int, long, float, double`. `byte, short, int, long` are *integral* types, `float` and `double` are *floating point* types.

The type of a value is determined by the notation, as mentioned before, `7` is an `int` and `7.0` a `double`. The other types are distinguished by a letter after the digits. For example, to distinguish literals of type `long` from those of type `int`, the first have an `L` after the digits, like `7L`. This can be used to ensure that the storage space is large enough for a result or intermediate value occurring during the computation: `(1000000L * 1000000L)/1000000L` remains within the range of a `long` and correctly evaluates to `1000000`. On the other hand, `(1000000 * 1000000)/10000000)` exceeds the size of an `int` during computation and wrongly evaluates to `-727`.

	description	example expression	result
*	multiplication	`5 * 2`	10
/	integer division ordinary division	`5 / 2` `5.0 / 2`	2 2.5
+	arithmetic addition string concatenation	`1+1` `"1"+"1"`	2 "11"
–	subtraction minus	`5-2` `-(5-2)`	3 -3
%	remainder after division	`10 % 5` `7 % 5`	0 2

Figure 2.1: Arithmetic operators and String concatenation

2.2.3 The textual type `char`

`char` is the type used to represent single characters.

Syntax

1. Literals: characters in single quotes, such as `'c'`, `'7'` and `'&'`.
2. Operators: although applicable, the numerical operations are not very useful. Sometimes the comparison operators (==, etc., see below) can be used.

Semantics

A character in single quotes denotes the corresponding character value.

2.2.4 The textual type `String`

`String` is a textual type, used to represent sequences of characters.

Syntax

1. Literals: sequences of characters in double quotes, such as `"Hello world!"` , `"7"` and `"17"`.
2. Operators: + concatenation.

Semantics

A string in double quotes denotes the corresponding string value. + denotes concatenation of strings, e.g., `"a"+"b"` has the value `"ab"`.

So the symbol + is overloaded: for `int` and `double` arguments addition is performed, for `String` arguments concatenation is performed.

In the following example, we let two strings be concatenated and then be output.

```
// String concatenation
class HelloString {

    // concatenates and outputs two Strings
    void show() {
        System.out.println("Hello, World, " + "this is a Java program."); //concatenate Strings
        System.out.println("Good luck, programmer!");
    }

    // start−up code
    public static void main(String[] args) {
        new HelloString().show();
    }
}
```

Output is

```
Hello World, this is a Java program.
Good luck, programmer!
```

2.2.5 The logic type `boolean`

`boolean` is the logic type, used to handle the logical values `true` and `false`. The role of `boolean` in programming is somewhat different from that of the other types. As will be shown in chapter 5, logic values play a supporting role in programming.

Syntax

operator	description	example	result
`&&`	conjunction, logical *and*	`0 < 2 && 2 < 3`	true
		`0 < 2 && 3 < 2`	false
`||`	disjunction, logical *or*	`0 < 2 || 2 < 3`	true
		`2 < 0 || 3 < 2`	false
`!`	negation, logical *not*	`!(2<0)`	true
		`!true`	false
`^`	exclusive or, either or	`2 < 0 ^ 0 < 2`	true
		`0 < 1 ^ 0 < 2`	false

Figure 2.2: Boolean operators

operator	description	example	result
`<`	less than	`2 < 3`	true
`>`	greater than	`2 > 2`	false
`<=`	less than or equal	`2 <= 2`	true
`>=`	greater than or equal	`3 >= 5`	false
`==`	equal	`1 == 1`	true
`!=`	unequal	`1 != 1`	false

Figure 2.3: Comparison operators (Booelan result)

1. Literals (just two): `true`, `false`.
2. Operators: `||`, `&&`, `!`, and `^` .

Note that there are also operators `&` and `|`. These behave a little different from `&&` and `||` and are of more specialised use. They are no treated here.

Semantics

The two literals denote the corresponding truth values. `||` denotes logic "or" (disjunction), `&&` denotes logic "and" (conjunction) `!` denotes logic "not" (negation), `^` denotes "exclusive or".

The following example shows the values of some logic expressions. Figure 2.2.4 gives an overview with examples.

```
// logic expressions
class LogicExpressions {

   // evaluates logic expressions and outputs results
   void show() {
      System.out.print("true || false evaluates to ");
      System.out.println(true || false);

      System.out.print("true && false evaluates to ");
      System.out.println(true && false);

      System.out.print("! true evaluates to ");
      System.out.println(! true);

      System.out.print("true ^ true evaluates to ");
      System.out.println(true ^ true);

   }

   // start-up code
```

```
    public static void main(String[] args) {
        new LogicExpressions().show();
    }
}
```

Output is:

```
true || false evaluates to true
true && false evaluates to false
! true evaluates to false
true ^ true evaluates to false
```

2.2.6 Comparison operators

There is something more to operations on data types: there are also operations that take values from one data type but produce a value in another data type. An example are the comparison operations, where comparing two values of some type yields a value `true` or `false` of type `boolean`. Because the result is of a boolean type, these are called boolean expressions.

Comparison operators, applicable to all numeric types are == (equals), != (does-not-equal), > (greater than), >= (greater than or equal), < (smaller than), and <= (smaller than or equal). Figure **??** gives an overview. Note that they work for all numeric types, not for Strings. For alphabetic ordering and (in)equality of Strings, see section 2.2.6.

Java uses so-called *Unicode*, a numerical encoding, to represent characters as from `char`. A character in single quotes, like `'c'`, has as numerical value the corresponding Unicode. The equality operator (==) will later be shown to be of use in programming; as it compares the unicode values, the comparison is case-sensitive. The < operator then means "less than" according to the ordering of the Unicodes. Because of the Unicode values chosen, this is like an alphabetic ordering, although it is not always useful, since it, e.g., orders all lower case characters before (less than) all uppercase characters.

Comparison of Strings

The ordinary comparison operators can not be used for comparing Strings. Instead, use *equals* for (in)equality and *compareTo* for alphabetic (lexicographic) ordering. Examples:

`"abc".equals("def")` evalates to false
`"abc".equals("abc")` evalates to true
`"abc".compareTo("def")` evalates to -1
`"def".compareTo("abc")` evalates to 1
`"abc".compareTo("abc")` evalates to 0

The unusual syntax comes from the fact that `equals` and `compareTo` are not real operators, but methods. This will be explained later.

Remarks Note that == (and `!=`) can be applied to Strings, but gives different results from `equals`. This will also be explained later.

The following example shows some comparisons.

```
  // comparisons
class Compare {

  // evaluates comparisons and outputs results
  void compare() {
```

```
    System.out.print("boolean expression 7 + 17 > 25 evaluates to ");
    System.out.println(7 + 17 > 25);

    System.out.print("boolean expression (true || false) == true evaluates to ");
    System.out.println((true || false) == true);

    System.out.print("boolean expression 'c' == 'C' evaluates to ");
    System.out.println('c' == 'C'); // these are chars, don't do this with Strings!

    System.out.print("boolean expression \"Hello world\".equals(\"Hello \" + \"world\") evaluates to ");
    System.out.println("Hello, World".equals("Hello, " + "World"));
  }

  // start-up code
  public static void main(String[] args) {
    new Compare().compare();
  }
}
```

Output is:

```
boolean expression 7 + 17 > 25 evaluates to false
boolean expression (true || false) == true evaluates to true
boolean expression ('c' == 'C') evaluates to false
boolean expression "Hello world".equals("Hello " + "world") evaluates to true
```

Comparing expressions such as these and printing "true" or "false" does not seem very useful. In the next chapters, when more programming constructs are available, more useful applications of comparisons will be shown.

2.2.7 Mixing types in an expression

The idea of a type as concerning just one closed set of values is somewhat too restrictive for many practical situations. Sometimes it is practical to mix similar types in expressions, say adding an integer to a double. Therefore there is some flexibility in writing typed expressions.

In expressions it is allowed to mix different numeric types. The values are then automatically converted to the type used in the expression that has the greatest range or precision. This is called *widening*. For example, `7 + 17.0` has `double` value `24.0`.

For a complete description of how these matters are resolved in Java, see the [1], Chapter 5 *Conversions and Promotions*.

Warning: be careful with the overloaded operator `+`. Do not apply it to Strings and numbers in the same expression, or at least separate the numeric parts from the String-parts.

2.2.8 Type correctness

Because literals are typed, we know which operators can be applied to them and also, in case of overloading, which operation an operator denotes. Furthermore, the expression itself has a type, following from the type(s) of its constituents. So we know which evaluable expressions can be formed, namely those consistent with the type(s) used in the expression. Such an expression is called *type correct*. It is not necessary to evaluate the subexpressions to determine the type of an expression: it suffices to know the types of the subexpressions. Type correctness thus can be, and is, checked by the compiler before execution.

The notion of type correctness and the compiler match the flexibility that allows mixing types to some extend.

2.3 Execution model

Evaluation of expressions in Java follows the rules of arithmetic and in general the results should not be surprising. When expressions get more complicated, it is sometimes helpful to know what exactly is going on. Therefore, the evaluation process is explained in some detail here.

Evaluation of expressions involves two activities: determining the operands for each operator, *parsing*, and applying the operators in some order.

2.3.1 Parsing

The subexpressions that are the operands for an operator follow from the binding rules of the operators. These rules are are as in arithmetic (multiplication before addition, etc.), with the additional clause that if two operators are equal in binding strength, they are applied from left to right (so-called *left-association*).

The following example shows the parsing process in steps. The binding of the operators is shown by means of extra parentheses.

`7 - 5.0 + 2 * 3 + 17` gets parsed into

`7 - 5.0 + (2 * 3) + 17` because of stronger binding of `*`; this gets parsed into

`(7 - 5.0) + (2 * 3) + 17` because of left-association; this gets parsed into

`((7 - 5.0) + (2 * 3)) + 17` because of left-association.

Parsing is part of the compilation process, expressions are analyzed syntactically and the operators are associated to operands.

When during the execution of the program an expression is evaluated, the process is as follows.

2.3.2 Evaluation step by step

At each step in the evaluation, a so-called *evaluable* subexpression is evaluated and is replaced by the resulting value. A subexpression is evaluable if its operands are values, i.e., not composite subexpressions. When there is more than one evaluable subexpression, the left-most one is chosen.

The following example explains this. Before each step the subexpression that is going to be evaluated is underlined.

$\underline{7 - 5.0} + 2 * 3 + 17 \longrightarrow$
$2.0 + \underline{2 * 3} + 17 \longrightarrow$
$\underline{2.0 + 6} + 17 \longrightarrow$
$\underline{8.0 + 17} \longrightarrow$
25.0

Presented here is the most detailed level of modeling: the evaluation of each subexpression is pictured as a step. Depending on the level of detail desired, steps can be omitted. For example, all left-to-right evaluations could be pictured as one step; this would remove the second state in the example. Also, the whole evaluation could be pictured as one step; this would remove all but the first and last state in the example.

2.4 Remarks

2.4.1 Three kinds of errors

The first kind of error is, as we have seen already, that a program is not syntactically correct. The compiler checks for this, without running the program and produces error messages when the program is not correct.

Example: a `;` is omitted. The program is not executed and the compiler issues a message such as

```
Compare.java:8: ';' expected
    System.out.println(7 + 17 > 25)
```

Syntactic correctness however, does not guarantee the absence of a second kind of errors, *run-time* errors, that occur during execution and disrupt the execution, causing the program to *abort*. For example, when in a computation-step division by `0` occurs, the compiler will not report an error, because the problem only shows up when the expression is evaluated. During execution, a message like the following will occur:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at InterestError.computeInterest(InterestError.java:13)
      at InterestError.main(InterestError.java:18)
```

Even when a program does not abort, there can still occur a third type of errors, so-called *logical* errors, namely that the program does not produce the intended result. For example, when the wrong formula for computing the interest on a balance is used the program may well execute without aborting, but the outcome will be wrong. More subtly, if in a computation a data type is used that does not have the appropriate precision or range, the outcome may be wrong too.

For example, the Java statement `System.out.println(1000000000+2000000000);` will give as output:

-1294967296

because the correct answer, 3000000000, can not be represented in an int.

Chapter 3

Data storage – Typed variables

In the previous chapter it was shown how to calculate with data values. Often it is necessary to also store values. For example when the same initial value is to be used several times or when intermediate values need to remain available for later use in a computation. Additional to providing built-in data types for values and operations on them, Java enables storage of values.

3.1 Aim

We want to store and retrieve data values of built-in data types.

3.2 Means

We use *typed variables* as storage space for values of a built-in type in the computer. We use the *assignment operator*, = to store a value in a variable of that type.

As an example, we use the types `double` and `int` to store the contents and the interest rate of a bank account. We use the variables in computing the interest and new balance and outputting data. We write the code stepwise.

To obtain a space for values, we *declare* variables at the top of the class, before the method. A declaration of a variable consists of its type followed by the name of the variable, which we choose. At creation of the object, the necessary storage space for values of that type is reserved.

```
// handles account
public class AccountInterest {

   double balance; // variable decalaration
   int rate; // variable declaration
}
```

To store a value in a variable, we use the assignment operator, =. On the left of = we put the name of a variable. On the right of = we put an expression, e.g., a simple value, of the corresponding type. Together this forms an *assignment statement*. When the assignment statement is executed, the expression on the right hand side of the = is evaluated and the resulting value is stored in the variable on the left hand side.

```
// handles account
public class AccountInterest {
```

```
  double balance;
  int rate;

  // updates balance
  void update() {

    balance = 60.00; // storing the balance value
    rate = 3; // storing the rate value
  }
}
```

To retrieve a value, the notions of expression and evaluation are extended. The name of a variable is also an expression. Note that an expression and thus also a variable as well, can be used to form larger expressions. When at execution an expression is evaluated that contains a variable, the value stored in the variable is retrieved and used. E.g., the `System.out.print();` and `System.out.println();` commands can also be used for variables: in that case the value of the variable is output. The precise evaluation is explained by means of the execution model, in section 3.3.

```
// handles account
public class AccountInterest {

  double balance;
  int rate;

  // updates balance
  void update() {

    balance = 60.00;
    rate = 3;

    System.out.print("Current balance is ");
    System.out.println(balance); // retrieving the balance
    System.out.print("Interest rate is ");
    System.out.println(rate); // retrieving the rate

    balance = balance + (rate * balance)/100; // retrieving, computing and storing

    System.out.print("New balance is ");
    System.out.println(balance); //retrieving the new balance
  }

// start-up code
  public static void main(String[] args) {
    new AccountInterest().update();
  }
}
```

Output is

```
Current balance is 60.0
Interest rate is 3
New balance is 61.8
```

Variables `balance` and `rate` are initialized at the start of the program. If the same computation is desired for different values, only the initialization has to be changed. E.g., the code for the computation of the new balance `balance = balance + (rate * balance)/100` remains unchanged, because it uses values stored in variables rather than fixed values.

Note that the assignment statement is asymmetric: when it is executed, first the right-hand side is evaluated and then the result is stored in the variable on the left-hand side. In the example, in the update of `balance`, the old value stored in `balance` is retrieved and used in the evaluation of the expression on the right hand side of the =, and only then the occurrence of `balance` on the left hand side of = causes the new value to be stored in`balance`.

Be aware of the difference between = and ==. The first is the assignment operator and the second is the comparison operator that tests for equality. This different from mathematics, where = means equality (assignment is not used much in mathematics in this form). Some programming languages use := for assignment and = for equality.

3.2.1 Mixing types in assignment

As for values, types also allow some flexibility for variables: if a value is stored in a wider type than its own type, it is converted to that type. Note that the value is converted, i.e., if a value is stored in a variable of wider type, the value retrieved will be of the wider rather than the original type.

Conversely, the type of a value can be narrowed using the cast operator. For example, if one knows that at some point in the computation a variable `g` of type `long` in fact holds a value that is only of int-size, one can convert this to a value of type `int` and store it in a variable `i` of type `int` through casting: `i = (int)g`. If the actual value of `g` is greater than an `int` can hold, the operation will nevertheless be executed and the result is probably not what was intended. It is wise to avoid the cast operator whenever possible.

3.2.2 Type correctness

Type correctness as introduced for expressions with just values in chapter **??** extends to variables and assignment.

Because variables are typed, we have the information which type of values can be stored in a variable, and thus also which type of values will be retrieved. So we know which evaluable expressions can be formed with those variables, namely those for which the type of the variables are consistent with the types that we want in the expression. We also know which values can be stored in a variable, namely those of corresponding type. For example, if `x` is an `int` variable and `s` a `String` variable, the expression `x * s` is illegal and the assignment `x = s` is illegal as well.

Even though the values are stored in the variables at run-time, the type of the variables is fixed, so type correctness can be checked by the compiler using the type of the variables (which restricts the type of values that can be stored), i.e., before execution. It is therefore called *static type checking*. *Strong typing* means that the static type check guarantees that no run time errors will occur because of wrong typing. Java supports strong typing to a great extent.

3.3 Execution model

The execution model is extended for variables.

The object state is extended with variables that are put above the dotted line in the object. These are pictured as little boxes containing the value of the variable and labeled with name and type of the variable. When the object is created the boxes contain the standard initial value of the type, which is 0 for all numeric types and `false` for the boolean type. Strings are initialized with the special value `null`, which will be explained later. The execution model provides the dynamic, changing, information about an object, which includes, obviously, the value of variables.

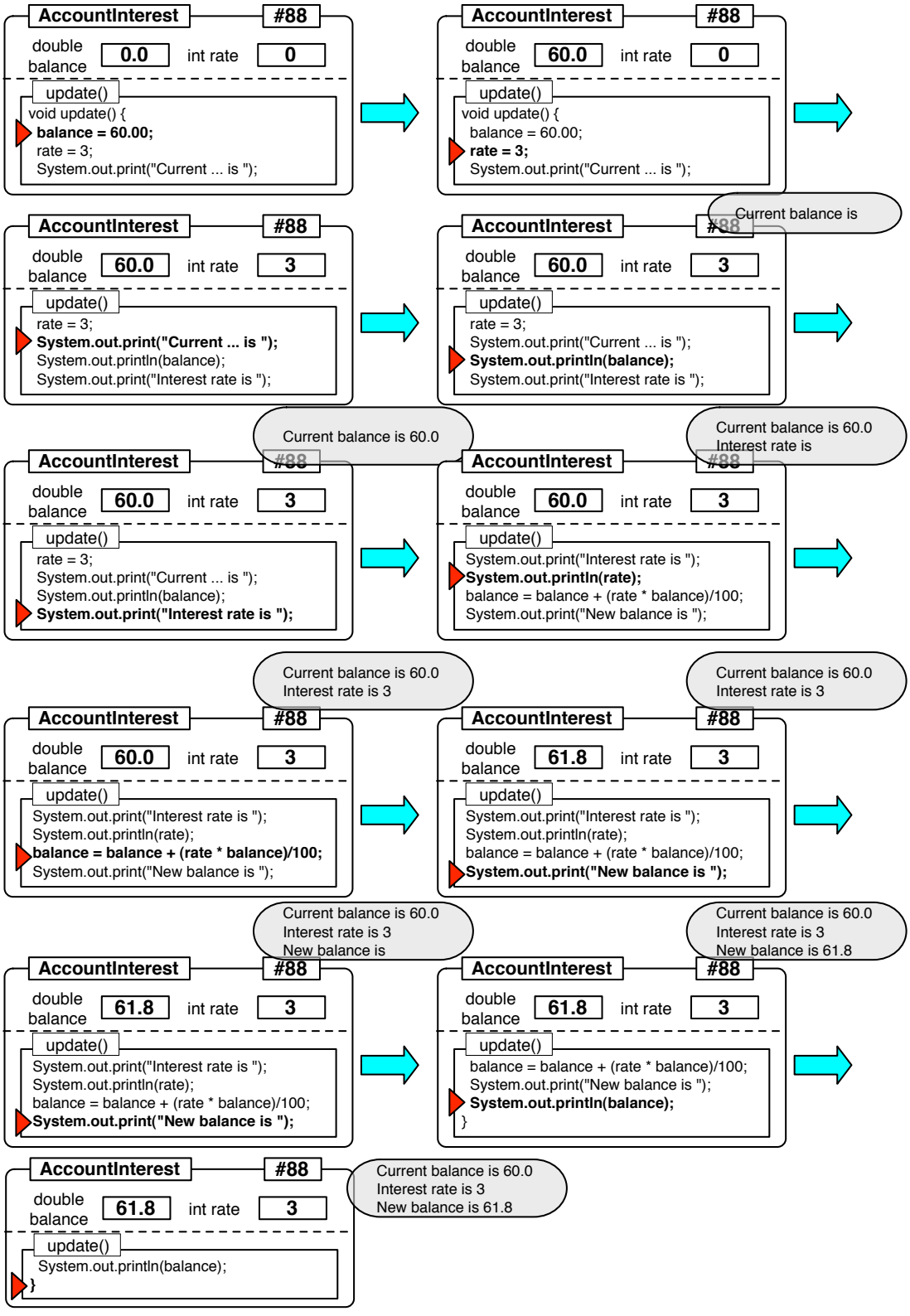
When an assignment statement is executed the value of the expression on the right hand side of the =-sign is written in the box of the variable which name appears on the left hand side of the =-sign.

In addition to this, variable names can also occur in expressions. Therefore, evaluation of expressions is extended with retrieving the values stored in variables. When we want to show the execution of an expression in detail we include a step for each variable evaluation in the expression. The order in which the values of these variables are retrieved is from left to right.

The execution of `AccountInterest` is depicted in figure 3.1.

In the execution of the assignment statement `balance = balance + (rate * balance)/100;` the two occurrences of the variable `balance` on the right hand side of the assignment evaluate to `60.0`, as this is stored in the variable `balance` in the corresponding states. The resulting value `61.8` is put into the variable `balance` in the state where the assignment is completed.

Figure 3.1: Execution of AccountInterest



Chapter 4

Console input – Scanner and System.in

In the previous chapter variables were initialized through assignment of values that were written in the method code. That meant that each time you want the program run on different values, you have to change the code, compile again, and run the program. We want more flexibility, where the user can enter values when the program runs. Java provides several ways to provide data to a running program, such as: from the network, from a file on disk, from the user operating the mouse or the keyboard. In this chapter, we focus on keyboard input, typed into the *console*, the window (or window pane) where output from `System.out.println` is presented.

The result of data processing by a program was shown to the human by outputting these on the console. It is often convenient to also enable the user to input values to a program via the console. Java provides predefined code that enables to do this.

4.1 Aim

We want to input values from the console.

4.2 Means

We use *console input* statements that are provided by a predefined *utility class*, `Scanner`.

For using input in Java we have to do a little more than we have to do for using output. We have to add some standard code to obtain this utility – full explanation of this code is deferred.

1. Before the code of the class we add: `import java.util.*;`
2. Before the variable declarations in the class we add: `Scanner sc = new Scanner(System.in);` (instead of `sc` any name will do, as long as you use it consistently throughout your program.

The effect of these statements is that statements are available in your program to get input from the console, e.g., `sc.next()`, that is explained below.

When `sc.next()` is up for execution, the user can type a word, i.e., a sequence of characters without spaces, on the console window. When subsequently the user types a return, the statement takes the input from the console to the program (one says, the statement *reads* the word. When an input statement is up for execution and there is no input available, the program waits until the user types something and presses return. To prompt the user that an input is expected, we often put a request for input on the console, using a `System.out.println...` statement preceding the input statement.

After input, `sc.next()` acts as an expression that has as a value the text that has been typed in. Since every value has a type, also `sc.next()` has a type: `String`. To use this value in the program, it can be assigned to a variable of type String using an assignment statement. For example, the statement `str = sc.next();` will input a word from the console (typed in by the user) and store this as a String in `str`, assuming this is a variable of type String.

If a value of another type than `String`, e.g., of type `int` an input statement has to be used that returns that type. Java provides input statements for the built-in types, except `char` and a few others, in particular the following.

- `sc.nextInt()` inputs an integer number and has a value of type `int`
- `sc.nextDouble()` inputs a floating point number and has a value of type `double`
- `sc.nextBoolean()` inputs a boolean (the word "true" or "false") and has a value of type `boolean`
- `sc.next()` inputs a word (a sequence of characters not containing space, tab, or return) and has a value of type `String`

Their semantics is a little subtle, to enable practical use of console input. On the console, we can put not only one word , but several, separated by by *whitespace*, i.e., one or more spaces, tabs, or newline characters. When followed by a return, these words are taken from the console and put in a queue inside the computer and then processed one by one by the input statements that, consecutively, are up for execution. Processing means that the word is interpreted as a value of the right type (depending on the input statement, `nextInt`, `nextDouble`, etc.) and the word is removed from the queue. When the queue is empty, the execution of the program is held up until the user provides new words on input (and typing return).

Additionally, there are two input statements that are somewhat different.

- `sc.nextLine()` inputs a whole line and has a value of type `String`

Different from the other statements, whitespace is not used as marking the end of the input, but only the end of the line. The resulting value is a String with all text from the current position to the end of the line.

- `sc.next().charAt(0)` has a value of type `char`

Different from the other statements, this statement takes as input single character values. It still inputs the complete word. The rest of the word is discarded.

Note that all these commands only work well when a value of the right type is typed. E.g., when `sc.nextInt();` is executed and the user types `abc`, a run-time error will occur.

In the example below we use the additional code and the direct input of `double` and `int` values.

```
import java.util.*; // provides Scanner description
// handles account
public class AccountInput {

  Scanner sc = new Scanner(System.in); // enables console input via Scanner

  double balance;
  int rate;

  // inputs account info and outputs it using console
  void infoAccount() {

    System.out.println("Type amount for balance and for interest rate");
    balance = sc.nextDouble(); // inputs balance amount from console
    rate = sc.nextInt(); // inputs rate from console
```

```
    System.out.println("Current balance is " + balance + ".");
    System.out.println("Interest rate is " + rate + ".");

    balance = balance + (rate * balance)/100;

    System.out.println("One year interest has been added.");
    System.out.println("Current balance is " + balance + ".");
  }

  // start-up code
  public static void main(String[] args) {
    new AccountInput().infoAccount();
  }
}
```

Output and input (indicated by underlining) are:

```
Type amount for balance and interest rate.
60.0
3
Current balance is 60.0.
Interest rate is 3.
One year interest has been added.
Current balance is 61.8.
```

Because words are separated by any amount of white space input could also have been given as follows.

```
60.0 3
```

or

```
          60.0

  3
```

NB The command `nextDouble()` looks for a decimal point or a decimal comma, depending on the settings of your computer. E.g., if your computer is configured for the Dutch settings (so-called *locale*), it will only accept a comma in double numbers. In this book, we will use a decimal point.

4.3 Execution model

The input is shown in the console window. Text that hasn't been read is presented in bold face.

Example to be supplied.

4.4 Remarks

4.4.1 Separating input and conversion

Another approach to inputting values is to use only the `sc.next()` statement for inputting values. `sc.next()` can input any word, i.e., sequence of non-whitespace characters, that is put on the console and delivers it as a value of type `String`.

If the value is intended to be of another type than `String`, e.g., of type `int`, it is separately converted from the type `String` to that type. For each built-in type there is a conversion operator. For the type `int` this is `Integer.parseInt()` which converts a `String` value to an`int` value. Inputting a value of intended type `int` and assigning it to a variable `i` of type `int` therefore is done by

```
i = Integer.parseInt(sc.next()).
```

When the input word can not be converted to the requested type (e.g., some letters are read and `parseInt` is applied, an error message is issued and execution is stopped.

Chapter 5

Data dependent execution order – data dependent control flow

The execution order of statements in a method is determined by the order in which control passes to statements, the control flow. Up till now, statements were executed in the order as written; control flow was sequential. Furthermore, this order was independent of the data values being processed; control flow was data independent. In many cases this is too restrictive, as at certain points in the execution it may depend on the current data values which of several possible actions should be taken, i.e., which of several statements should be executed. Java provides data dependent control statements that make this possible.

5.1 Choice – if-else statement

Depending on data values one of two pieces of code is chosen for execution.

5.1.1 Aim

We want to describe that one of two statements that are both written in the program are executed, where which one is executed depends on data values that may differ for different executions of the program.

5.1.2 Means

We use the *if-else* statement, which selects one of two statements for execution, depending on the value of a *selection guard*. The flow of control is data dependent: which statement is chosen for execution depends on data values and may, e.g., differ for executions with different input values.

In the example below, different updates are performed on an account depending to whether a balance is solvent or overdrawn.

```
import java.util.*;

// handles account
public class AccountUpdate {

    Scanner sc = new Scanner(System.in);
    double balance; // balance of account in some currency
    int rate; // interest rate in percent
    int debitRate; // debit rate in percent
```

```
    // set and update balance
    void handleBalance() {
        System.out.println("Type interest rate.");
        rate = sc.nextInt();

        System.out.println("Type debit rate.");
        debitRate = sc.nextInt();

        System.out.println("Type amount for balance.");
        balance = sc.nextDouble();

        if (balance>=0) { // guard
            balance = balance + (rate * balance)/100;
            System.out.println(rate + "% interest has been added.");
        } else {
            balance = balance + (debitRate * balance)/100;
            System.out.println(debitRate + "% debit interest has been subtracted.");
        }

        System.out.println("New balance is " + balance + ".");
    }

    // start-up code
    public static void main(String[] args) {
        new AccountUpdate().handleBalance();
    }
}
```

Resulting output is, in case of inputting a non-negative amount of, e.g., 60:

```
Type interest rate.
3
Type debit rate.
5
Type amount for balance.
60
3% interest has been added.
New balance is 61.8.
```

Resulting output is, in case of inputting a negative amount of, e.g., -30:

```
Type interest rate
3
Type debit rate.
5
Type amount for balance.
-30
5% debit interest has been subtracted.
New balance is -31.5.
```

When the `if-else` statement is executed, first the guard, `(balance >= 0)`, is evaluated. In case the guard is true, i.e., the account is solvent, the statements between the braces following the word `if` are executed, adding interest. In case the guard is false, i.e., the account is overdrawn, the statements between the braces following the word `else` are executed, imposing the penalty. After execution of the chosen statement, execution of the `if-else` statement is completed and control goes, in both cases, to the next statement, showing the new balance.

Syntax Templates

The if-else statement comes in three versions.

Choice between two alternatives:

```
if ( boolean expression ) {
      statements1
} else {
      statements2
}
```

Choice between executing or not executing a statement:

```
if ( boolean expression ) {
      statements1
}
```

Choice between more than two alternatives:

```
if ( boolean expression ) {
      statements1
} else if ( boolean expression ) {
      statements2
...
} else {
      statements
}
```

A group of statements surrounded by braces, a so-called *code block* is again one statement. By convention, we always write the statements in the two clauses of the `if-else` statement between braces, also if a clause consists of only one statement. The complete `if-else` statement together with its sub-statements is again one statement. If the program only needs to perform a statement when the guard is true, and does not need to do anything in case the guard is false, the `else` clause may be omitted.

Semantics

For the standard if-construction with two alternatives, the guard is evaluated, if `boolean_expression` is true, `statements1` is executed, else `statements2` is executed. Then the execution of the `if-else` statement is completed and control goes to the next statement in the program.

5.1.3 Execution model

Example to be added

5.2 Repetition – while statement

Depending on, changing, data values a statement is executed repeatedly.

5.2.1 Aim

We want to describe that a statement (or group of statements) is executed repeatedly, where how often the statement is executed depends on variables that change during the execution of the statement.

5.2.2 Means

We use the *while statement* to repeatedly execute a statement, the *while body*, depending on the value of a *repetition guard*. The flow of control is data dependent: how many times the statement is executed depends on data values that change during the execution of the program. The data values that govern the control flow should be updated in the statement that is repeatedly executed: data change is used to stop the repetition.

In the example below, the build-up of compound interest of 5% from an inputted balance is shown, until the owner is a millionaire.

```
import java.util.*;

// calcualtes account build−up
public class AccountCompound {

  Scanner sc = new Scanner(System.in);

  double balance;

  int rate;

  // shows compound interest build−up
  void handleBalance() {

    rate = 5;

    System.out.println("Type amount for balance.");
    balance = sc.nextDouble();

    while (balance < 1000000) {
      System.out.println("Balance is " + Math.round(balance) + ".");
      balance = balance + (rate * balance)/100;
    }
    System.out.println("Balance is " + Math.round(balance) + ".");
    System.out.println("Hello, millionaire.");
  }

// main instantiates and uses the object
  public static void main(String[] args) {
    new AccountCompound().handleBalance();
  }
}
```

Console shows (input is underlined):

```
Type amount for balance.
800000
Balance is 800000.
Balance is 840000.
Balance is 882000.
Balance is 926100.
Balance is 972405.
```

```
Balance is 1021025.
Hello, millionaire.
```

When control arrives at the *while statement*, the guard, `balance<1000000` is evaluated.

If it is *true*, the body

```
balance = balance + (rate * balance)/100;
System.out.println("Balance is " + Math.round(balance) + ".");
```

is executed and after that control goes back to the guard and the execution repeats.

If it is *false*, the *while* statement is completed and control goes to the next statement.

The general shape and meaning of the while statement are as follows.

Syntax Template

`while` (*boolean expression*) {
    statements
}

Semantics

When control arrives at the while-construct, the *guard* is evaluated. When it is *false*, the while-construct terminates immediately, otherwise the *body* is repeatedly executed. After each execution of the body, the guard is evaluated and the execution of the while-construct is terminated if it is false.

5.2.3 Execution model

In figure 5.1 the first states of the execution of AccountCompound are shown. The value of the variable `scanner` is presented as a number similar to the number that labels the AccountCompound object. We will explain this later.

5.3 More repetition constructs

There are two more repetition constructs that are similar to the `while` construct. There is no essential difference, but in some case one construct is intuitively easier to use than the other.

5.3.1 The do-while-loop

The do-while is similar to the while construct, with the difference that the body is executed at least once, also when the guard would be initially false.

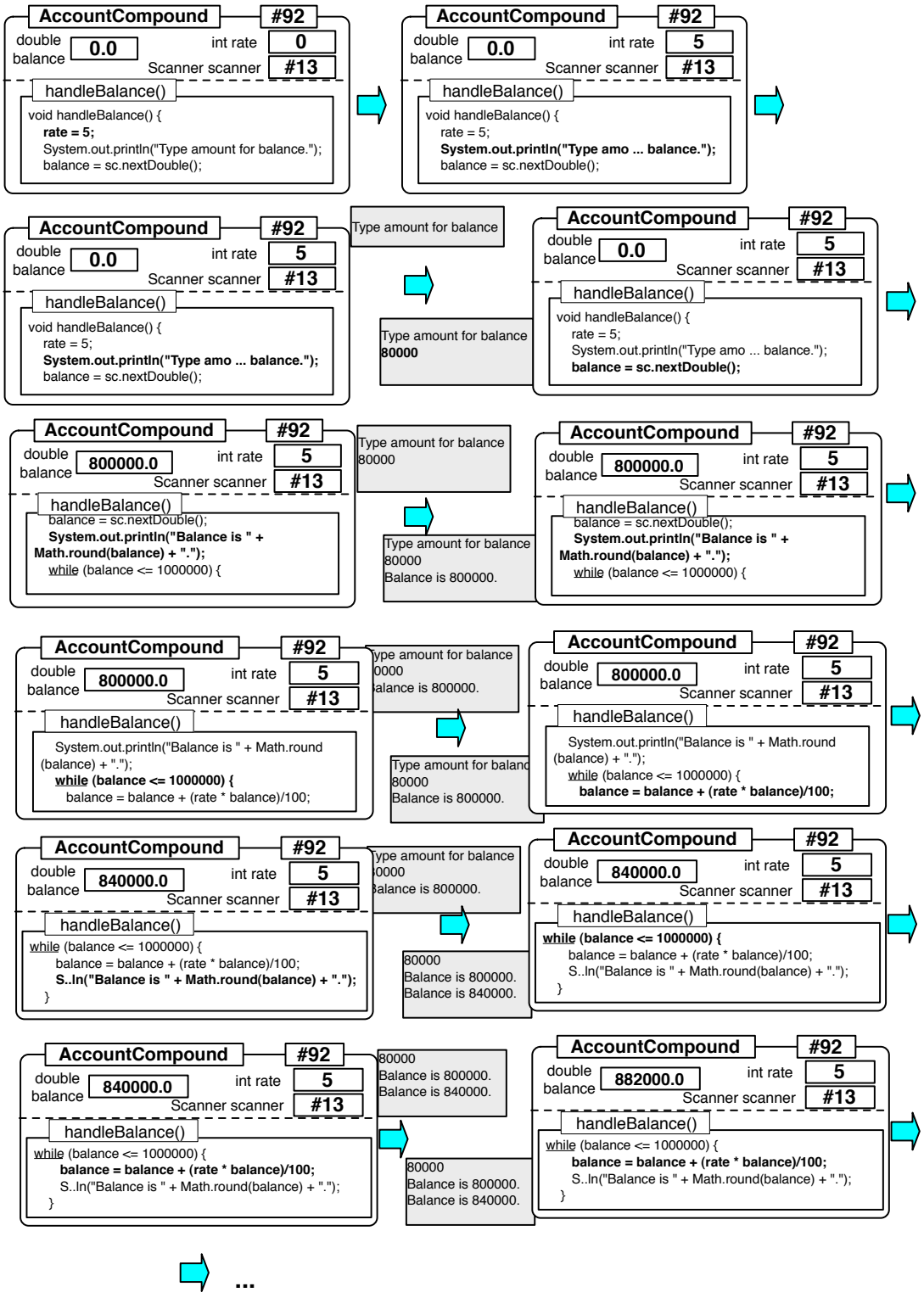
```
import java.util.*;

// calcualates balance build-up, one or more deposits
public class AccountDoWhile {

  Scanner sc = new Scanner(System.in);

  double balance;
  double deposit;
```

Figure 5.1: Initial part of the execution of AccountCompound



```
  char choice;

  // shows build−up
  void add() {
    balance = 0;

   do {
      System.out.println("Type amount for deposit.");
      deposit = sc.nextDouble();
      balance = balance + deposit;
      System.out.println("Balance is " + Math.round(balance) + ".");

      System.out.println("Do you want to make a deposit? Type y/n");
      choice = sc.next().charAt(0);
    } while (choice == 'y');
  }

// main instantiates and uses the object
  public static void main(String[] args) {
    new AccountDoWhile().add();
  }
}
```

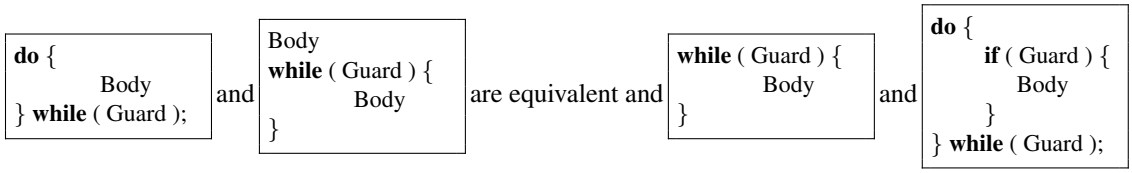
Syntax Template

`do` { *statements* } `while` (*boolean expression*) **;**

Semantics

When control arrives at de do-while-construct, the body (*statements*) is executed. After this, and every following execution of the body, the guard (*boolean expression*) is evaluated. If it is true, the body is executed again, if it is false, the loop is terminated, i.e., execution continues after the construct. Note that after one execution of the body, this constracut behaves the same as the while construct.

The while and do-while construct are equivalent in expressive power: one can easily convert a while-loop into a do-while and vv. The figure below explains how.



5.3.2 The for-loop

In the for-loop, the initialization, the evaluation of the guard and the update of the guard are grouped together and the body is separated from these.

This is the preferred construct when it is known on beforehand how many repetitions are required. For example, if it is required to update a known balance 10 times, the code for the repetition is:

```
for (int i = 0; i < 10; i = i+1) {
    balance = balance + (5 * balance)/100)
    System.out.println("balance is " + balance);
}
```

Syntax Template

`for` (*initialization* ; *guard* ; *update*) {
    statements
}

Semantics

The *initialization* gives the variable in the guard its starting value (if more than one initialization statement is needed, these have to be separated by commas). Then control arrives at the guard. If the guard is true, control arrives at the *body*, else control goes to the end of the for-statement. Then the *update* is executed and control returns to the guard.

Note that the update is executed after each execution of the body.

All three repetition constructs of java have the same expressive power.

5.3.3 Which control statements should there be – goto?

Intuitively, this dependency of execution order on data could be achieved more flexibly by a control statement like:
if control is at a certain point and some variable has certain value, then control goes to some other point, for example indicated by a label. This idea was adopted in early programming languages. This so-called `goto` control statement fully achieved data dependent control flow, intuitively to encode all control flow orderings.

However, the resulting programs became very difficult to understand, because the labels could be placed anywhere: from the static coded of program it was very hard to see where during the dynamic excution control would go, resulting in error-prone programs. Therefore modern programming languages do not have a `goto` statement anymore, but the more well-structured data dependent control statements given above: the `if-else` construct and the `while` (or one or more variants).

This does not limit the programming tasks that can be solved: the two statements can be shown to enable to write programs with the same computation power as the `goto`, i.e., provide the same *expressive power* but in a more structured, less error-prone manner.

5.3.4 Computability

Not only provide the above statements the same computation power as the goto, but a much stronger and more general result holds.

There is an intuitive, not formally defined, notion of *effectively computable*: "anything that can be computed following a recipe of simple instructions". There are various quite different formalizations of this intuitive notion, notably Turing machines, an operational definition of an idealized computer, and recursive functions, a non-operational, mathematical notion. These and more formal notions have been proposed and all of them are proven to be equivalent in computational power.

This led to *Church's Thesis*: The effectively computable functions are the ones that are computable by a Turing machine (or a recursive function, or any of the other provably equivalent formal notions).

The result now is that, given some suitable coding agreements, any effectively computable function is computable by a computer that has the data type integer and the control flow constructs sequential composition, choice and while, provided there is unlimited storage space.

This means that at this point we can in principle program everything! However, the main problem in programming is not what can be programmed in principle, but how to obtain the programs. For this, *structuring*, more specifically *decomposition* into manageable parts and *abstraction* to a manageable level is the main option. This is what the rest of the book is about.