Object Oriented Programming – with Java
and Visual Interactive support

Kees Huizing Ruurd Kuiper

Preface

This book provides a working knowledge of Object Oriented programming and Java.

Computers are used almost everywhere. Their role is processing data. A computer can perform any data processing concern we desire. We provide the appropriate instructions, a program, written in a programming language. Executing the program on a computer then results in performing the desired data processing concern.

Programs and their executions 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.

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. The emphasis is on concepts rather than on details. Knowledge is build up incrementally, without branching or making de-tours, chapters generally depending on previous ones. 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 address one programming concept. A (learner) programmer is driven by programming aims, and has to (learn to) select the means, the concepts that match the aims. The chapters have a fixed structure.

A section **Aim** makes the programming aim explicit.

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 that shows its workings. This model is more detailed and more explicit than is usually provided. In OOP one's thinking often moves between the static, fixed, program text that is provided as classes, and the dynamically changing objects that are created from the classes when executing, processing the 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. Such a model aids one's thinking about programming, e.g., to understand programming concepts. Also, in discussions about programming issues it is helpful 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, maintain-

ing the structure as just outlined.

A **Remarks** section gathers additional information.

Exercises are provided, separately, for each chapter. To learn OOP, and Java, it is necessary to do a substantial amount of the exercises, preferably right 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.

An execution model viewer, called *CoffeeDregs*, for concrete programs visualizes the execution model on screen in an interactive manner.

Additional web sources

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

- The Java development environment is available from the Oracle website: `http://www.oracle.com/technetwork/java/javase/downloads/` (August, 2011)
- For explanations about Java programming in general, see the SUN Java tutorials: `http://download.oracle.com/javase/tutorial/` (August, 2011)
- For a complete description of the Java language, see The Java Language Specification, third edition: `http://java.sun.com/docs/books/jls/` (August, 2011)
- For information about predefined Java classes and interfaces, see the Application Programmers Interface (API): `http://download.oracle.com/javase/7/docs/api/` (August, 2011)
- A good general book on Java is *Learning Java*, by Niemeyer and Knudsen, O'Reilly, 2005.
- A good free book on the web is `http://math.hws.edu/javanotes/` (August, 2011).

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. Versions 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 7.0.1*, available from http://www.netbeans.org/ .
- To execute Java programs, a run-time environment needs to be present. The Java Development Kit, JDK7, 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* will be made available when ready.

Contents

Part I

Basic sequential programs – One class, one object, one method, local variables

General Aim: Perform a data processing task. Get data from the user as input, store and manipulate the data producing result data, and provide the result data to the user as output.

General Means: One class, one object, one method, local variables. One class is written. One object is instantiated from the class. The method has local variables to store values and uses basic statements for the manipulation.

Chapter 1

Console output – System.out

A *computer* is a machine, *hardware*, that stores and manipulates *data values*, like numbers or text: it performs a *data processing task*. A computer performs a task we desire if we provide it with appropriate written instructions, *software*, for how to perform that concern.

1.1 Aim

Output a text message on the screen.

1.2 Means

We develop in the programming language Java a *program* that contains a (one) *class* that contains a (one) *method* with written instructions for a programming task. These instructions are in terms of *statements* that the computer can perform. We *execute* the program on a computer, which results in an *object* in the computer with a method that performs the task.

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

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. For this purpose, support software is used, called an *Integrated Development Environment* (IDE). Examples of IDEs are: DrJava, NetBeans, Eclipse (supported by IBM) and JCreator. We assume that an IDE has been installed. (See Preliminaries.) Amongst other things, an IDE provides an *editor* which we use to write the program.

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.

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, consists of 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 which, by rule, has the same name as the class, with the extension .java; in our case HelloWorld.java.

```
1 // demonstrates console output
2 public class HelloWorld { // class header (class declaration)
3     // here the class body will be written
4 }
```

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 the class body the statements to perform the task will be written. To enable more structuring of the code later, this code is placed in a sub-container, a *method*. We call the method `show`.

```
1 // demonstrates console output
2 public class HelloWorld {
3
4     // puts message on console
5     void show() { // method header (method declaration)
6         // here the method body will be written
7     }
8 }
```

The text that describes a method also consists of a header and a body. By convention, method names start in lowercase. The parentheses `()` have to be there and identify the code as being a method. `void` in the method header has 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.

```
1 // demonstrates console output
2 public class HelloWorld {
3
4     // puts message on console
5     void show(){
6         System.out.println("Hello world, this is a Java program.");
```

```
7         System.out.println("Good luck, programmer!");
8     }
9 }
```

In this example we use only one type of output statement, `System.out.println`. Between the parentheses we put the text that the output command will put on the screen, called "print" in programmer's jargon. Text to be output 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 in a standard form, by adding the following special piece of start-up code to the class.

```
1     public static void main(String[] args) { // marks start−up code
2         ( new HelloWorld() ).show(); // creates object, starts method
3     }
```

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`. The `show()` part, a *method call*, describes to start the execution of the method `show` of the object. The dot `.` connects the two parts, the semicolon `;` terminates the statement.

`( 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.

By convention, we put the main method 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.

```
1  // demonstrates console output
2  public class HelloWorld {
3
4      // puts message on console
5      void show(){
6          System.out.println("Hello world, this is a Java program.");
7          System.out.println("Good luck, programmer!");
8      }
9
10     // start−up code
11     public static void main(String[] args) {
12         ( new HelloWorld() ).show();
13     }
14 }
```

If we write another class for another data processing concern, the name of that class and the code in the body will be 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.

Syntax template for a class with one method

```
1  // <description of class>
2  public class <class name> {
3      // <descrioption of method>
4          void <method name>() {
5              <statements>
6          }
7
8       // start−up code
9       public static void main(String[] args) {
10          ( new <class name>() ).<method name>();
11      }
12 }
```

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 via a menu option or button with a name such as Compile File or Build File, depending on the particular IDE used. 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 bytecode

To be executable, the human-readable Java code that we have written needs to be translated into machine-executable code, so-called *bytecode*. The compiler also provides this translation: when no syntactic errors occur in the code, compilation automatically produces the machine-executable code. The bytecode 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 bytecode

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

The effect that the execution of a program (after its translation into bytecode) has is called its *semantics*.

We start the actual execution by choosing the Run (or a similar name).

Now the object is created, the method `show` is called and the command statements are executed in the order that they occur in the code.

Output is:

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

The semantics of an output statement thus is that a line of text is output in a fixed text window on the screen where the output-cursor is. This window is called the *console*; this kind of output is called *console output*.

Furthermore, the output-cursor is moved to the next line on output, i.e., possible further output is put on the next line. Then the computer is ready to execute the next statement in the code.

For an output statement it is quite obvious what the semantics is. Essentially, the semantics for the syntax template is that an object is created and the method executed. A more detailed version is given in in terms of the execution model, provided in Section 5.3. This is general practice in the book: in the **Means** Section a brief description of the semantics of a new concept is given that is made more explicit in the **Execution model** section.

1.3 Execution model

In the previous section, two issues about programming showed up. First, a program is a description of how a data processing concern should be performed. The program is a syntactic, static, entity: code that does not change during the execution. Second, 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 involves 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 consider in detail, say at the level of the electronics, what goes on inside the computer. Therefore, a conceptual, pictorial representation of the semantics is given, called the *execution model*.

The execution model represents the execution as a sequence of snapshots, called *states*, connected by *steps*. A state represents the execution-information inside the computer at one moment in time. A step represents the effect of one further action by the computer. The information in a state, combined with the program text, is sufficient to define the next state. Thus, the stepwise execution of a program is represented as the changes in the states. Note, that the program text is available to the executing computer, but is not part of the state.

The execution model for the `HelloWorld` program is as follows.

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, a state that consists of an object that is labeled with the class name, `HelloWorld`, and a number, #90 in this case. This number identifies the object, when, at a later stage, we will have several objects, each one will have a different number. The actual value is chosen by the computer and cannot be used or changed by the 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. Note that the start-up code is not present at all in the object.

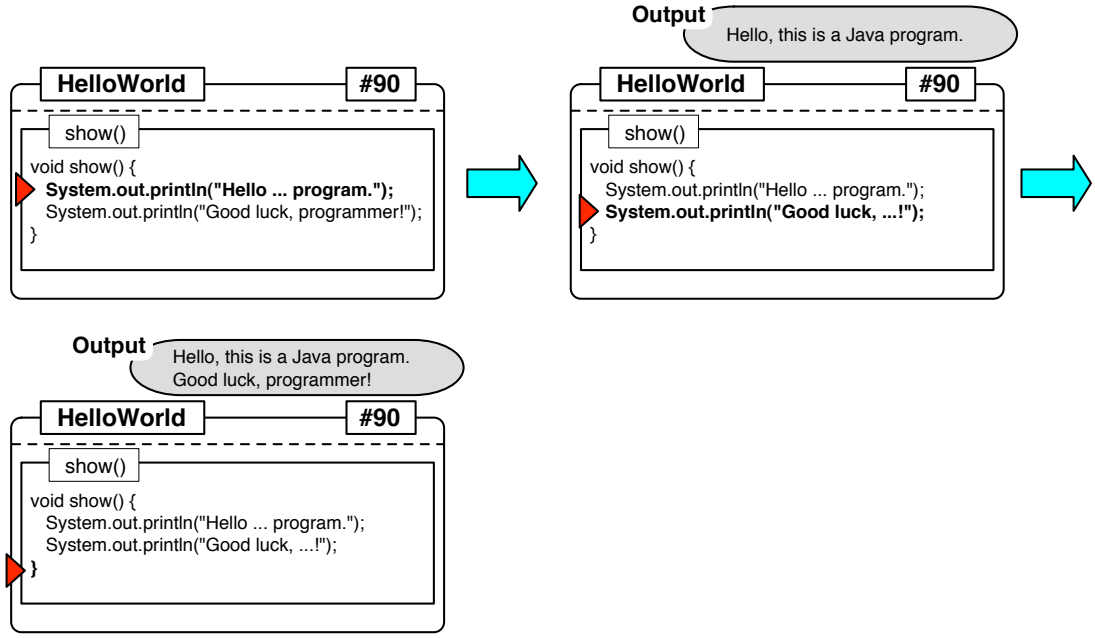
The computer executes one instruction at the time, namely the one where the so-called *control* is at. The control-position information is also part of the state: The statement to be executed next is highlighted and marked with a cursor symbol. The main difference between the code in the class and the text in the dynamic object is the presence of control.

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

The picture shows a sequence of states, beginning immediately after the start-up code is finished. For `HelloWorld`, this is when control is just before the first line of the body of `show`, the output statement. The step models the execution of that statement. For the output statement, the step is defined as follows. In this case, `Hello world, this is a Java program.` is put on the output console. Furthermore, the state change is that control has moved to the next, second, line of the body of `show`. This movement of control is called the *control flow*.

The semantics of a statement is thus modeled as the step.

Figure 1.1: Execution of `HelloWorld`



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

The model viewer, CoffeeDregs, builds the sequence state by state, picturing what happens during execution. The next state is obtained by clicking the Next button. An extra box containing `main` shows up, representing the execution of the start-up code. Just ignore this and click till the `HelloWorld#1` object appears. Also, after control has reached the end of the method `show`, some more clicks make it return to `main` – again, ignore this.

To build the next state, only information from the present state and information from the program code is used - in the example only once, to make the object and put the method in it, at the start. In the viewer the code is therefore present as text on the right side of the window. To make it easy to find the part of the code needed, a bar indicates the code in the program text that corresponds to where control is in the execution model. Note, that the execution model is the result of executing the program code: the code is not part of the execution model.

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 [**?**], 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 The Java Language Specification should become more appreciable when progressing through the book.

1.4.2 Bytecode and the Java Virtual Machine

For many programming 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 *bytecode*, which is an intermediate level of code that is not executed by the computer directly, but by a special program, the *Java Virtual Machine* (JVM). The bytecode is the same for all types of computer.

This standardized intermediate ground between source code and computer has advantages for *portability*, the extent to which programs can be executed by different computers, and for *security*, the protection of the computer 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 may encounter the terms bytecode 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 bytecode. The command to compile source code in, say, the file `HelloWorld.java` to bytecode is `javac HelloWorld.java` which does the syntax check and yields the file `HelloWorld.class` containing the byte code. The command to execute the bytecode stored in `HelloWorld.class` is `java.program`.

Chapter 2

Data values and operations (numeric) – Types int and double

In the previous chapter it was shown how to output 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, *data types* to do this are introduced now. There are several such data types because there are several quite different kinds of data values.

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

This chapter concerns numeric types, which correspond to the familiar notion of calculating with numbers.

2.1 Aim

Calculate new numerical data values from given ones.

2.2 Means

The Java built-in numeric types.

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

With the *syntax* (form) of data types we can build *expressions*, formulas with *semantics* (meaning) such that the computer can calculate their value.

2.2.1 The integral type `int`

The type `int` is used to calculate with integers, i.e., whole numbers.

Syntax

[1] We call these built-in types because a programmer can also define new data types; later we will see how to do this

The syntax consists of *literals*, a notation for data values, and *operators* that take *arguments* to form *compound expressions*. The arguments of an operator can be literals, but also compound expressions, possibly using parentheses.

1. Literals: `...`, `-1`, `0`, `1`, `...`.
2. Operators: `+` addition, `-` subtraction, `*` multiplication, `/` integer division, `%` modulo (remainder).

An example of an expression is `(7 + 17) * 3`.

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

Semantics

1. The semantics of a literal is a *value* in the data type, i.e., a number. These values are represented inside the computer as sequences of bits (0/1-values). 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.
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 the sequence of bits that represents the new value. The operators denote the corresponding operations. 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.

The semantics of an expression is the resulting value produced by successively applying the operations on (intermediate) values, so called *evaluation* of the expression. Like in arithmetic, parentheses may be used to influence the order of evaluation. For example, `(7 + 17) * 3` evaluates to the value `72`. In general, expression evaluation follows the rules that we know from arithmetic. The precise evaluation is explained by means of the execution model, in section 4.3.

Inside the computer, all data values and the manipulations on them are represented as sequences of bits. Usually, it is not relevant for our understanding what exactly the representation is: the only thing to realize is, that inside the computer all syntactical entities are represented in some way that enables the computer to perform the manipulations according to their semantics, for example when an expression is evaluated.

It is important to be aware of the bounded range of `int` values, because if an operation would bring the result outside the range of the data type, the result will nevertheless be within the range of the type anyhow, this giving rise to unexpected outcomes.

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. Note that when an expression is encountered during execution, it is evaluated and its value is then used: the expression is not put on the output console, but its value is.

```
1  // demonstrates integer division
2  public class InterestInt {
3
4    // computes interest
5    void computeInterest() {
6      System.out.print("For a balance of 60 and a rate of 3%, ");
7      System.out.print("interest rounded down to an integer is ");
8      System.out.println((3 * 60)/100);//no "", computes integer (!) expression: 1
9    }
10   public static void main(String[] args) {
11     new InterestInt().computeInterest();
12   }
```

```
13 }
```

Output is:

```
For a balance of 60 and a rate of 3%, interest rounded down to an integer 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 `1.2345E2` denote the same value. The values in `double` range from about $-1.7976931348623157 \cdot 10^{308}$ to $1.7976931348623157 \cdot 10^{308}$ and smallest positive value is about $4.9 \cdot 10^{-324}$.

- The operations behave approximately as in decimal number arithmetic.

The idea of a "floating point" is that the range is maximized and 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.

For ease of presentation, we speak of decimal fractions and decimal point. The computer calculates, however, with *binary* floating point numbers. This difference affects the precision. E.g., the decimal number 0.2 is a recurring fraction when written as a binary number: 0.00110011 ... . Hence, it cannot be exactly represented in a `double`.

Roughly, sixteen digits (53 bits) are available for the number part. The decimal point is put 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, i.e., 0.0144. As we see, the floating point representation achieves one digit more precision than the fixed point representation in this example.

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 types of the operands

determine the operation that will be used. In the case of the division operator, integer division will only be used when both operands are of type `int`, otherwise ordinary division is used.

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

```
1  // demonstrates double division
2  public class InterestDouble {
3
4    // computes interest
5    void computeInterest() {
6      System.out.print("For a balance of 60.0 and a rate of 3.0%, ");
7      System.out.print("interest is ");
8      System.out.println((3.0 * 60.0)/100.0);//computes double (!) expression: 1.8
9    }
10   public static void main(String[] args) {
11     new InterestDouble().computeInterest();
12   }
13 }
```

```
For a balance of 60 and a rate of 3%, interest is 1.8
```

Java has several more numerical types, providing for different requirements as regards range, precision, and amount of storage space used. The numeric types are, ordered according to increasing range, `byte`, `short`, `int`, `long`, `float`, `double`. Of these, `float` and `double` are floating point types, the others are integral 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 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 sub-expressions to determine the type of an expression: it suffices to know the types of the sub-expressions. Type correctness thus can be, and is, checked by the compiler before execution.

2.2.4 Mixing types in an expression

It is allowed that an expression contains more than one type, to some extent. One can, for instance, add an integer to a double. The notion of type correctness and the compiler match the flexibility that allows mixing types to some extent.

In the case of numeric types, the values are automatically converted to the type that has the greatest range or precision. This is called *widening*. Examples:

`7 + 17.0` has type `double` and value `24.0`.

`1000000L * 1000000 / 1000000` has type `long` and value `1000000`.

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

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 operands are associated to operators.

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 in 1.2.2, 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, often even 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, a program may contain errors. These are errors of the third type, the 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. This situation is called *overflow* and, other than with dividing by zero, the program will not be aborted and no message will be issued. The program will silently continue with this wrong result.

This is just one example. There are many different ways to have a program that does not produce the intended result.

Chapter 3

Data values and operations (textual) – Types char and String

This chapter concerns textual types, corresponding to the use of characters, words, sentences, etc.

3.1 Aim

Calculate new textual data values from given ones.

3.2 Means

The Java built-in textual types.

Java has two kinds of built-in textual data types, `String` and `char`. Technically, the latter type is actually a numeric type. This sounds confusing and we believe it *is* confusing and it is one of the reasons we will not use the `char` type directly. For most purposes we can use just the type `String`. (Behind the scenes, `String` uses `char`.)

3.2.1 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 `'&'`; the single quote, the *tab*, and the *newline* are represented by `\'`, `\t`, and `\n`. Numeric literals can also be used. They are interpreted as the ASCII-code of the character.
2. Operators: since the type `char` in Java is technically a numeric type, all numeric operators are applicable, although most of them are not very useful, except for the comparison operator (`==`).

Semantics

A character in single quotes denotes the corresponding character value. For example, `System.out.println('c');` results in the output

```
c
```

3.2.2 The textual type `String`

`String` is a textual type that represents sequences of characters.

Syntax

1. Literals: sequences of characters in double quotes, such as `"Hello world!"` , `"7"` and `"17"`. Note that the special character codes `\t` and `\n` (see above) can be used in String literals as well.
2. Operator: `+` concatenation. The operator `==`, although applicable, is not very useful, since it does not compare String values. Instead, use `equals` (see section 4.2.1).

Semantics

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

If the operator `+` is applied to an argument with a type other than String, this argument is first converted to a String.

So the symbol `+` is overloaded: for `int` and `double` arguments addition is performed, for `String` arguments concatenation is performed. When mixed, the String-interpretation prevails, e.g., `1+"1"` results in the String `"11"`.

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

```
1  // String concatenation
2  class HelloString {
3
4      // concatenates and outputs two Strings
5      void show() {
6          System.out.println("Hello, World, " + "this is a Java program."); //concatenate Strings
7          System.out.println("Good luck, programmer!");
8      }
9
10     // start-up code
11     public static void main(String[] args) {
12         ( new HelloString() ).show();
13     }
14 }
```

Output is

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

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.

3.3 Execution model

No change.

Other String operations

There are several other operations in Java on Strings. These are treated together with the String comparison operators in section 4.2.1.

Chapter 4

Data values and operations (logic) – Type boolean

The current chapter concerns the logic data types, as appear in logic.

4.1 Aim

Calculate logic data values from given ones.

4.2 Means

The Java built-in type `boolean`.

`boolean` is the logic type, having only two values: `true` and `false`. The role of `boolean` in programming is somewhat different from that of the other types. As will be shown in chapters 7 and 8, logic values play a supporting role in programming.

Syntax

1. Literals: `true`, `false`.
2. Operators: `||` (or), `&&` (and), `!` (not), and `^` (exclusive or).

Note that there are also operators `&` and `|`. These behave a little different from `&&` and `||` and are of more specialized 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 4.1 gives an overview with examples.

```
1 // logic expressions
2 class LogicExpressions {
3     // evaluates logic expressions and outputs results
4     void show() {
5         System.out.print("true || false evaluates to ");
6         System.out.println(true || false);
7 
8         System.out.print("true && false evaluates to ");
```

```
 9        System.out.println(true && false);
10
11        System.out.print("! true evaluates to ");
12        System.out.println(! true);
13
14        System.out.print("true ^ true evaluates to ");
15        System.out.println(true ^ true);
16      }
17
18      // start-up code
19      public static void main(String[] args) {
20          ( new LogicExpressions() ).show();
21      }
22  }
```

Output is:

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

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

Figure 4.1: Boolean operators

4.2.1 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 4.2 gives an overview. Note that they work for all numeric types, not for Strings. For alphabetic ordering and (in)equality of Strings, see section 4.2.1.

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 and other operations

The type `String`, as hinted by the uppercase with with the name starts, is somewhat different from the other built-in types in that is technically a class. There are some consequences, most notably the format of the operations and that

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 4.2: Comparison operators (Boolean result)

operator	description	example	result
equals	equal	"ab".equals("a"+"b")	true
	unequal	!"ab".equals("ab")	false
compareTo	lexicographic precedence	"a".compareTo("ab")	-1
		"ab".compareTo("a")	1
		"ab".compareTo("ab")	0
length	number of characters	"joy".length()	3
substring	substring from position up to position	"abcde".substring(2,4)	"cd"
indexOf	position of substring if present	"abcde".indexOf("cd")	2
		"abcde".indexOf("dc")	-1

Figure 4.3: String operations

the numeric comparison operators can not be used. Instead, use *equals* for (in)equality and *compareTo* for alphabetic (lexicographic) ordering.

Below you will find some useful operations on Strings, including the comparison operators.

equals compares two Strings for equality:
`"abc".equals("def")` is false; `"pqr".equals("pq"+"r")` is true;

compareTo compares two String in lexicographical order; the result is an integer: `"abc".compareTo("def")` evaluates to -1
`"def".compareTo("abc")` evaluates to 1
`"abc".compareTo("abc")` evaluates to 0

indexOf gives the position where a given String occurs in the String, -1 when it doesn't occur; the first position is 0; examples:
`"abc".indexOf("bc")` evaluates to 1
`"abc".indexOf("a")` evaluates to 0
`"abc".indexOf("cb")` evaluates to -1

length gives the number of characters in the String:
`"abcd".length()` evaluates to 4
`"".length()` evaluates to `""` (the empty String is denoted by `""`

subString gives the String of characters that starts and ends in the given positions; examples: `"abcd".substring(1,3)` evaluates to `"bc"`
`"abcd".substring(0,1)` evaluates to `"a"` `"abcd".substring(1,1)` evaluates to `""`

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

Remarks The operators == (and !=) can be applied to Strings (they can, in fact, be applied to any Java type), but they do not give the desired result. This also will be explained later.

The following program shows some comparisons.

```
1   // comparison examples
2 class Comparison {
3   // evaluates comparisons and outputs results
4   void compare() {
5
6     System.out.print("boolean expression 7 + 17 > 25 evaluates to ");
7     System.out.println(7 + 17 > 25);
8
9     System.out.print("boolean expression (true || false) == true evaluates to ");
10    System.out.println((true || false) == true);
11
12    System.out.print("boolean expression 'c' == 'C' evaluates to ");
13    System.out.println('c' == 'C'); // these are chars, don't do this with Strings!
14
15    System.out.print("boolean expression \"Hello world\".equals(\"Hello \" + \"world\")
          evaluates to ");
16    System.out.println("Hello, World".equals("Hello, " + "World"));
17  }
18
19  // start-up code
20  public static void main(String[] args) {
21    ( new Comparison() ).compare();
22  }
23 }
```

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.

There are several operations on Strings. We give an overview of the most useful in figure 4.2.1.

4.3 Execution model

No change.

Chapter 5

Data storage – Local variables, assignment

Up till now we could compute new values by writing expressions with given values. Then to do that computation for new values the expressions have to be rewritten with these values. We now want to write the expressions once, in terms of variables, such that new values can be computed for by giving different values to the variables instead of changing the expressions in the program. Also, for some computations intermediate values need to remain available for later use in a the computation. Additional to providing built-in data types for values and operations on them, Java enables storage of values.

5.1 Aim

Store and retrieve values of built-in data types.

5.2 Means

Typed local variables as storage space for values of a built-in type. The *assignment operator*, =, is used to store a value in a variable that can hold values of that type.

As an example, we use variables of 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 method. A declaration of a variable consists of its type followed by the name of the variable, which we choose. At the creation of the object, the necessary storage space for values of that type is reserved.

```
1 // handles account
2 public class AccountInterest {
3
4     // updates balance
5     void update() {
6         double balance; //creates storage variable balance for double value
7         int rate; //creates storage variable rate for int value
8     }
9 }
```

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.

```
1  // handles account
2  public class AccountInterest {
3
4      // updates balance
5      void update() {
6          double balance;
7          int rate;
8
9          balance = 60.00; // storing the balance value
10         rate = 3; // storing the rate value
11     }
12 }
```

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 5.3.

```
1  // handles account
2  public class AccountInterest {
3
4      //updates balance
5      void update() {
6          double balance;
7          int rate;
8
9          balance = 60.00;
10         rate = 3;
11
12         System.out.print("Current balance is ");
13         System.out.println(balance);//retrieving the balance
14         System.out.print("Interest rate is ");
15         System.out.println(rate);//retrieving the rate
16
17         balance = balance + (rate * balance)/100;//retrieving, computing and storing,
18                                                  //using balance twice
19
20         System.out.print("New balance is ");
21         System.out.println(balance);//retrieving the new balance
22     }
23
24     // start-up code
25     public static void main(String[] args) {
26     new AccountInterest().update();
27   }
28 }
```

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 method. 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 is 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.

Local variables have to be assigned a value before being used in an expression.

5.2.1 Mixing types in assignment

As for values, types also allow some flexibility for variables: if a value is to be 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.

5.2.2 Type correctness

Type correctness as introduced for expressions with just values in chapter 2 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, and can be changed, 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.

5.3 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 box 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 the name of which appears on the left hand side of the =-sign.

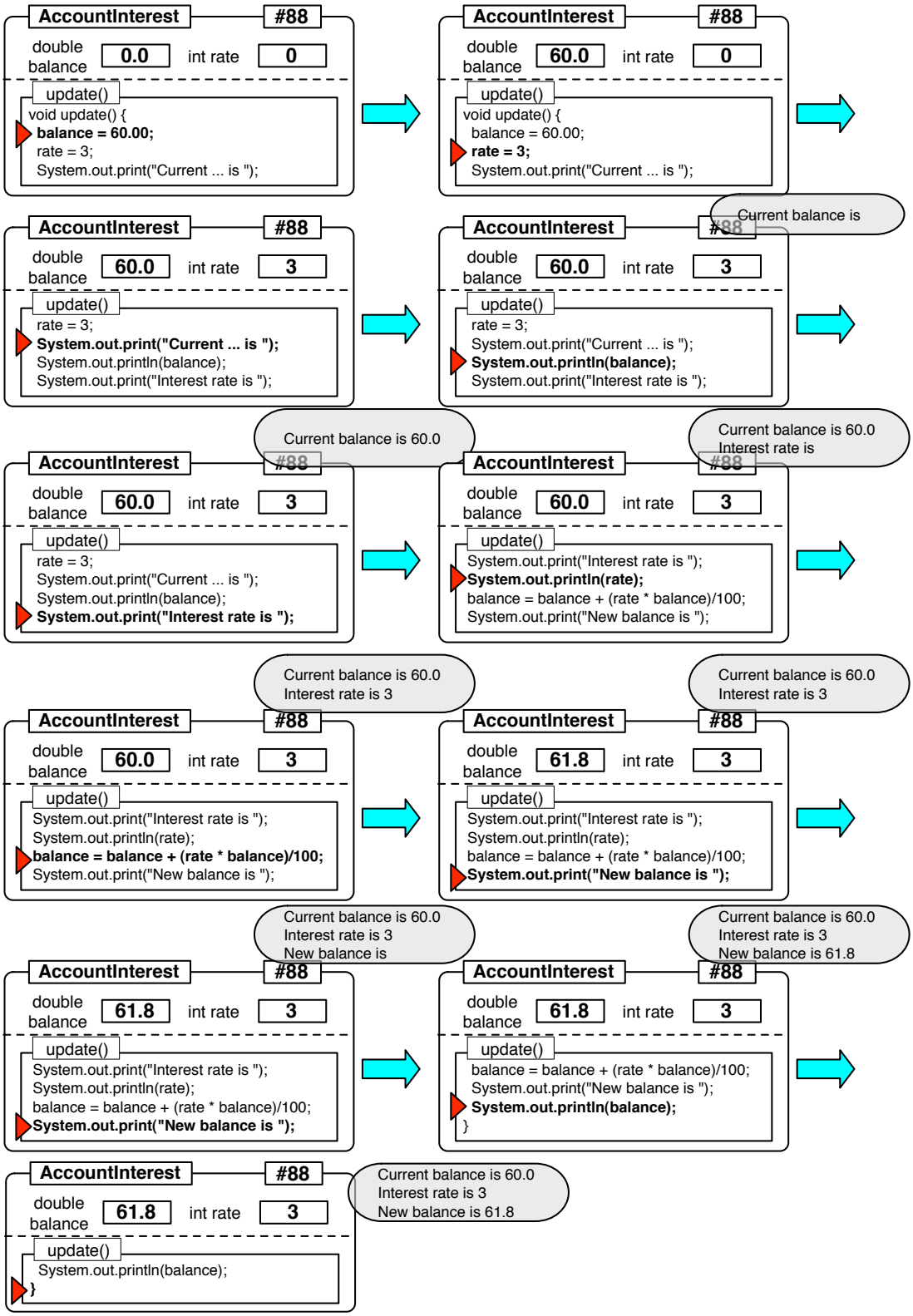
In addition to this, variable names can also occur in expressions. Therefore, evaluation of expressions is extended with retrieving the values stored in variables. When we want to show the execution of an expression in detail we include

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 5.1. NB There is a mistake here: the variables should appear as boxes in the method, not above the method in the class!

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 5.1: Execution of AccountInterest



Chapter 6

Console input – Scanner and System.in

Up till now, variables were initialized through assignment of values that were written in the method code. That meant that each time we want the program to run on different values, we 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. 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.

6.1 Aim

We want to input values from the console.

6.2 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` is required, 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 *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 types 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.

```
1  import java.util.*; // provides Scanner description
2  // handles account
3  public class AccountInput {
4      Scanner sc = new Scanner(System.in); // enables console input via Scanner
5
6      // inputs account info and outputs it using console
7      void infoAccount() {
8          double balance;
9          int rate;
10
11         System.out.println("Type amount for balance and for interest rate");
12         balance = sc.nextDouble(); // inputs balance amount from console
13         rate = sc.nextInt(); // inputs rate from console
14
15         System.out.println("Current balance is " + balance + ".");
16         System.out.println("Interest rate is " + rate + ".");
17
18         balance = balance + (rate * balance)/100;
19
20         System.out.println("One year interest has been added.");
21         System.out.println("Current balance is " + balance + ".");
22     }
23
24     // start-up code
25     public static void main(String[] args) {
26         (new AccountInput()).infoAccount();
```

```
27     }
28 }
```

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.

6.3 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.

6.4 Remarks

6.4.1 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 non-numeric characters are read and `parseInt` is applied, an error message is issued and execution is stopped.

Chapter 7

Choice – if-else

The execution order of statements in a method was determined by the order in which control passes to statements, the *control flow*. Up till now, statements were executed in the order as written, the control flow was sequential. In fact, control was steered by the implicitly used sequential control flow statement `;`. 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. Data dependent control statements for choice make this possible.

7.1 Aim

Select one of two statements for execution, depending on data values that may differ for different executions of the program.

7.2 Means

The `if-else` statement, which selects to which one of two blocks of statements control goes, depending on the value of a *selection guard*. The flow of control is data dependent: which block of statements is chosen for execution depends on data values that may, e.g., differ for executions with different input values. After the chosen block has been executed, control goes to the next statement after the complete `if-else` statement.

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

```
1  import java.util.*;
2  // account with differen rates for credit/debt
3  public class AccountChoice {
4      Scanner sc = new Scanner(System.in);
5
6  // set and update balance
7      void handleBalance() {
8          double balance; // balance of account in some currency
9          int rate; // interest rate in percent
10         int debitRate; // debit rate in percent
11
12         rate = 3;
13         debitRate = 5;
14
15         System.out.println("Type amount for balance");
16         balance = sc.nextDouble();
```

```
17
18          if (balance>=0) { // guard, balance >= 0
19              balance = balance + (rate * balance)/100; //in the black
20              System.out.println(rate + "% interest has been added");
21          } else { // balance < 0
22              balance = balance + (debitRate * balance)/100;//in the red (hence the +)
23              System.out.println(debitRate + "% debit interest has been subtracted");
24          }
25
26          System.out.println("New balance is " + balance);//output balance in either case
27      }
28
29      // start-up code
30      public static void main(String[] args) {
31          ( new AccountChoice()).handleBalance();
32      }
33  }
```

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 control arrives at `if-else` statement, 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. Note, that in the case of more than two alternatives no opening brace is placed between the `else ifs`; this is allowed by the Java syntax and avoids having to place all the corresponding closing brackets at the end of the statement.

Semantics

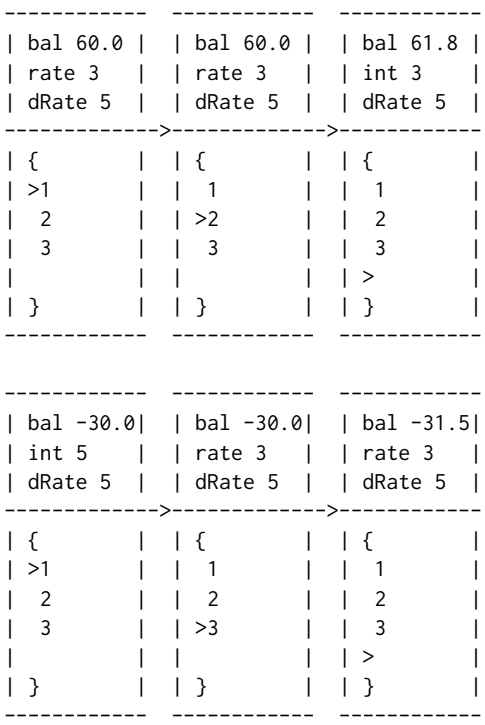
For the standard if-construction with two alternatives, the guard is evaluated, if `boolean_expression` is true, `statements1` is executed, else `statements2` is executed. Then the execution of the `if-else` statement is completed and control goes to the next statement in the program. The semantics for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

7.3 Execution model

In the execution model for the `if-else` statement control moves from the guard to either of two places: if the guard is true, control move to the statement after the guard, if the guard is false, control moves to the statement after the `else`.

Provisional picture, to be replaced We only depict the piece of code considered, with two different start situations, i.e., with two different input values for balance, one solvent, one overdrawn.

Of course the code fragment is part of a method but we don't picture that.



7.4 Remarks

7.4.1 The `switch` statement

It has been shown above how choice between more alternatives can be programmed by repeated use of the `if-else` statement. For cases where the choice depends on discrete values, there is a statement that specifically supports writing elegant code, the `switch` statement which selects one of several statements for execution, depending on the value of a *controlling expression*.

The value of the controlling expression should be of the primitive types `int`, `byte`, `short`), or `char`, or of an enumeration type (not treated in this book). Since Java 7, it can also be of the type String.

In the example below, different operation are performed on an account depending on the choice of the user.

```
 1 import java.util.*;
 2 // allows choice from several actions on account
 3 public class AccountSwitch {
 4   Scanner sc = new Scanner(System.in);
 5
 6   // enables choice of an action on the balance
 7   void handleBalance() {
 8     double balance; // balance of account in some currency
 9     double amount; // amount to use in processing
10     String choice;
11
12     balance = 0;
13
14     System.out.println("Type choice of action (display balance: "+
15                        "'d' or 'b', add: 'a', withdraw: 'w')");
16     choice = sc.next();
17
18     switch (choice) { //takes value stored in choice to select case
```

```
19        case "d":
20        case "b":
21          System.out.println("Current balance is " + balance);
22          break; //exits the switch statement
23        case "a":
24          System.out.println("Type amount to add");
25          amount = sc.nextDouble();
26          balance = balance + amount;
27          break;
28        case "w":
29          System.out.println("Type amount to withdraw");
30          amount = sc.nextDouble();
31          balance = balance - amount;
32          break;
33        default: // covers all other input strings
34          System.out.println("Illegal choice");
35      }
36
37      System.out.println("New balance is " + balance);
38    }
39
40    // start-up code
41    public static void main(String[] args) {
42      ( new AccountPreSwitch() ).handleBalance();
43    }
44  }
```

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

```
Type choice of action (display: d, add: 'a', withdraw: 'w').
a
Type amount to add.
60
New balance is 60.0.
```

When the `switch` statement is executed, first the controlling expression, `(choice)`, is evaluated. In case the value is "a", i.e., an amount should be added to the balance, the statements following the `case "a"` up to the first `break` are executed, adding the amount and displaying the new value of the balance. If a "d" or a "b" is entered the current balance is displayed. This is the consequence of not having a `break` after the `case "d":` label. After execution of the chosen statement, execution of the `switch` statement is completed and control goes to the next statement, displaying the new balance.

Syntax template

`switch(` *controlling expression* `) {`
`case` *value-1* `:`
statements-1
`break;`
`...`
`case` *value-*n_1 `:`
`...`
`case` *value-*n_m `:`
statements-n
`break;`
`case default:`
statements-default
`}`

The `default` label is optional.

Semantics

When the `switch` statement is executed, first the controlling expression is evaluated. If the controlling expression has a value that occurs as a label in the cases list, the group of statements following that label and terminated by the break statement is executed. If the controlling expression has a value that does not occur in the list, the group of statements following the default label is executed. If there is no `default` label and no applicable label the switch statement terminates without execution of any statements.

The execution model is evident.

Chapter 8

Repetition – while

Up till now, statements were executed in the order as written, possibly skipping over statements according to the control flow governed by choice statements. This is still too restrictive, as depending on the data values it might be the case that certain statements should be executed repeatedly. Data dependent control statements for repetition make this possible.

8.1 Aim

Execute a statement repeatedly, where how often the statement is executed depends on variables that change during the execution of the statement.

8.2 Means

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 a balance that is input is shown, until the owner is a millionaire.

```
 1 import java.util.*;
 2 // calculates account build-up
 3 public class AccountCompound {
 4     Scanner sc = new Scanner(System.in);
 5
 6     // shows compound interest build-up
 7     void handleBalance() {
 8         double balance;
 9         double rate;
10         int years;
11         int i; // years counter
12
13         System.out.println("Type amount for balance, rate and interest years");
14         balance = sc.nextDouble();
15         rate = sc.nextDouble();
16         years = sc.nextInt();
17
18         i = 0;
19         while (i < years) {
```

```
20              System.out.println("Balance after "+i+" years: "+Math.round(balance));
21              i = i + 1;
22              balance = balance + (rate * balance)/100;
23          }
24
25          System.out.println("balance after "+i+" years: "+Math.round(balance));
26      }
27
28  // main instantiates and uses the object
29      public static void main(String[] args) {
30          new AccountCompound().handleBalance();
31      }
32  }
```

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, first the guard, `balance<1000000` is evaluated. If it is *true*, the body

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

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

In case the guard 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.

8.3 Execution model

In the execution model for the `while` statement control moves from the end of the loop body back to the guard: if the guard is true, control move to the body, if the guard is false, control moves to end of the `while` statement.

8.4 Remarks

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.

8.4.1 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.

```
1  import java.util.*;
2
3  // calcualates balance build-up, one or more deposits
4  public class AccountDoWhile {
5      Scanner sc = new Scanner(System.in);
6
7      // shows build-up
8      void add() {
9          double balance;
10         double deposit;
11         char choice;
12
13         balance = 0;
14
15         do {
16             System.out.println("Type amount for deposit.");
17             deposit = sc.nextDouble();
18             balance = balance + deposit;
19             System.out.println("Balance is " + Math.round(balance) + ".");
20
21             System.out.println("Do you want to make a deposit? Type y/n");
22             choice = sc.next().charAt(0);
23         } while (choice == 'y');
24     }
25
26 // main instantiates and uses the object
27     public static void main(String[] args) {
28         new AccountDoWhile().add();
29     }
30 }
```

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 construct 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.

```
do {
  Body
} while (Guard);
```

and

```
Body
while (Guard) {
  Body
}
```

are equivalent and

```
while (Guard) {
  Body
}
```

and

```
do {
  if (Guard){
    Body
  }
} while (Guard);
```

8.4.2 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:

```
1 for (int i = 0; i < 10; i = i+1) {
2   balance = balance + (5 * balance)/100)
3   System.out.println("balance is " + balance);
4 }
```

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.

8.4.3 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.

8.4.4 Computability

Not only do the above statements provide 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.

Chapter 9

Indexed lists of variables – Arrays of variables

Up till now, only the built-in types were used. To enable structured storage there is a type constructor in Java to make new types from already available ones.

9.1 Aim

We want to store several data items of the same type in a list that can be accessed using indices.

9.2 Means

We use the *array* construct to make a new type that is a list of fixed length of variables of an already available type, with integer indices. To use such a type involves three things.

1. Definition of the new type, an array of the desired type. Java provides variables of the new array type if `[]` is put behind the type of the desired items (see the example).
2. Declaration of a variable of the new type. (Step 1 and 2 are performed together.)
3. Creation of an array of the desired type. Java provides the new array type if `[]` is put behind the type of the desired items. The length, i.e., the number of elements, of the array is set here, by putting it between the `[]`. After creation the length of an array cannot be changed anymore.

The indexing starts with 0. Elements of an array are accessed by writing the array name and the index, say i: ⟨*arrayname*⟩`[`i`]`. Each element is a variable of the given type, so all operations of that type can be performed on it, and also assignment.

Provided with an array is the variable ⟨`arrayname`⟩`.length`, that contains the length of the array. Note, that since the indexing starts at 0, the last index in the array is the length minus 1.

As an example we write a program that stores numbers of type `int` in an array `numbers`.

```
1  // handles array of integer numbers
2  import java.util.*;
3
4  class ArrayDemo {
5    Scanner sc = new Scanner(System.in);
6
7    void demo(){
8      int[] numbers; // defines and declares int[] array numbers
9      numbers = new int[5]; // creation of the array numbers
10
11     for (int i = 0; i < numbers.length; i++) {
12       System.out.println("Type array element with index " + i);
13       numbers[i] = sc.nextInt(); // initialization of the elements of numbers
14     }
15
16     System.out.println("First array element is " + numbers[0]);
17     System.out.println("Array element with index 2 is " + numbers[2]);
18     System.out.println("Last array element is " + numbers[numbers.length - 1]);
19   }
20
21   public static void main(String[] args) {
22     new ArrayDemo().demo();
23   }
24 }
```

NB In the example we put the creation of the array in the method. It is also possible, to create the array outside the method, when the variable is declared: `int[] numbers = new int[10];` Java does not allow to do declaration and creation as separate steps outside the method.

Arrays of more dimensions are also possible in Java: for example, a 2-dimensional array of integers is defined as an array of arrays through `int[][]`. Creation then is split over several statements. First, a statement where between the first `[]` pair the number of rows is written, leaving the next `[]` pair empty. Second, statements for each row where between the next `[]` pair the number of columns for that row, i.e., the length of the row-array, is written. Numbering starts from the left-hand upper corner, top to bottom for rows (!), left to right for columns.

Also, two dimensional arrays with rows of unequal length are possible: ragged arrays. The number of row-arrays is obtained through ⟨*arrayname*⟩`.length`, the number of elements in each of the row-arrays `i` through ⟨*arrayname*⟩`[i].length`.

Chapter 10

Resizable indexed lists of variables – ArrayList of variables

The size (length) of an array in Java is fixed on creation. After that, the size can not be altered[1]. Often, it is not known how many elements are needed when the array is created and started to be used, e.g., when a sequence of values is read from input and needs to be stored.

10.1 Aim

Have resizable arrays.

10.2 Means

The class *ArrayList* provides the functionality of an array (storing and retrieving elements of the same type by index) plus the possibility of adding and removing elements, which makes the structure grow and shrink automatically. It is part of the *Java Collections Classes*, that will be treated in more depth later in this book.

package: `import java.util.ArrayList;`

Declaration: `ArrayList<E> variableName;`, where `E` is a the type of the elements to be stored

Important methods of **ArrayList**⟨**E**⟩ are:

`E get(int i)` returns the element at position *i*;

`E set(int i, E e)` changes the element at position *i* to *e*; returns the element that was previously at position *i*;

`void add(int i, E e)` shifts the elements at position *i* and elements after that one place; inserts element *e* at position *i*;

`void add(E e)` appends element *e* at the end of the list;

`int size()` returns the number of elements in the list;

Example

```
1 // reads words from input, terminated by __END__, that is not part of the sequence
2 // and prints the words that are longer than the average word length of the sequence
3 // assumes that the sequence is not empty
```

[1] A new array can be created and assigned to the original variable. This will destroy the old values.

```
 4
 5 import java.util.Scanner;
 6 import java.util.ArrayList;
 7
 8 class ArrayListExample {
 9   Scanner scanner = new Scanner(System.in);
10
11   int n;
12   ArrayList<String> words;
13   String word;
14   double totallength = 0;
15   double averagelength;
16   void aboveAveraging() {
17
18
19     words = new ArrayList<String>( );
20     do {
21       word = scanner.next( );
22       totallength += word.length();
23       words.add(word);
24     } while (! scanner.hasNext("__END__"));
25
26     averagelength = totallength / words.size();
27
28     for (String w : words) {
29       if (w.length() > averagelength) {
30         System.out.println(w+" is above average");
31       }
32     }
33   }
34
35   public static void main(String[] args) {
36     new ArrayListExample().aboveAveraging();
37   }
38 }
```

10.2.1 Storing primitive types

Type parameters can only be class types, not primitive types. Therefore, values of type `int`, `double`, `boolean`, etc. can not be directly stored in collections classes. The solution is to put them in an object that just contains an instance variable of the appropriate type. Java provides *wrapper classes* for this purpose. They have the primitive type, written with a capital initial letter, i.e., `Double`, `Boolean`, etc. The wrapper class for `int` is `Integer`.

Conversion from value to object can be done with the constructor, e.g,

```
Integer wrap = new Integer(37);
```

and the other way around with a method:

```
int value = wrap.intValue();
```

Fortunately, this conversion is done by the compiler for you.

Apart from a method to retrieve the value that is stored in the object, the wrapper classes provide some other useful methods concerning the type. See the API for further information.

Part II

Structured sequential programs – One class, one object, multiple methods, instance and local variables

General Aim: Decompose the data manipulation of a data processing concern into tasks. The tasks possibly work on shared data. When writing code for one task, abstract from the others.

General Means: Write a method for each task and compose these to perform the task. Methods share the processed data through instance variables. When writing a method, abstract from the others, using local variables.

Chapter 11

Decomposition into subtaks without shared data– Methods, local variables

Up till now we wrote the data manipulation task in a class in just one method. When the task is large, it may be difficult to write, understand, and maintain the code for it.

When we write a large, complex program, a powerful general strategy is to *decompose* the program in more manageable subtasks. Then we can write methods for these parts separately and (re-)use them to put a complete program together. *Composition*: making the whole from less complex parts that put together form the whole.

To reduce complexity, writing a separate part should be simpler than writing the whole directly. The descriptions of parts therefore need to be at the appropriate level of *abstraction*, i.e., providing sufficient detail for us to write the part, but omitting detail that is not necessary for it. That such an abstraction can be made, depends on choosing coherent and relatively independent parts.

Decomposition and abstraction is now applied for the case that manipulation and data together can be decomposed into independent subtasks.

11.1 Aim

Decompose, if possible, a task into subtasks that do not share data. When writing one subtask, abstract from the others.

11.2 Means

Write a method for each subtask and compose these to perform the task. When writing a method, make it more abstract (less dependent from the others) using local variables.

Applying the idea of composition, we declare and define in a class several methods, with different names, that each contain the data and code for a subtask. One method is, as usual, started from `main`. This starts the processing; from this method others are started, *called*, these may call other methods, and so on. A method call takes place when control arrives at a statement that consists of the method name followed by a (for now empty) pair of parentheses. In this way the parts are combined.

The process of defining methods is called “grouping and naming”: grouping of code in a method that has a name by which it can be called. This code can be used repeatedly and at different places in the code by writing a call where execution of the grouped code is desired.

Applying the idea of abstraction, we write a short description of the effect of the method, which in the case of independent data parts can only by in terms of I/O.

How the method accomplishes the effect and what the intermediate values of local variables are is abstracted from.

Example:

```
1  import java.util.*;
2  // handles account, decomposition into two methods, local variable balance
3  public class AccountDecomposedLocal {
4
5      // shows a welcome
6      void welcome() {
7          System.out.println("Welcome at the bank");
8      }
9
10     // puts 10 into the account and shows the amount
11     void initializeAndShow() {
12         double balance; //local variable
13
14         balance = 10;
15         System.out.println("balance = " + balance);
16     }
17
18     // handles the account
19     void handleAccount() {
20         welcome(); //calls welcome, i.e., initiates its execution
21         welcome(); //calls welcome
22         initializeAndShow(); //calls initializeAndShow
23     }
24     // start-up code
25     public static void main(String[] args) {
26         new AccountDecomposedLocal().handleAccount(); //calls handleAccount
27     }
28 }
```

Output is (input in bold) :

```
Welcome at the bank
Welcome at the bank
balance = 10.0
```

We declare and define methods in the class, as before, but now several rather than just one. The class has three methods, `welcome`, `initializeAndShow`, and `handleAccount`.

We obtain the code for the whole task as the composition of the methods for the subtasks. The composition is subtle, in the sense that there is no separate combination operator: we combine the descriptions of the parts through the calls, in this case inside one of them, `handleAccount`. The same method can be called several times, e.g., `welcome` is called twice here.

`handleAccount` is executed first, by calling it in the `main`.

The abstraction is that to write each of the methods we only have to consider the abstract descriptions of the other methods. We provide the abstract descriptions as a comment above each method that describes its activity. The code then needs not be read.

When sensible names are chosen for the methods, often even just these provide enough information.

11.3 Execution model

graphics to be provided

The execution model reflects that a method becomes part of the object when it is called. In the start-up code the method `handleAccount` is called.

From then on, further calls are made from the methods in the object, like `welcome` being called from `handleAccount` Also the method from which the call is made, here `handleAccount` remains part of the object. When a method is executed, a new *method execution box* is drawn inside the object. The body of the method is displayed inside the box, below the dotted line. The control marker is put at the beginning of the body of the called method. A *passive control marker* is left at the place of the call and signals the place where control has to return when the called method is finished. After the method that has been called finishes, the method execution box is removed from the object and the passive marker becomes active again, putting control just after the method call.

So the object has a more permanent existence than the methods, notably the local variable values are only preserved for the duration of the method execution.

In the same way as partitioning in methods was used in writing the description of the manipulation task of the class, this partitioning is helpful in understanding the execution. As shown in the example, the existence of methods in an object may vary during execution – this is not directly seen from the class description but it is reflected in the execution model.

Chapter 12

Decomposition into subtasks with shared data – Methods, instance variables

Up till now we could decompose manipulation and data together into subtasks and implement these as separate methods.

Often the manipulation on some data is so complex, that decomposition of the manipulation into subtasks that manipulate shared data would help to manage the complexity.

Composition and abstraction is now applied for the case that manipulation can be decomposed in subtasks but these tasks operate on shared data.

12.1 Aim

Decompose a task into subtasks that share data.

12.2 Means

Write a method for each subtask and compose these to perform the task. Use *instance* variables to store data that the methods share.

Instance variables are declared in the container that also contains the methods, i.e., the class. By convention, we declare them at the top of the class.

The various methods manipulate the data stored in the instance variables of the class. These variables are accessible for all methods: it is the data manipulation that is partitioned, not the data storage.

Applying the idea of abstraction, we write a short description of the effect of the method on the instance variables as a comment just before the method declaration. Because the method only uses and changes the instances variables and is executed as a whole, only the change to the values of these instance variables from the moment of the call till the return is described. How the method accomplishes this and what the intermediate values are is abstracted from. As before, local variables are not used in the description.

```
1 import java.util.*;
2
```

```
 3 // handles account, decomposition into two methods, instance variable balance
 4 public class AccountDecomposedInstance {
 5     double balance;// instance variable – shared
 6
 7     //initializes balance
 8     void initialize() {
 9         balance = 10;
10     }
11
12     //shows balance on the console
13     void show() {
14         System.out.println("balance = " + balance);
15     }
16
17     // handles balance
18     void handleBalance() {
19         initialize();
20         show();
21     }
22
23     // start–up code
24     public static void main(String[] args) {
25         new AccountDecomposedInstance().handleBalance();
26     }
27 }
```

Output is (input in bold) :

```
balance = 10.0
```

The class has three methods, `initialize`, `show`, and `handleBalance`.

The balance data is shared between `initialize` and `show`, and therefore the variable `balance` is an instance variable, accessible for both methods.

The abstraction is that to write each of the methods we only have to consider the instance variables and abstract descriptions of the other methods. We provide the abstract descriptions as a comment above each method that describes its activity. Ideally, the code then needs not be read. Note that the instance variable `balance` is mentioned. When sensible names are chosen for the methods, often even just these provide enough information.

12.3 Execution model

graphics to be provided

The instances variables are visualized as boxes with values at the top of the object. The methods can take values from these as well as put values into these. The execution model reflects that the instance variables have the same permanence as the object.

12.4 Examples

The following example show how local variables can be used for non shared data and instance variables for shared.

```java
import java.util.*;
// handles account, partitioning into more methods
public class AccountPartitioned {
  Scanner sc = new Scanner(System.in);

  double balance;//instance variables!
  int rate;

  // initializes
  void initialize() {
    System.out.println("Provide amount for balance and rate");
    balance = sc.nextDouble();
    rate = sc.nextInt();
  }

  // shows balance
  void showBalance() {
    System.out.println("Balance is " + balance);
  }

  // updates balance
  void updateBalance() {
    balance = balance + (rate * balance)/100;
    System.out.println(rate +" % interest has been added");
  }

  // handles balance
  void handleBalance() {
    initialize(); // call to method initialize
    showBalance(); // call to method showBalance
    updateBalance(); // call to method updateBalance
    showBalance(); // call to method showBalance
    updateBalance(); // second call to method updateBalance
    showBalance(); // second call to method showBalance
  }

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

Output is (input in bold) :

```
Type amount for balance.
60
Balance is 60.0.
5 % interest has been added.
5 % interest has been added.
Balance is 66.15.
```

Chapter 13

Methods with flexible input – Parameters

Up till now, a method acted on values stored in instance variables of the object. This makes it not so flexible as regards selecting the values with which the method performs the task: these have to be stored in these fixed variables. Often the data manipulation task that a method performs is useful to apply for various values that are not all stored in these fixed places, but that rather are selected at different applications of the method.

Because a method provides abstraction, a language feature is introduced to explicitly provide values to special local variables of the method.

13.1 Aim

We want to provide data to a method explicitly in a flexible manner. This makes the method more abstract in the sense the input variables are not fixed but that input values can be chosen at the call; that also makes the method more transparent in the sense that the inputs are explicit.

13.2 Means

We provide data to a method through a *parameter*.

The idea to achieve this is to have special local variables in the method that contain the desired start values when the method starts. Then the method executes using these start values.

In the header of the method, inside the parentheses after its name, *formal parameters* and their types are added. A formal parameter can be used in the body of the method as a local variable, for example on the right hand side of an assignment. It can be viewed as a local variable, with one difference: it is initialized at the call. In the method call, we fill in expressions between the parentheses: the *actual parameters* or *arguments*. When the method is executed, as the first action the value of the actual parameters is assigned to the formal parameters in the body. The number and types of the actual parameters has to match the list of formal parameters exactly. The order in which the actual parameters appear in the call determines to which formal parameters their values are assigned.

In the example below the interest rate is a parameter to the method `computeInterest`.

```
1  import java.util.*;
2
3  // handles account
4  class AccountParameter {
5      Scanner sc = new Scanner(System.in);
6
7      double balance;
8      int rate;
9      double interest;
10
11     // computes interest
12         void computeInterest(int r) {// parameter r for flexible choice of rate
13         interest = (r * balance)/100;
14     }
15
16     void handleBalance() {
17          System.out.println("Type amount for balance.");
18          balance = sc.nextDouble();
19
20          System.out.println("Balance is " + balance + ".");
21
22          computeInterest(3); //call with value
23          System.out.println("Interest for rate 3 is " + interest + "."); // call to function interest(3)
24
25          computeInterest(7); //call with value
26          System.out.println("Interest for rate 7 is " + interest + "."); // call to function interest(7)
27
28          System.out.println("Type amount for interest rate.");
29          rate = sc.nextInt();
30          computeInterest(rate); // call with variable
31          System.out.println("Interest for rate " + rate + " is " + interest + "."); // call to function
                interest(rate)
32     }
33
34     //start-up code
35     public static void main(String[] args) {
36          new AccountParameter().handleBalance();
37     }
38 }
```

Output is (input in bold):

```
Type amount for balance.
 60
Balance is 60.0
Interest for rate 3 is 1.8
Interest for rate 7 is 4.2
Type amount for interest rate.
 5
Interest for rate 5 is 3.0
```

13.3 Execution model

Parameters are drawn as variable boxes above the dotted line in method executions. To distinguish them from local variables, their name is prefixed with a @-character. When the method execution is created, the

parameter boxes are filled with the value of the actual parameters.

Chapter 14

Methods with flexible output – Return values, function

Up till now, the result values of a method were stored in instance variables of the object. This makes it not so flexible as regards selecting the values that the method computes: these have to be stored in these fixed variables. Often the results of the data manipulation task that a method performs are useful at places that differ from these fixed places, but that rather are selected at different applications of the method.

Because a method provides abstraction in he sense that local variables are not accessible outside the method, a language feature is introduced to let a value be provided explicitly by the method.

14.1 Aim

We want to let a method provide data explicitly in a flexible manner. This makes the method more abstract in the sense that the output variables are not fixed but that where a value is used can be chosen at the call. It also makes the procedure more transparent in the sense that the output is explicit.

14.2 Means

We obtain data from a method through the `return` statement of that method. At the end of the method body, a `return` statement is added, consisting of the word `return` followed by an expression that has the desired data value, terminated by a semicolon.

We say that the value of this expression is *returned by the method* and the effect is that this value is filled in for the method call in the expression where the call appears. This value is used in the evaluation of the expression. The type of the returned value is given at the header of the method definition: instead of `void`, the type is written.

A method that contains a return is called a *function*. To distinguish between functions and methods without a return, i.e., of type `void`, the latter are sometimes called *procedures*.

Note that, in contrast to the input case where more than one parameter is possible, only one value can be returned. This enables to us functions in expressions.

In the example below the interest is returned by the method `giveInterest` and can either be displayed to be collected or added to the balance.

Note, that the two mechanisms, parameters and return value, are independent: as shown, a method can have parameters without having a return value and vice versa; a method can also have neither or both; they

can also be combined as will be shown later.

```
1  import java.util.*;
2
3  // handles account
4  class AccountReturn {
5      Scanner sc = new Scanner(System.in);
6      double balance;
7      int rate;
8
9      // initializes
10     void initialize() {
11         System.out.println("Type amount for balance.");
12         balance = sc.nextDouble();
13
14         System.out.println("Type amount for interest rate.");
15         rate = sc.nextInt();
16     }
17
18     // shows balance
19     void showBalance() {
20         System.out.println("Balance is " + balance);
21     }
22
23     // compute interest at stored rate
24     double computeNewBalance() {
25         return balance + (rate * balance)/100; // returns balance with interest
26     }
27
28     void handleBalance() {
29         String choice;
30
31         initialize();
32
33         System.out.println("For balance to be updated with interest, type 'u'.");
34         System.out.println("For balance plus interest just to be displayed, type any other
               letter.");
35         choice = sc.next();
36
37         if ( choice.equals("u") ) {
38             balance = computeNewBalance();
39         } else {
40             System.out.println("Balance plus interest would be " + computeNewBalance());
41         }
42
43         showBalance();
44     }
45
46     //start-up code
47     public static void main(String[] args) {
48         new AccountReturn().handleBalance();
49     }
50 }
```

Output is (input in bold):

```
Type amount for balance.
60
```

```
Type amount for interest rate.
 3
Balance to be updated with interest, type 'u'.
Balance plus interest just to be displayed, type any other letter.
 u
Balance is 61.8
```

or

```
Type amount for balance.
 60
Type amount for interest rate.
 3
Balance to be updated with interest, type 'u'.
Balance plus interest just to be displayed, type any other letter.
 d
Balance plus interest would be 61.8
Balance is 60.0
```

In the first case, the return value of `computeNewBalance` is assigned to the variable `balance`. In the second case, the return value is put on the console.

14.2.1 Side effect free functions

A function is a special method that returns a value. This feature makes a function fit to be used in expressions, in a similar fashion as in mathematics. Note, that apart from returning a value a function may also change an instance variable. This runs counter to the intuition of a mathematical function. Therefore, a class of functions, called *side effect free* functions is defined as the functions that do not change any instance variable value.

Furthermore, the side effect free functions are easy to use in programming, as only their dependence on instance variables has to be taken into account, not their effect on them.

We adhere to the style of only using side effect free functions.

The function in the example, `computeNewBalance`, is side effect free.

14.2.2 Pure functions

The functions we use are required to be side effect free, i.e., using the values from instance variables but not changing them. Using parameters, this independence can be extended to also not using values from instance variables, namely by only obtaining values through parameters. Functions in which only parameters occur are called *pure* functions.

Mathematical functions are pure: all values are provided through parameters, giving rise to one result.

An example of a function that is pure is `raise(int n, int p)` that computes n^p.

```
1 import java.util.*;
2
3 // handles account
4 public class RaiseDemo {
5       Scanner sc = new Scanner(System.in);
6
7       int number, power;
8
```

```
 9     // compute n^p
10     int raise(int n, int p) {
11         int i, result;
12
13         result = 1;
14         i = 0;
15         while (i < p) {
16             result = result * n;
17             i = i + 1;
18         }
19         return result;
20     }
21
22     // demo
23     void demo() {
24          System.out.println("Give number and power");
25          number = sc.nextInt();
26          power = sc.nextInt();
27
28          System.out.println(number + "^" + power + " = " + raise(number, power));
29     }
30
31     // main instantiates and uses the object
32     public static void main(String[] args) {
33         new RaiseDemo().demo();
34     }
35 }
```

Pure functions can thus be described in terms of the parameters only. For example, `square(int x)`is described by the return value being x^2.

Note, that to compute the return value as a function of the input parameter values often just using an expression does not suffice: an algorithm might be required that uses local variables to store intermediate values. This does not affects the side effect freeness or purity of a function: the only requirements are that no instance variable values are changed, respectively changed or used.

Note that pure implies side effect free, but not vice versa: the function `interest` from the example uses the instance variable `balance`.

14.3 Execution model

When a method with return value terminates, the return value is used in the evaluation of the expression in which the call is made. The value is not directly shown in the model, it is indirectly visible in the effect that it has in, e.g., the new value of a variable.

Chapter 15

Method recursion – Recursive methods

Some data manipulation tasks can, by their nature, best be performed as performing some small manipulation task on the result of first performing the same task on simpler data. This is repeated until such a simple task results that it can be performed directly. The small manipulation tasks created in this process are then performed, in the order opposite of the one in which they were created.

This is the *recursive* (recurring to itself) execution of a task: a recursive algorithm. A task is called recursive by nature if a recursive algorithm is the easiest kind of algorithm to design for it.

15.1 Aim

We want to perform a task recursively (the simplification usually being in terms of the data).

15.2 Means

We write a *recursive* method for the task. The method body firstly contains code that describes the recursive part of the task, followed by a call to the method itself, with simpler data. This part is executed when the simplest case is not reached yet. Secondly, the method body contains code that describes how to directly perform the task for the simplest case. This code is executed when the method is called with data corresponding to the simplest case.

The data that is relevant for the recursive execution of the task, and thus has to become simpler at each recursive call, is usually provided through parameters. To enable use of the results of the calls, in building the total result, the results of the calls are usually provided through the return.

NB A recursively performed task can always also be performed using a, different, iterative algorithm. Translation often is not easy. Usually an iterative algorithm is more efficient, especially in terms of memory use, than a recursive one.

In the example below, a recursive method is used to compute the factorial of an integer number.

```
1 import java.util.*;
2
3 class Factorial{
4   Scanner sc = new Scanner(System.in);
5
6   long factorial(int n){
7     if (n==0)
```

```
 8         return 1;
 9       else
10         return n*factorial(n-1);
11     }
12
13   public void demo(){
14     int n;
15
16     System.out.println("Provide a number to compute the factorial for");
17     n = sc.nextInt();
18     System.out.println("Factorial of " + n + " is " + factorial(n));
19     }
20
21 // main instantiates and uses the object
22   public static void main(String[] args) {
23     new Factorial().demo();
24   }
25 }
```

An alternative is, to not store each intermediate values in a separate variable, as is the case in the above approach using parameters and returns, but re-use the variables at each step. This involves undoing value changes as in the following adaptation of factorial.

```
 1 import java.util.*;
 2
 3 class FactorialVoidNoParameter{
 4   Scanner sc = new Scanner(System.in);
 5
 6   int n;
 7   long fac;
 8
 9   void factorial(){
10     if (n==0)
11         fac = 1;
12     else{
13         n=n-1;
14         factorial();
15         n=n+1;
16         fac=n*fac;
17     }
18   }
19
20   public void demo(){
21
22     System.out.println("Provide a number to compute the factorial for");
23     n = sc.nextInt();
24     factorial();
25     System.out.println("Factorial of " + n + " is " + fac);
26     }
27
28 // main instantiates and uses the object
29   public static void main(String[] args) {
30     new FactorialVoidNoParameter().demo();
31   }
32 }
```

15.3 Execution model

Each recursive call gives rise to a new instantiation of the method, until the simplest case is reached. When, starting with the simplest case, the execution of some method is completed, the result is returned to the previous method and the returning method disappears.

```
----------------------   --------------------
|>fac(3){            |   | fac(3){          |
| if (n==0)          |   | if (3==0)        |
|  return 1;         |->-|  return 1;       |->
| else               |   | else             |
|  return n*fac(n-1);|   |> return 3*fac(2);|
| }                  |   | }                |
----------------------   --------------------
----------------------------------------
| fac(3){            >fac(2){           |
| if (3==0)           if (n==0)         |
|  return 1;           return 1;        |->
| else                else               |
|- return 3*fac(2);    return n*fac(n-1);|
| }                   }                  |
----------------------------------------
---------------------------------------
| fac(3){             fac(2){          |
| if (3==0)           if (2==0)        |
|  return 1;           return 1;       |->
| else                else             |
|- return 3*fac(2);  > return 2*fac(1);|
| }                   }                |
---------------------------------------
-------------------------------------------------------------
| fac(3){             fac(2){           >fac(1){          |
| if (3==0)           if (2==0)          if (1==0)        |
|  return 1;           return 1;          return 1;       |->
| else                else               else             |
|- return 3*fac(2);  - return 2*fac(1);  - return 1*fac(0);|
| }                   }                  }                |
-------------------------------------------------------------
-------------------------------------------------------------
| fac(3){             fac(2){             fac(1){          |
| if (3==0)           if (2==0)           if (1==0)        |
|  return 1;           return 1;          return 1;       |->
| else                else                else             |
|- return 3*fac(2);  - return 2*fac(1);  > return 1*fac(0);|
| }                   }                   }                |
-------------------------------------------------------------
-----------------------------------------------------------------------------
| fac(3){             fac(2){             fac(1){            >fac(0){          |
| if (3==0)           if (2==0)           if (1==0)          if (0==0)         |
|  return 1;           return 1;          return 1;          return 1;        |->
| else                else                else               else             |
|- return 3*fac(2);  - return 2*fac(1);  - return 1*fac(0);   return 0*fac(-1);|
| }                   }                   }                  }                |
```

```
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| fac(3){             fac(2){             fac(1){             fac(0){           |
| if (3==0)           if (2==0)           if (1==0)           if (0==0)         |
|  return 1;           return 1;           return 1;         > return 1;       |->
| else                else                else                else              |
|- return 3*fac(2);  - return 2*fac(1);  - return 1*fac(0);    return 0*fac(-1);|
| }                   }                   }                   }               |
-------------------------------------------------------------------------------
-----------------------------------------------------
| fac(3){             fac(2){             fac(1){     |
| if (3==0)           if (2==0)           if (1==0)   |
|  return 1;           return 1;           return 1;  |->
| else                else                else        |
|- return 3*fac(2);  - return 2*fac(1);  > return 1*1;|
| }                   }                   }          |
-----------------------------------------------------
-----------------------------------   ---------------
| fac(3){             fac(2){     |   | fac(3){     |
| if (3==0)           if (2==0)   |   | if (3==0)   |
|  return 1;           return 1;  |->-|  return 1;  |
| else                else        |   | else        |
|- return 3*fac(2);  > return 2*1;|   |> return 3*2;|
| }                   }           |   | }           |
-----------------------------------   ---------------
```

15.4 Backtracking – Undo

In case of solving a problem where at each point in finding the solution different alternatives can be explored, backtracking can be applied. Backtracking amounts to pursuing one alternative at each choice point, and, if an attempt does not lead to the desired result, canceling that attempt and going back to the choice point asbove it, until a solution is found or all alternatives are exhausted.

Backtracking can be applied with parameters, as below.

```
 1 /** Nimm.java – recursive solution of the game of Nimm
 2  * Rules: two players remove 1 or 2 chips from the heap, taking turns;
 3  * whoever takes the last chip loses.
 4  * @author Kees Huizing
 5  * @date 31 October 2008
 6  */
 7
 8 public class NimmParameter {
 9     java.util.Scanner scanner = new java.util.Scanner( System.in );
10     int heap; //starting number of chips; always > 0
11
12     /** calculates winning move, if it exists, recursively
13      * @return number of chips to be taken, or 0 if there is no winning move
14      */
15     int winningMove(int h) {
16         int move = 0;
17
18         if (h == 1) { //already lost
19             return 0; //indicates there is no winning move
```

```
20          } else { // heap > 1
21              // try all moves
22              move = 0;
23              for (int m=1; m<3; m++) {
24                  if (h > m) { // else m illegal
25                          if ( winningMove(h−m) == 0 ) {
26                          // no win for opponent: good move!
27                          move = m;
28                          }
29                  }
30              }
31              return move;
32          }
33      }
34
35      void yourMove() {
36          int m;
37
38          System.out.println("What's your move? ");
39          m = scanner.nextInt();
40          while (m>2 || m>heap || m<1) {
41              System.out.println("What's your move? ");
42              m = scanner.nextInt();
43              if (m>2 || m>heap || m<1) {
44                  System.out.println("Illegal move");
45              }
46          }
47          heap = heap − m;
48      }
49
50      void play() {
51          System.out.println("what's the heap? ");
52          heap = scanner.nextInt();
53          while (heap < 1) {
54              System.out.println("what's the heap? ");
55              heap = scanner.nextInt();
56          }
57
58          while (heap > 0) {
59              System.out.println("Heap is "+heap);
60              if (heap == 1) {
61                  System.out.println("I take 1");
62                  heap = heap − 1;
63                  System.out.println("I lose");
64              } else {
65                  int m = winningMove(heap);
66                  if (m == 0) {
67                      // no win, make a random move
68                      m = (int)(Math.random() ∗ 2) + 1;
69                      System.out.println("I take "+m+" (but you can win)");
70                      heap = heap − m;
71                  } else {
72                      System.out.println("I take "+m);
73                      heap = heap − m;
74                  }
75                  System.out.println("Heap is "+heap);
76
77                  yourMove();
```

```
78                  if (heap == 0) {
79                      System.out.println("You lose.");
80                  }
81              }
82          }
83      }
84
85      public static void main(String[] a) {
86          new NimmParameter().play();
87      }
88  }
```

Backtracking can also be combined with the undo approach, as below.

```
1  /** Nimm.java – recursive solution of the game of Nimm
2   * Rules: two players remove 1 or 2 chips from the heap, taking turns;
3   * whoever takes the last chip loses.
4   * @author Kees Huizing
5   * @date 31 October 2008
6   */
7
8  public class NimmUndo {
9      java.util.Scanner scanner = new java.util.Scanner( System.in );
10     int heap; // number of chips; always > 0
11
12     /** calculates winning move, if it exists, recursively
13      * @return number of chips to be taken, or 0 if there is no wiinig move
14      */
15     int winningMove() {
16         int move = 0;
17
18         if (heap == 1) { //already lost
19             return 0; //indicates there is no winning move
20         } else if (heap == 2) {
21             return 1; // obviously
22         } else if (heap == 3) {
23             return 2; // ditto
24         } else { // heap > 3
25             // try all moves
26             move = 0;
27             for (int m=1; m<3; m++) {
28                 heap -= m; // take m
29                 if ( winningMove() == 0 ) {
30                     // no win for opponent: good move!
31                     move = m;
32                 }
33                 heap += m; // undo "take m"
34             }
35             return move;
36         }
37     }
38
39     void yourMove() {
40         int m;
41
42         System.out.println("What's your move? ");
43         m = scanner.nextInt();
```

```
44          while (m>2 || m>heap || m<1) {
45              System.out.println("What's your move? ");
46              m = scanner.nextInt();
47              if (m>2 || m>heap || m<1) {
48                  System.out.println("Illegal move");
49              }
50          }
51          heap -= m;
52      }
53
54      void play() {
55          System.out.println("what's the heap? ");
56          heap = scanner.nextInt();
57          while (heap < 1) {
58              System.out.println("what's the heap? ");
59              heap = scanner.nextInt();
60          }
61
62          while (heap > 0) {
63              System.out.println("Heap is "+heap);
64              if (heap == 1) {
65                  System.out.println("I take 1");
66                  heap -= 1;
67                  System.out.println("I lose");
68              } else {
69                  int m = winningMove();
70                  if (m == 0) {
71                      // no win, make a random move
72                      m = (int)(Math.random() * 2) + 1;
73                      System.out.println("I take "+m+" (but you can win)");
74                      heap -= m;
75                  } else {
76                      System.out.println("I take "+m);
77                      heap -= m;
78                  }
79                  System.out.println("Heap is "+heap);
80
81                  yourMove();
82                  if (heap == 0) {
83                      System.out.println("You lose.");
84                  }
85              }
86          }
87      }
88
89      public static void main(String[] a) {
90          new NimmUndo().play();
91      }
92  }
```

Chapter 16

Name overloading – Method overloading

Method names can be used to indicate similar methods if the methods differ in the parameter lists.

16.1 Aim

Sometimes methods perform essentially the same tasks, but only differ in the kind of information that is provided as input through their parameters. Such methods can be given the same names, using the difference in parameter lists to distinguish between them.

16.2 Means

We use *overloading* of method names, i.e., methods with the same names but with a different list of parameters are regarded as different methods. Note that a difference in the *names* of parameters only does not count as a difference in methods. Only a difference in the list of *types* is what counts, i.e.., a different number of parameters, a different type of a certain parameter, or a different ordering of types.

In the example below, there are three methods with names that differ only in the parameter lists.

```
1  class AccountOverloader{
2      double balance;
3      int rate;
4
5      void initialize() {
6          balance = 0;
7          rate = 0;
8      }
9
10     void initialize(double b) {
11         balance = b;
12         rate = 0;
13     }
14
15     void initialize(double b, int r) {
16         balance = b;
17         rate = r;
```

```
18      }
19
20
21      void showData(){
22          System.out.println("Balance is " + balance);
23          System.out.println("Interest rate is " + rate);
24      }
25
26      void demo(){
27          initialize();
28          showData();
29
30          initialize(50.0);
31          showData();
32
33          initialize(50.0, 5);
34          showData();
35      }
36
37      // start-up code
38      public static void main(String[] args) {
39          new AccountOverloader().demo();
40      }
41  }
```

The three calls lead to the execution of three different methods.

16.3 Execution model

In the execution model, the method that is executed as a result of a call is made visible in the object. A different actual parameter list will lead to a different method being executed and hence a different method body will be drawn in the object.

16.4 Remark

– Overloading and narrower or wider types

As discussed in chapter **??**, types may be in a relation, e.g., `int` being a narrower type than the wider type `long`. E.g., values of a narrower type can be assigned to variables of a wider type. Similarly, a value of a narrower type can be used in method calls where the formal parameter is of a wider type.

For the selection of an overloaded method, the "best match" is then chosen, i.e., if there is only a method `m(T t)` and a call `m(v)`is made with with a variable `v` of narrower type `S` of `T`, this method will be executed. However, if there is also a method `m(S s)` available, then this will be the method that is executed at the call `m(v)`.

Subtle mixtures of types could occur, e.g., in case of methods with more than one variable. This may lead to calls where it is ambiguous which method to apply. As an example, consider again the type `T` with narrower type `S`. Let `m(T t, S s)` and `m(S s, T t)` be overloaded methods. Let `v` and `w` be variables of type `T`. In the case of a call `m(v, w)` both methods match, neither of them more precise than the other. This is called ambiguous invocation and results in a compilation error. In general, it is not wise to define overloaded methods that differ only in subtle ways like this.

Chapter 17

Initialization – Constructors

Up till now, initialization of instance variables was often done using a method, say, `initialize`, and calling this method before most of the other methods. Java incorporates this idea in the language, thus providing a convenient way to initialize objects and to guarantee that this code is executed before any other method.

17.1 Aim

We want to initialize the instance variables of an object before any other code is executed on the object.

17.2 Means

We use a *constructor* to initialize objects. A constructor is a special method with as name the name of the class that it belongs to. A constructor is automatically called, directly when an object is created. By convention, the definitions of the constructors are positioned before the other methods in the class, just after the declaration of the instance variables.

A class can have several constructors. Overloading is used to distinguish between these, i.e., the constructors differ in the parameters they have. When an object is created, values for parameters (the arguments) are given in the new-clause, between the parentheses following the class name. The types of these arguments, and possibly their order, determine which constructor will be executed.

Constructors are different from ordinary methods in the following way:

1. a constructor has no return type (not even *void*);
2. the name of a constructor is the same as the name of the class; an ordinary method can *not* have the name of the class as its name;
3. a constructor can not be called directly; it is automatically called at creation of an object (by the `new` statement), and only then.

In the example below the constructor is demonstrated.

```
1 // demonstrator of constructors
2 class AccountC {
3     double balance;
4     int rate;
5
```

```
 6     // constructor 1
 7     AccountC(double b) {
 8         balance = b;
 9         rate = 0;
10     }
11
12     // constructor 2
13     AccountC(double b, int r) {
14         balance = b;
15         rate = r;
16     }
17
18     void showData(){
19         System.out.println("Balance is " + balance);
20         System.out.println("Interest rate is " + rate);
21     }
22
23
24     // start-up code
25     public static void main(String[] args) {
26         // create AccountC object using constructor 1
27         // and call showData on it
28         new AccountC(50.0).showData();
29     }
30 }
```

Because one actual `double` parameter `50.0` is filled in, the constructor with that parameter is called.

Output is:

```
Balance is 50.0
Interest rate is 0
```

To call the other constructor, the startup code should be:

```
1     // start-up code
2     public static void main(String[] args) {
3         // creates Account object using constructor 2
4         new AccountC1(50.0, 5).showData();
5     }
```

Output is:

```
Balance is 50.0
Interest rate is 5
```

Remark When a class doesn't contain constructors, an object can only be created without parameters. (This is the way we did it until now.) Once one or more constructors are present in the class, this way of creating is not available anymore, unless a parameterless constructor is added to the class.

The idea behind this is that a programmer can decide whether or not a parameterless constructor should be available, while having the convenience that a class doesn't need to have constructors.

Part III

Object structures – Composition

General Aim: Decompose a data processing concern into sub-concerns. When writing code for one sub-concern, abstract from the others.

General Means: Write a class for each sub-concern and compose these to perform the data processing concern. When writing a class, abstract from the others, using method calls for the interaction and parameters and returns for data transfer.

Chapter 18

Control transfer between objects – Method calls between objects

So far data processing concerns were described in one class and performed by one object. However, a data processing concern often consists of several smaller, interacting sub-concerns. To improve code organization such sub-concerns and their relations are described in separate classes. The classes are instantiated into objects, so the activity in the computer is structured similarly. Two cases are treated: obtaining more objects by writing for each object a class that describes it, and also creating several objects from the same class. Communication between objects is by means of method calls.

18.1 More classes for more objects

We show how more classes and objects can be used to structure a concern into sub-concerns.

18.1.1 Aim

We want to organize code and execution by dividing a data processing concern into sub-concerns.

18.1.2 Means

For each sub-concern we describe in a separate class the data processing to be performed by the corresponding object. Setting-up the relations between objects and their interaction is also described in the classes.

First, we consider the decomposition structure. Every object has a unique number by which it can be addressed, called *address* hereafter. One class can use another class by describing that an object of this second class is created and by describing that it has the address of that object.

Note that structuring occurs at both the class and the object level: The class refers, in its code, to another class; the object has the address of another object. A *reference variable* stores the address, also called the *reference*, of an object of a certain class.

The `new` command, that up till now only occurred in the `main`, is now used at other places in the class code as well. The type notion is extended to include classes, and reference variables are provided that can hold the address of an object. In `Class1` a reference variable, say `object2`, is declared of type `Class2`, that can store the address of an object of type `Class2`. Reference variables can occur as instance variables, as local

variables, and as parameters. As instance variables they are declared at the top of the class description, together with the other instance variables.

In `Class1`, generally in its constructor, the statement`object1 = new Class2();` describes that `Class2` is instantiated and that the address of the created object is assigned to the variable `object1`.

As before, the first object is created by instantiating `Class1` through `main`. The other objects are then created and connected through the (automatic) execution of the constructor of the coordinating object.

Second, we consider the collaboration: an object can call methods of the objects it has references to. This is done through a command of the form $\langle$*reference variable*$\rangle$.$\langle$*method name*$(\ldots)\rangle$. The reference variable holds the address of the object on which the method should be called.

As before, the first (non-constructor) method of the first object is called from `main`. From this method, other methods, also of other objects, can then be called.

Abstraction is achieved by not referring in one object directly to variables in other objects, but only calling methods of such objects. In Java it is possible to refer directly to variables of other objects, but we will not do so. It is also possible to prohibit this to some extend by explicit scoping restrictions, which we generally will apply.

Thus, convention: method calls are the only means of communication between objects.

Note that an object can only call a method of another object if it has a reference to that object. Also note that when a call from a certain object is terminated, control returns to this object. So collaborating objects do so sequentially: several objects simultaneously exist and store data, but only one at the time is active. A call from one object to another is asymmetric. The calling object needs to have a reference to the other object; the other object, however, does not need, and in many cases does not have, a reference to the calling object. Lastly note, that control is passed along references (and passed back against references).

The following example is about a bank that has a safe from which money can be put into an account, which in turn can be spent. We organize the code and the execution by modeling the bank and the account as two separate objects.

```
1  class Account {
2    double balance;
3
4    Account() {
5      balance = 0;
6      System.out.println("Account has been created, contains " + balance);
7    }
8
9    void add100() {
10     balance = balance + 100;
11     System.out.println("100 has been added to account, contains " + balance);
12   }
13
14   void take10() {
15     balance = balance − 10;
16     System.out.println("10 has been taken from account, left is " + balance);
17   }
18 }
19
20
21 public class BankOne {
22    double safe;
23    Account account; //reference variable to store address
24
25    BankOne() {
26       safe = 0;
```

```
27          account = new Account(); //creates an account
28          System.out.println("Bank with 1 account has been created, stores " + safe);
29
30      }
31
32    void banking(){
33      safe = 1000;
34      System.out.println("Safe is filled with " + safe);
35
36      safe = safe - 100; //to mimick transfer to account, money is taken from safe
37      System.out.println("100 is taken from safe, safe now contains " + safe);
38      account.add100(); //reference to account used to call add100
39    }
40
41    // main instantiates and uses the object
42    public static void main(String[] args){
43      new BankOne().banking();
44    }
45 }
```

Output is:

```
Account has been created, contains 0.0
Bank with 1 account has been created, stores 0.0
safe is filled with 1000.0
100 is taken from safe, safe now contains 900.0
100 has been added to account, now contains 100.0
```

The class `BankOne` describes the declaration of a reference variable of type `Account` and the instantiation of an object of that type as well as the assignment of the address to the reference variable. The description of the class `Account` is given separately.

Executing `main` instantiates an object of the class `BankOne`. This object has an instance variable that can hold a reference to an object of type (i.e., class) `Account`. Then the constructor of the `BankOne` object creates an object of the type `Account` and stores its address in the instance variable `account`.

The `BankOne` object calls, from its method `banking`, methods of the `Account` object; it uses the address in the reference variable `account` to refer to this object.

Note that interaction between objects consists solely of control being passed between methods of these objects; as yet no data is transferred between objects.

In the class `Account` there is no longer one special method where the processing starts. This is because from the viewpoint of an object that uses another object, there are usually several equally important methods that process data; in this case the `add100` and the `take10` method.

18.1.3 Execution model

Every object is labeled with a unique number, prefixed with a #-character to indicate that it is an address. This number is not chosen by the programmer but by the system and has no meaning, except that it is different for different objects. One can not expect that in an actual execution in the computer these exact numbers are used. When a reference variable holds a reference to an object, the address of this object is put in the box of the variable.

Instead of just one object, several may be drawn in an execution state. Passing control between objects behaves as expected: a passive control marker is placed in the calling method and a new method box is drawn in the called object, where the active marker is placed.

18.2 More objects from one class

We show how more objects from one class can be used to structure a concern into collaborating smaller concerns.

18.2.1 Aim

To have a structure of classes and collaborating objects. Some of the objects may be of the same class.

18.2.2 Means

To obtain several objects of the same class, the `new`-command for that class has to be executed several times. To distinguish between these objects, different reference variables need to be used.

As an example, two objects of the class `Account` are created and the addresses of these objects are stored in two reference variables.

In case multiple objects are instantiated from the same class, in the coordinating object different variables refer to each object. We added a variable `name` to put the name as information in the object itself, for example to show it on screen, to distinguish between objects of the same class.

```
1  class Account {
2    String name;
3    double balance;
4
5    Account(String n) {
6           name = n;
7           balance = 0;
8      System.out.println("Account " + name + " has been created, contains " + balance);
9    }
10
11   void add100() {
12     balance = balance + 100;
13     System.out.println("100 has been added to account " + name + ", now contains " + balance);
14    }
15
16   void take10() {
17     balance = balance − 10;
18     System.out.println("10 has been taken from account " + name + ", left is " + balance);
19    }
20 }
21
22
23 class BankMore {
24    double safe;
25    Account account1; //reference variable to store address
26    Account account2; //reference variable to store address
27
28
29    BankMore() {
30        account1 = new Account("FirstAccount"); //creates an account
31        account2 = new Account("SecondAccount"); //creates an account
32    }
33
34   void banking(){
```

```
35      safe = 1000;
36      System.out.println("Safe is filled with " + safe);
37
38      safe = safe − 100; //to mimick transfer to account1, money is taken from safe
39      System.out.println("100 is taken from safe, safe now contains " + safe);
40      account1.add100(); //reference to account1 used to call add100
41
42      safe = safe − 100; //to mimick transfer to account2, money is taken from safe
43      System.out.println("100 is taken from safe, safe now contains " + safe);
44      account2.add100(); //reference to account2 used to call add100
45    }
46
47    // main instantiates and uses the object
48    public static void main(String[] args){
49      new BankMore().banking();
50    }
51 }
```

Output is:

```
Account FirstAccount has been created, contains 0.0
Account SecondAccount has been created, contains 0.0
Safe is filled with 1000.0
100 is taken from safe, safe now contains 900.0
100 has been added to account FirstAccount, now contains 100.0
100 is taken from safe, safe now contains 800.0
100 has been added to account SecondAccount, now contains 100.0
```

18.2.3 Execution model

Nothing really new here!

Example to be provided

Note, that the reference variables `account1` and `account2` are both in the object of class `BankMore` and that the variable `name` in the objects instantiated from class `Account` holds the name of each account.

Chapter 19

Objects as service providers – Method calls with data parameters and returns

Up till now, only control was passed between objects. This chapter considers also communicating data values. A centrally organized structure with data communication is shown, other structures are analogous.

19.1 Aim

We want to describe collaborating objects with data communication. To not compromise the idea of having a structure of relatively independent objects, the data communication is at an abstract level.

19.2 Means

We use methods with data parameters to communicate data values from the calling object to the called object. We use data returns to communicate data values from the called object to the calling object.

An important convention in object-oriented programming is to use method calls as the only means to communicate between objects. This means that instance variables of objects are not approached directly from other objects. The advantage of this convention is that another implementation (e.g., with a more efficient data representation, using different instance variables) of a class can be used in place of the original one without changing the code of other classes. These other classes, namely, do not refer to the instance variables directly and communicate only via calls to methods of the changed class. If the code of these methods is changed to deal with the new data representation, the changes remain within the class.

Note that using the parameter and return mechanisms the communication is two-way, but that the construction is not symmetric: only the calling object needs to have a reference to the called object, but not the other way around. Furthermore, the initiative for the communication lies with the calling object.

An important design principle for object oriented programs is to have no more dependencies, in the form of references, than necessary.

Example

```
1 class Account {
2   double balance;
3
4   Account() {
```

```
 5      balance = 0;
 6    }
 7
 8    double getBalance(){
 9     return balance;
10    }
11
12    void add(int a) {
13      balance = balance + a;
14     }
15  }
16
17
18  public class BankDataParameterReturn {
19     Account account; //reference variable to store address
20
21     BankDataParameterReturn() {
22         account = new Account(); //creates an account
23     }
24
25    void banking(){
26      System.out.println("Account at start contains " + account.getBalance());
27
28      account.add(100); //reference to account used to call add
29
30      System.out.println("Account after adding 100 contains " + account.getBalance());
31     }
32
33     // main instantiates and uses the object
34     public static void main(String[] args){
35       new BankDataParameterReturn().banking();
36     }
37  }
```

Output is:

```
Account at start contains 0.0
Account after adding contains 100.0
```

From the program it can be seen how information goes into the object through the parameters and goes out from the object through the return.

19.3 Execution model

Again, nothing new. Data is passed via parameters and return values as described in the chapters on methods.

Chapter 20

Building object structures – Passing references

Previously, only data values were passed between objects. This chapter considers also communicating reference values. There is no difference in mechanism between the case of data or reference values. A centrally organized structure with reference communication is shown, other structures are analogous.

20.1 Aim

We want to describe collaborating objects with data communication. To not compromise the idea of having a structure of relatively independent objects, the data communication is at an abstract level.

20.2 Means

We use methods with reference parameters to communicate reference values from the calling object to the called object. We use reference returns to communicate references from the called object to the calling object.

Thus we respect the important OO convention to use method calls as the only means to communicate values of (reference) data between objects. This means that reference variables of objects can not be approached directly from other objects, hence the use of methods.

A reference variable holds the address of an object, which can also be viewed as a pointer. Changing the value of a reference variable therefore can be viewed as changing the address value, or, equivalently, as re-directing the pointer in the variable to point to the object with the new address.

An example of passing information to an object, using parameters, is the following.

```
1 class Account {
2   double balance;
3
4   Account() {
5     balance = 0;
6   }
7
8   void add100() {
9     balance = balance + 100;
```

```
10    }
11
12    void show() {
13      System.out.println("Balance is " + balance);
14    }
15  }
16
17  class Client {
18   Account account;
19
20   Client() {
21          //account = null; //makes null reference
22   }
23
24   void add100() {
25     System.out.println("Client adds 100 to account.");
26     account.add100();
27   }
28
29   void setAccount(Account a) { //enable others to set reference to account client
30          account = a;
31   }
32  }
33
34  public class BankReferenceParameters {
35     Client client;
36     Account account;
37
38     BankReferenceParameters() {
39          account = new Account(); //creates and makes reference to account
40          client = new Client(); //creates and makes reference to client
41     }
42
43    void banking(){
44
45     System.out.println("Bank adds 100 to account");
46     account.add100();
47
48     System.out.println("Bank sets clients reference to banks account.");
49     client.setAccount(account);
50
51      client.add100();
52      account.show();
53     }
54
55     // main instantiates and uses the object
56     public static void main(String[] args){
57       new BankReferenceParameters().banking();
58     }
59  }
```

Output is:

```
Bank adds 100 to account
Bank sets clients reference to banks account.
Client adds 100 to account.
Balance is 200.0
```

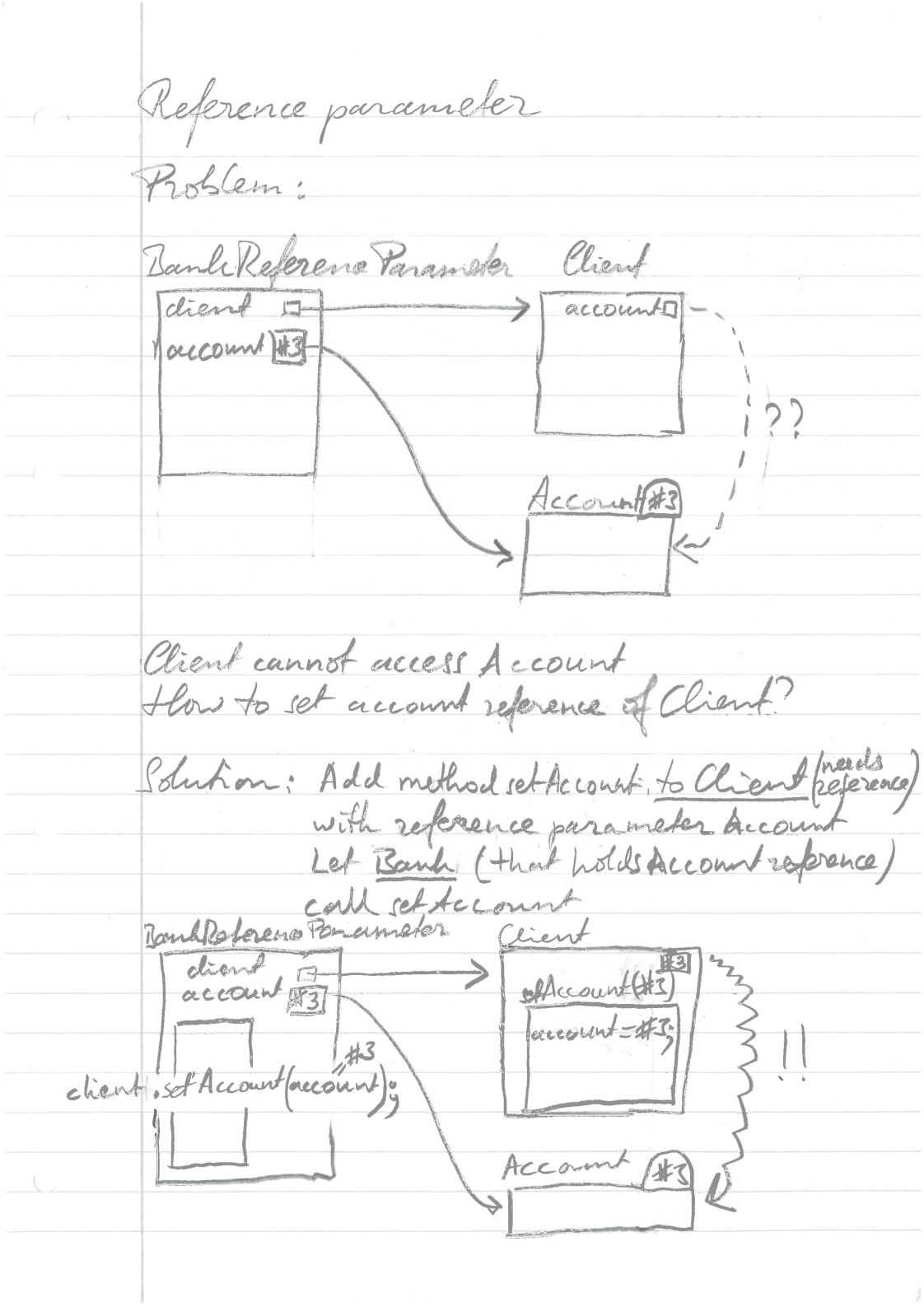
From the program it can be seen how information, here reference information, goes into the object through the parameter.

Figure 30.1 pictures this in a diagram.

An example of obtaining information from an object, using the return, is the following.

```
1  class Account {
2    double balance;
3
4    Account() {
5      balance = 0;
6    }
7
8    void add100() {
9      balance = balance + 100;
10   }
11
12   void show() {
13    System.out.println("Balance is " + balance);
14   }
15 }
16
17 class Client {
18  Account account;
19
20  Client() {
21        account = new Account(); //creates and makes reference to private account
22  }
23
24  void add100() {
25    System.out.println("Client adds 100 to account.");
26    account.add100();
27  }
28
29  Account getAccount() { //provides reference to account client to others
30        return account;
31  }
32 }
33
34 public class BankReferenceReturn {
35    Client client;
36    Account account;
37
38    BankReferenceReturn() {
39        account = null; // makes null reference
40        client = new Client(); //creates and makes reference to client
41    }
42
43   void banking(){
44
45    client.add100();
46
47    System.out.println("Bank gets clients reference to clients account.");
48    account = client.getAccount();
49
50    System.out.println("Bank adds 100 to account");
51     account.add100();
52     account.show();
```

Figure 20.1: BankReferenceParameters



```
53    }
54
55    // main instantiates and uses the object
56    public static void main(String[] args){
57      new BankReferenceReturn().banking();
58    }
59 }
```

Output is:

```
Client adds 100 to account.
Bank gets clients reference to clients account.
Bank adds 100 to account
Balance is 200.0
```

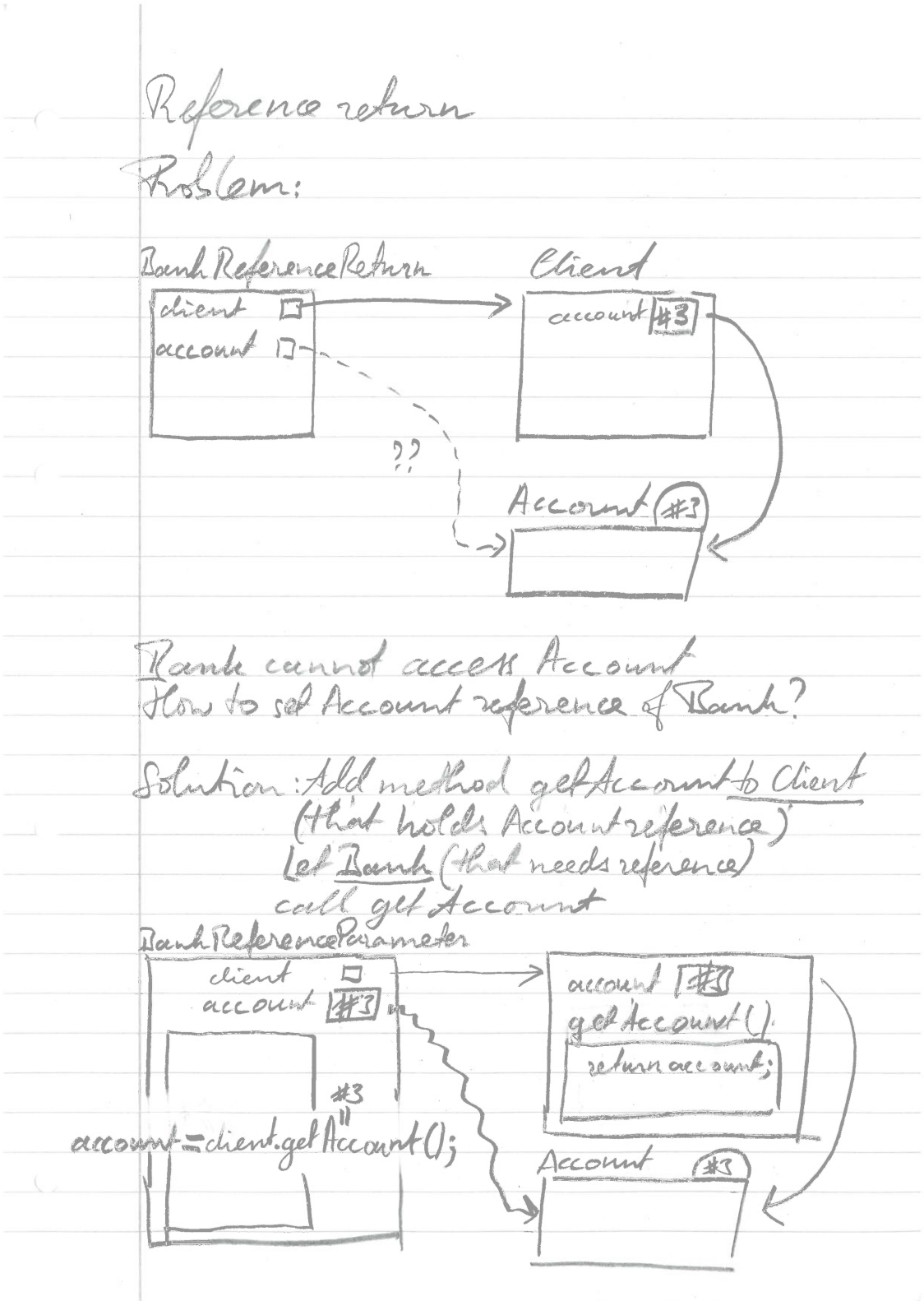
From the program it can be seen how information, here reference information, is obtained from the object through the return.

Figure 20.2 pictures this in a diagram.

20.3 Execution model

Nothing new, models for both examples follow directly. The address value of the account is passed as a parameter, respectively returned, in the same way as any other value.

Figure 20.2: BankReferenceReturn



Chapter 21

Exception handling – Exception mechanism

Errors may occur beyond the influence of the programmer. For example, when the program tries to read from a disk file that has been deleted. A good (*robust*) program should take all these situations into account. This will result in a lot of extra code that tends to clutter the rest of the code.

In some situations, there is no right value to return for a function (i.e., a method that has a return type). For example, there is no good value to return as a result of division by zero.

21.1 Aim

1. Separate code to handle exceptional situations from the rest of the code.
2. Have a possibility to signal that a method did not terminate normally and the return value should be ignored.

21.2 Means

The mechanism of exception handling realizes these aims.

ad 1 Code in which an exception may occur is surrounded by a *try-block* and the code that handles the exception is written after this block in one or more *catch-blocks*. When during execution of the try-block an exception occurs, the execution skips the rest of the code in the try-block and jumps to the catch-block. The code that handles the normal situation is in the try-block, nicely separated from the code for the exceptional situations in the catch-block(s).

ad 2 A method declaration can be extended with a *throws-clause* that signals that an exception may occur during execution of the method. When this exception occurs, the execution of the method is aborted, no return value is provided and an exception occurs in the calling code. There execution continues as described under 1. We say that the exception is *passed* to the calling code.

21.3 Example of catching an exception

Opening a file for reading may cause, or *throw* in Java parlance, an exception of the type FileNotFoundException. In the code below, we show how to handle such an exception.

```
1  // ExceptionExample.java – handling an exception while reading from a file
2  import java.util.Scanner;
3  import java.io.*;
4
5  public class ExceptionExample {
6      Scanner scanner;
7      File file;
8      String filename = "blah.txt";
9
10     public void demo() {
11         file = new File( filename );
12
13         try {
14             scanner = new Scanner( file );
15             String word = null;
16             int n = 0; // number of words in file
17             while ( scanner.hasNext() ) {
18                 word = scanner.next();
19                 n = n+1;
20             }
21             System.out.println("The file contains "+n+" words.");
22             scanner.close();
23         } catch( FileNotFoundException e) {
24             System.out.println("The file "+filename+" could not be found :-(");
25         }
26     }
27 }
```

If the file `blah.txt` doesn't exist, an exception will be thrown during execution of line 14. In that case, control leaves the try block immediately and moves to the catch block. So all code from line 15 to 22 is skipped and the line 24 is executed. When the file is available for reading, however, execution will continue with line 15 and line 24 will not be executed.

it is important that the try-block is left immediately when an exception occurs. Otherwise, code such as `scanner.hasNext` and `scanner.next()` will be executed. This code is meaningless when the file doesn't exist and could lead to more errors and havoc.

21.4 Example of passing an exception

Instead of handling an exception in a try/catch construction, it may be propagated to the calling method. The method has to signal (declare) this in the header.

```
1  // ExceptionExample.java – handling an exception while reading from a file
2  import java.util.Scanner;
3  import java.io.*;
4
5  public class ExceptionExample2 {
6      String filename = "blah.txt";
7
8      int countWordsInFile(String filename) throws FileNotFoundException{ // possible trow
9          Scanner scanner;
10         File file;
11
12         file = new File( filename );
13         scanner = new Scanner( file ); //if blah.txt does not exist, an exception is thrown
```

```
14          String word = null;
15          int n = 0; // number of words in file
16          while ( scanner.hasNext() ) {
17              word = scanner.next();
18              n = n+1;
19          }
20          scanner.close();
21          return n;
22      }
23
24      public void demo2() {
25          int n;
26          try {
27              n = countWordsInFile( filename );
28              System.out.println("The file contains "+n+" words.");
29          } catch( FileNotFoundException e) {
30              System.out.println("The file "+filename+" could not be found :-(");
31          }
32      }
33
34      public static void main(String[] args) {
35         new ExceptionExample().demo();
36      }
37  }
```

Exception propagation rules:

1. An exception that is not caught inside a method is passed on to the place where the method was called. No return value will be provided.

2. A method that doesn't handle a certain type of exceptions itself (by means of a try and catch construction) should declare that it may propagate (throw) this type of exception in the header of the method, as follows:

   ```
   int countWordsInFile(String filename) throws FileNotFoundException
   ```

   The compiler will check that all exceptions that could occur in the method body and are not caught by try/catch-blocks are declared.

21.5 Unchecked exceptions

There is a special kind of exceptions that are not specific to certain methods and are usually related to programming errors, not to external errors. These do not have to be caught or passed on, because they can occur at so many places. The compiler does not check whether they are declared in the headers and are therefore called *unchecked exceptions*. They are all subclasses of the class RuntimeException (see below in the section on Exception hierarchy). You may have encountered these already. Examples are NullPointerException, occurring when a method is called on a reference that is `null`, and ArrayIndexOutOfBoundsException, occurring when an array is accessed with an index that doesn't exist. You could catch them if you want, but it is not recommended. Since they are usually caused by a programming error, you should not hide them, but prevent them.

21.6 Exception hierarchy

For each type of exception, there is a corresponding class with the same name. When an exception occurs, an object of the corresponding class is created. This object is passed as a kind of parameter to the catch-block. It contains some information that can be used there.

Exception classes have an inheritance hierarchy. E.g., all exceptions that are a subclass of the class *RuntimeException* don't have to be declared. Exceptions that have to do with input and output to files, such as FileNotFoundException, are subclasses of *IOException*. This makes it possible to catch several types of exception in one catch-clause. For example, instead of

```
try {
    ...
} catch( FileNotFoundException e) {
    ...
} catch( EOFException e) {
    ...
} catch( InterruptedIOException e) {
    ...
}
```

one can also write

```
try {
    ...
} catch( IOException e) {
    ...
}
```

The catch-blocks are searched in the order they appear. The first block that matches the exception (i.e., its type is the same as the exception thrown or a supertype) will be executed.

21.7 Exceptions and the API

The API, Applications Programmer's Interface, describes the available classes and their methods. When a method could throw an exception, this is mentioned in the API. This way, you can see which exceptions you can expect when you use a method. For example, the description in the API[1] of the constructor of the class *Scanner* looks as follows:

Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:
`source` - A file to be scanned
Throws:
`FileNotFoundException` - if source is not found

---

[1] http://java.sun.com/javase/6/docs/api/

21.8 Throwing exceptions and creating exception classes

Although in most programs exceptions occur as a result of a statement or a method call, there could be situations when you as a programmer need to have explicitly generate an exception. For this, the keyword `throw` is provided. So if you want to create a FileNotFoundException, you could write

```
throw new FileNotFoundException("Just for fun");
```

Probably, it makes more sense to use an exception that you have defined in your code, as follows.

```
class AdultContentException extends Exception {
        public AdultContentException(String info) {
                super(info);
        }
}

class ExceptionCreationExample {
        void protectedRead( String filename ) throws AdultContentException, FileNotFoundException {
                Scanner scanner;
                File file;

                if (filename.equals("sex.txt") {
                        throw new AdultContentException("Filename suggests not suited for minors");
                } else {
                        scanner = new Scanner( new File( filename ) );
                        while ( scanner.hasNext() )
                                ...
                }
        }

        void demo() {
                String filename = ...
                try {
                        protectedRead( filename );
                } catch (IOException e) {
                        System.err.println( "An I/O-problem occurred while accessing file "+ filename
                              +"; further infor: "+e);
                } catch (AdultContentException e) {
                      System.err.println( "You can't read "+filename+" because "+e);
                }
        }
}
```

Chapter 22

File output and input – java.io

Up till now, data was stored in program variables and hence not preserved if a program terminated. Now more permanent storage is introduced.

22.1 Aim

We want to store data more permanently than just for the execution of a program. For example, the results computed by one program might be stored for later use with another program that runs when the first program is not active anymore.

22.2 Means

Data is stored in and retrieved from files, for example on the hard disk of the computer, like we treat editor files. The difference is that it is now the running program that organizes the storage and retrieval.

We consider output by a Java-program to a file and input by a Java-program from a file: the perspective is always from the viewpoint of the program. A complication is that errors can occur, in which case Java throws a so-called exception that has to be dealt with in your program. (See chapter 21 for more information about exceptions.)

We limit ourselves to text files as these can be accessed in a way that is already known: a Scanner for input and println and print for output. Furthermore, text files can be inspected and changed using an editor. Another, not essentially different, option are the more efficient binary files. For a treatment of these we refer to textbooks and the Java documentation.

We now describe how the objects that perform output and input are obtained and how they are used.

22.2.1 Output to file

An object of the class File contains the name and location (its *pathname*) on disk. File object: Declare, create File object (the file itself is not created!) and set the disk file name.

```
File f = new File("df.txt");
```

Write object: Declare and create PrintWriter object and link it to a file for writing:

```
PrintWriter pw = new PrintWriter( f );
```

Use of such a PrintWriter object is very similar to using System.out (which we use for writing to the console).

Write a String s and go to a new line:

```
pw.println(s);
```

Write a String s without going to a new line:

```
pw.print(s);
```

When you don't want to use the file for output anymore, you *close* the file:

```
pw.close();
```

Example

```
1  //Writing to a file
2  import java.util.Scanner;
3  import java.io.*;
4
5  public class FileWrite {
6      Scanner sc;
7      String filename;
8      File f;
9      PrintWriter pw;
10
11     public void demo() {
12         sc = new Scanner( System.in );
13         String choice;
14         String word;
15
16         filename = "df.txt";
17         f = new File( filename );
18
19         try {
20             pw = new PrintWriter( f ); //(if not exists makes,) empties and opens file,
21                                        //directs pw where to write to
22             System.out.println("Do you want to add a word to the file? (y/n)");
23             choice = sc.next();
24             while ( choice.equals("y") ) {
25                 word = sc.next();
26                 pw.println(word); //writes word into file
27                 System.out.println("Do you want to add a word to the file (y/n)?");
28                 choice = sc.next();
29             }
30             pw.close();
31         } catch( FileNotFoundException e) {
32             System.out.println("The file "+filename+" could not be opened for writing");
33         }
34     }
35
36
37     public static void main(String[] args) {
38         new FileWrite().demo();
39     }
40
41 }
```

In fact, there is a little more to opening a file for writing (by creating a PrintWriter object) than we explained thus far. The statement,

```
PrintWriter pw = new PrintWriter( f );
```

creates a new file if `df.txt` did not exist yet or, if it did exist, empties it. It will throw a `FileNotFoundException` if an error occurs, e.g., when the file is not writable.

Location of files

When you create a file by given a name, it is put in the current working directory. Where this is, depends on your Java system. If you want to read or write a file in a different directory, you can:

- use an absolute pahtname, for example `new File("C:
  Documents and Settings
  df.txt")` (Windows) or `new File("/Users/kees/df.txt")` (Linux, Mac);
- use a relative pathname: `..` is the parent directory of the current directory;
- change the current working directory by tweaking your IDE;
- use a JFileChooser object (see below).

Appending

If you don't want a file to be emptied before writing to it, but want to append text to the end of the file, you need to change the code as follows (assume f is an appropriate File object).

```
FileWriter fw = new FileWriter( f, true );
PrintWriter pw = new PrintWriter( fw );
```

You can give FileWriter as a second argument a boolean. In case the boolean is true, the existing file is not emptied and text is appended to the original file (i.e., added at the end). In case the boolean is absent or false, the file will be emptied before the PrintWriter starts writing to it. The FileWriter object is an intermediate object that is automatically created when a PrintWriter object is created from a File object. Here, we create it explicitly to be able to set the append option, since this option is not available in the constructors of PrintWriter.

Storage location: Assume there is a file on disk (in the program directory) with the name

```
df.txt
```

File object: Declare, create and set the disk file name.

```
File f = new File("df.txt");
```

Read object: Declare and create Scanner object, link it to the File object and open the file for reading.

```
Scanner sc = new Scanner( f );
```

Reading from a Scanner object is known from reading from the console (keyboard). E.g., to read a word from the file, you program:

```
String s = sc.next();
```

When you read from a file, you may reach the end of the file. When the end of the file has been reached, the call `sc.hasNext()` will return `false`. You can use this, e.g., in the guard of a loop that reads through the file.

Example

```
1  //Reading from a file
2  import java.util.Scanner;
3  import java.io.*;
4
5  public class FileRead {
6      String filename;
7      File f;
8      Scanner sc;
9
10     public void demo() {
11         String word;
12
13         filename = "df.txt";
14         f = new File( filename );
15
16         try {
17             sc = new Scanner( f ); //(if not exists) makes and) opens file,
18                                    //directs sc where to read from
19             while ( sc.hasNext() ) {
20                 word = sc.next(); // reads word from file
21                 System.out.println(word);
22             }
23             sc.close();
24         } catch( FileNotFoundException e) {
25             System.out.println("The file "+filename+" could not be found");
26         }
27     }
28
29     public static void main(String[] args) {
30         new FileRead().demo();
31     }
32 }
```

NB It is not possible to simultaneously write to and read from the same file. Before changing from writing to reading or vice versa, you need to close the file and open it again by creating a new read or write object.

Chapter 23

Class members – Static members

To be changed to account examples

So far, data storage and manipulation have been done through objects, instantiated from classes. Some data and manipulation is at the level of the class rather than the objects instantiated from it. For this, the class object is used: it is an object that is created when the file that the class definition is in is executed, i.e., even before any instance object is created.

23.1 Aim

To have a place for data storage and manipulation that is at class level rather than at the level of individual objects.

23.2 Means

Apart from the description in the class of variables and methods that get instantiated when an object is instantiated from that class, `static` variables and methods can be described. Static variables and methods do not belong to any of the objects that are explicitly instantiated from the class, but to a separate class object that is implicitly, automatically, instantiated when a class is executed. The class object is not identified by a reference, but by the name of the class it belongs to. Static methods are called through <classname>.<methodname>. `main` is a special case of such a static method; special in the sense that it is called automatically when the class is executed. Also, this is the only way `main` can be called ("listen carefully, `main` only gets called once").

Example

We want to keep count of the number of animals instantiated. This is clearly not the task of any individual animal. Also, as the aim is to keep count of all animals created, it would not be correct to make this the task of the farmer, as, for example, there might be more farmers, each instantiating cows. It is typically a task at the level of the Animal class. We extend the animal class accordingly.

```
1 class Animal{
2   static int numberOfAnimals;
3
4   static {
5     numberOfAnimals = 0;
6   }
7
```

```
 8   static int getNumberOfAnimals(){
 9     return numberOfAnimals;
10   }
11
12   int stomach;
13
14   Animal(){
15     stomach = 0;
16           numberOfAnimals = numberOfAnimals + 1;
17   }
18
19   void eat(){
20     stomach=stomach+1;// 1 meal eaten
21     System.out.println("Stomach contains after meal " + stomach);
22   }
23 }
24
25
26 public class FarmerStatic{
27   Animal animal1;
28   Animal animal2;
29
30   void farm(){
31           System.out.println("Number of animals is "+ Animal.getNumberOfAnimals());
32
33     animal1 = new Animal();
34     System.out.println("Number of animals is "+ Animal.getNumberOfAnimals());
35
36     animal2 = new Animal();
37     System.out.println("Number of animals is "+ Animal.getNumberOfAnimals());
38   }
39
40   // main instantiates and uses the object
41   public static void main(String[] args){
42     FarmerStatic farmerStatic = new FarmerStatic();
43     farmerStatic.farm();
44   }
45 }
```

The following new lines occur.

`static int numberOfAnimals` keeps the counting information.

`static` is the constructor for the class object `Animal`. By rule it is named `static`.

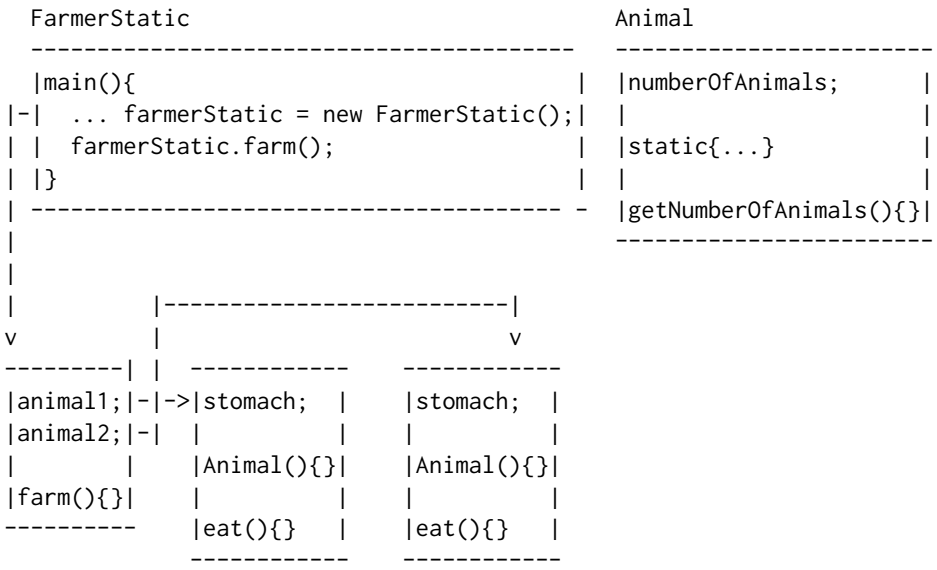
`numberOfAnimals = numberOfAnimals + 1;` in the constructor of `Animals` increases the animal count by 1 each time a new object is instantiated from `Animal`.

`getNumberOfAnimals` is the getter method that enables to get the value of the counter.

23.3 Execution model

Thus far, we only modeled the objects that were instantiated from a class. We extend the model with the class objects. Because class objects are not identified by a reference, we put the name of the class they belong to to them. Note, that the main is also a static method and therefore belongs in the class object: We sketch the model.

Example



NB The `farmerStatic` object has referencs to both animal objects `animal1` and `animal2`. Therefore, the farmerobject can call the methods of the animal objects. Static variables and methods of a class, belonging to the class object of that class, can be accessed and called without references being needed. For example, the constructor `Animal` that belongs to the instantion `animal1` can update the static variable `numberOfAnimals` in the class object for `Animal`. Also, the static method `getNumberOfAnimals` in the class object for `Animal` can be called from the method `farm` in the object `farmerStatic`.

Chapter 24

Encapsulation - Access modifiers

So far, we have only used conventions to limit dependencies between classes. One category of such conventions concerns accessibility of variables and methods. Now we introduce language constructs that enable to enforce this accessibility.

24.1 Aim

Decrease dependencies between classes that are in a composition relation

24.2 Means

Restrict what classes can see and use from other classes. Methods and variables can be hidden from other classes. This way, the other classes can not use the hidden methods and variables and hence do not depend on them.

24.2.1 Visibility modifiers

A visibility modifier is a special word that you write at the start of a declaration of methods, variables, and classes. Examples:

```
private double income;

private double calculateIncome() {
    ...
}
```

This variable and method are *private*. They can only be "seen" (i.e., for the method, called, for the variable, used) inside the class where they are declared.

```
public int score;

public void actionPerformed(ActionEvent e) {
    ...
}
```

This variable and method are *public*. They can be seen from any class in the system.

Between these two extremes there are two intermediate levels of visibility that have to do with grouping of classes into so-called *packages*.

Packages

A *package* is a group of classes that belong together and are labeled as such by the programmer. An example is `java.io`, a set of classes that deals with input and output and files. The class `Scanner` is part of the package `java.io`. You can declare that the classes in a file belong to package `myPackage` by putting the line

```
package myPackage;
```

at the beginning of the file *and* putting the classes together in a folder with this name.

When you leave out the line with `package`, the classes belong to the so-called *anonymous* package for which the name of the folder is irrelevant.

Overview of visibility modifiers from wide to narrow:

- `public`: no limitations, visible in all classes;
- *none* (sometimes called ”package”): visible only in the classes of the same package (this is what we have used mostly so far);
- `private`: visible only in the same class.

Part IV

Class structures – Inheritance

{**General Aim:** Describing a data processing concern using classes that are extensions or specializations of other classes,giving rise to execution in specialized objects.

General Means: Use `extends` or `implements` to add or change code of an existing class. Writing the extension or change in a compositional manner, adding to the existing code in an extension of the existing class, not copying or changing the code of it.

Chapter 25

Adding functionality – Adding methods and variables with extends

In this chapter, adding functionality to a class is treated. Code is re-used by creating a new class from an existing one, without the need to copy code. This is advantageous because duplicating and changing existing code is tedious, error-prone, and sometimes even impossible because the code is not accessible. This can also be viewed as grouping and naming shared code.

25.1 Aim

To write a new class that adds functionality of an existing one without the need to copy code. We want to extend data storage as well as data manipulation. The extension should result in an is-a relation between the new class and the existing one.

25.2 Means

inheritance, using `extends` to extend an existing class. We write the new class as an extension of an existing one as follows.

1. We acquire the functionality of the existing class for the new class by means of the keyword `extends`: the first line of the class definition is $\langle NewClass \rangle$ `extends` $\langle OldClass \rangle$. Thus, without copying code, all variables and methods of the old class are available in the new class.
2. The rest of the code of the new class provides the additional functionality: added variables and/or methods, for now with names different from the ones present.

Because the functionality of the present class carries over to the new class, this relationship is called inheritance. In the inheritance relationship, the new class is called a *subclass* of the old one, and the old one is called a *superclass* of the new one.

Example

As an example, we extend an animal to a man by adding a brain and a method that prints its size.

```
1 class Animal {
2     int stomach;
3
```

```
 4      Animal() {
 5          stomach = 2;
 6      }
 7
 8      void print() {
 9          System.out.println("Stomach content is " + stomach);
10      }
11  }
12
13  class Man extends Animal {
14      int brain;
15
16      Man() {
17          brain = 7; // extra variable
18      }
19
20      void printBrain() { // extra method to print man's brain info
21          System.out.println("Brain size is " + brain);
22      }
23  }
24
25  public class ExtensionAddDemo {
26      Animal animal;
27      Man man;
28
29      void demo(){
30          animal = new Animal();
31          man = new Man();
32
33          animal.print();
34
35          man.print();//"animal" info through animal's print(), still available for man
36          man.printBrain();//"man"'s info through, new, man's printBrain()
37
38          //animal.printBrain(); gives compiler error: animal has no method printBrain
39      }
40
41      // main instantiates and uses the object
42      public static void main(String[] args){
43          new ExtensionAddDemo().demo();
44      }
45  }
```

```
Stomach content is 1
Stomach content is 1
Brain size is 5
```

`Man` has as variables `stomach` (old, from `Animal`) and `brain` (new) and as functionality `print` (old, from `Animal`) and `printBrain` (new). On an object of type `Man`, say `man`, call `man.print()` can still be performed. On an object of type `Animal`, say `animal`, call `animal.printBrain()` cannot be performed: `animal` does not have that method; a compiler error will report this.

All variables from the superclass are available to methods in a subclass.

For example, in `Man` a method can be put that prints both the stomach and the brain information, like:

```
void printAll() {  // extra method to print all info
```

```
System.out.println("Stomach content is " + stomach);// superclass variable used in su
System.out.println("Brain size is " + brain);
```

The constructor in the subclass when executed first executes the corresponding constructor in the superclass. There are subtleties here that will be addressed when appropriate.

25.3 Execution model

Objects of the subclass are modeled as one object, according to the combined description in the super- and subclass. The calls to the printing methods of both the version with separate implementation of `Man`'s `printBrain()` and the version using `printAll` are shown.

Figure 25.1: ExtensionAddDemo

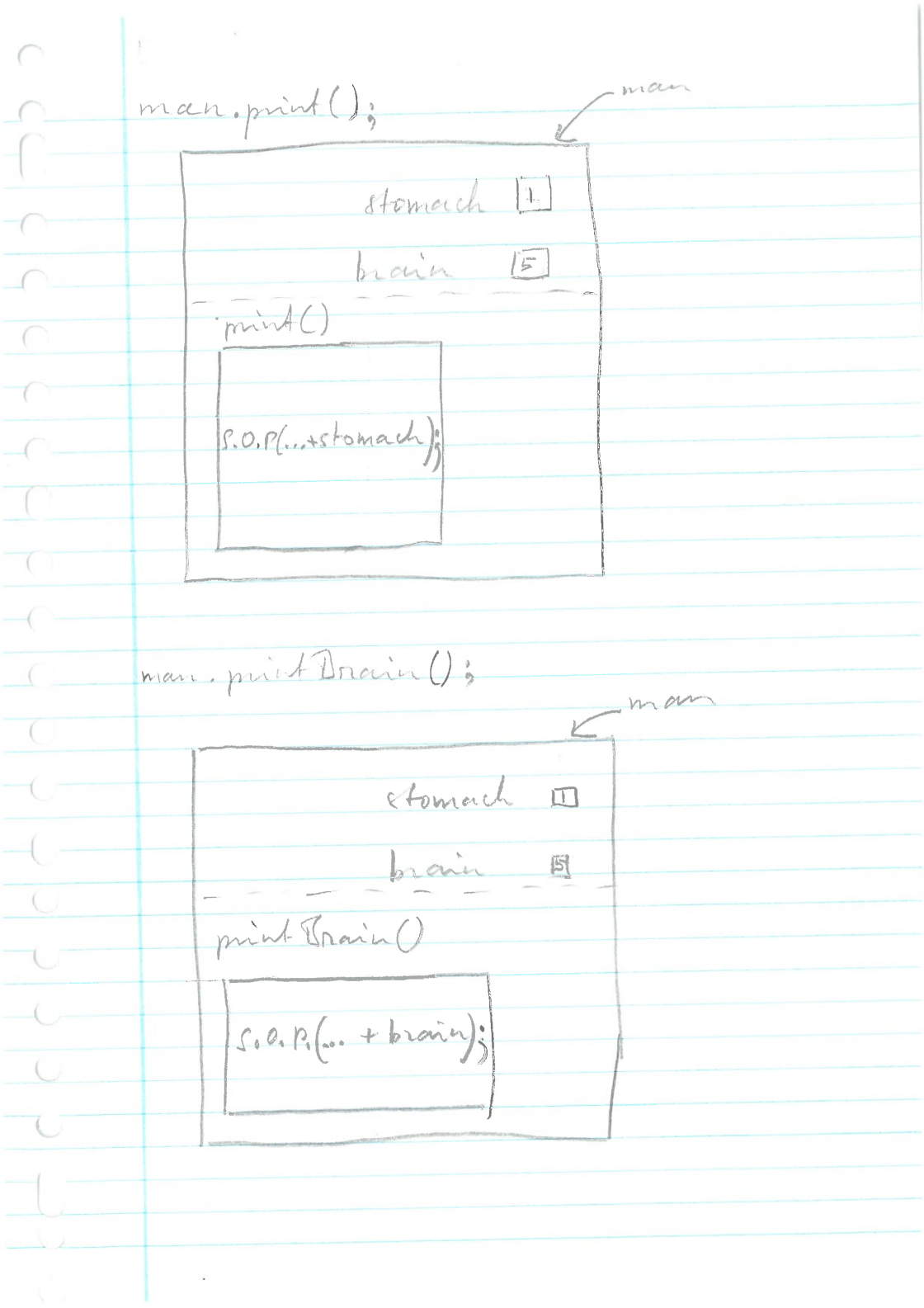
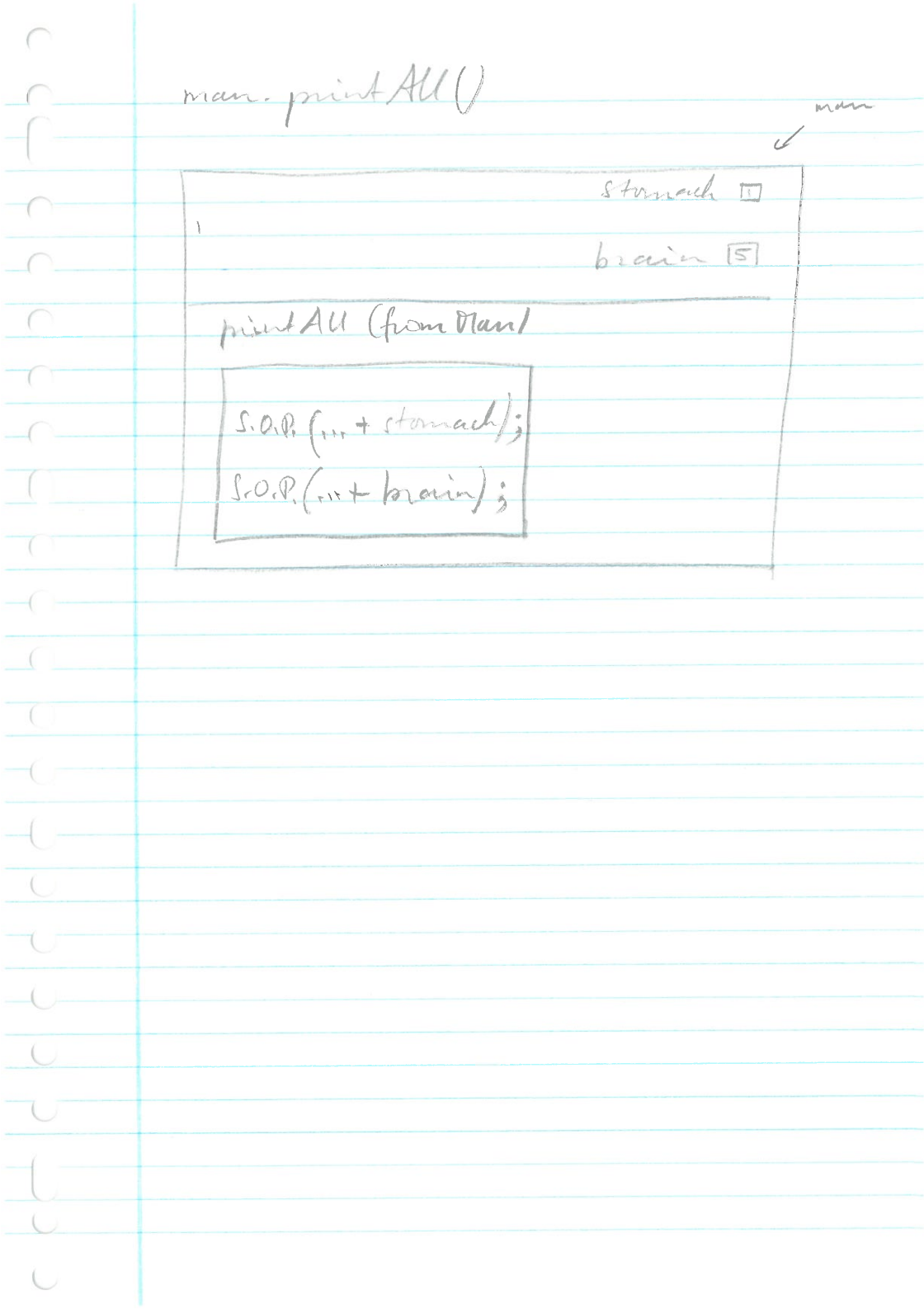


Figure 25.2: ExtensionAddAllDemo



Chapter 26

Super level code – subtyping, polymorphism, dynamic binding

Up till now, inheritance was used as a means to re-use classes and, more generally, to organize code with respect to an is-a relation. We now use the inheritance relation between classes as a subtype relation. This enables to write code that can use variables that can hold values of both a superclass and its subclasses.

26.1 Aim

To write code that treats super- and subclass objects as being the same in the is-a sense as regards their shared code.

26.2 Means

Subtyping with *polymorphism* and *dynamic binding*, enabling to write *superlevel* code.

Subtyping is the language feature that the subclass relation can be used as a subtype relation: a reference to a subclass can be stored in a variable of a superclass. The compiler accepts this. This is called polymorphism: such code is called polymorphic.

Dynamic binding is the language feature that if a method is called on an object of a subclass that does not have that method, than the method of the nearest superclass will be executed.

Example

In the following example, the method `demo` contains superlevel code, using a variable `Animal` of type `Animal` in which either a reference to an object of type `Man` or to an object of type `Man` is stored.

The output is shown for choices a (animal) and m (man).

```
1 import java.util.*;
2
3 class Animal {
4     int stomach;
5
6     Animal() {
7         stomach = 1;
8     }
```

```
 9
10      void print() {
11          System.out.println("Stomach content is " + stomach);
12      }
13  }
14
15  class Man extends Animal {
16      int brain;
17
18      Man() {
19          stomach = 2;// to show by printing content difference between animal and man
20          brain = 7;
21      }
22  }
23
24  public class SuperlevelCodeExtensionDemo {
25    Scanner sc = new Scanner(System.in);
26
27    void demo() {
28    Animal animal;
29    char choice;
30
31    System.out.println("Choose a(nimal) or m(an)");
32    choice = sc.next().charAt(0);
33
34      if (choice == 'a') {
35        animal = new Animal();
36      } else {
37        animal = new Man(); // value of subtype stored in variable of supertype
38      }
39
40      animal.print(); // superlevelcode: print is also used for objects of type Man
41    }
42
43    public static void main(String[] args){
44      new SuperlevelCodeExtensionDemo().demo();
45    }
46  }
```

```
Choose a(nimal) or m(an)
> m
Stomach content is 1

Choose a(nimal) or m(an)
> m
Stomach content is 2
```

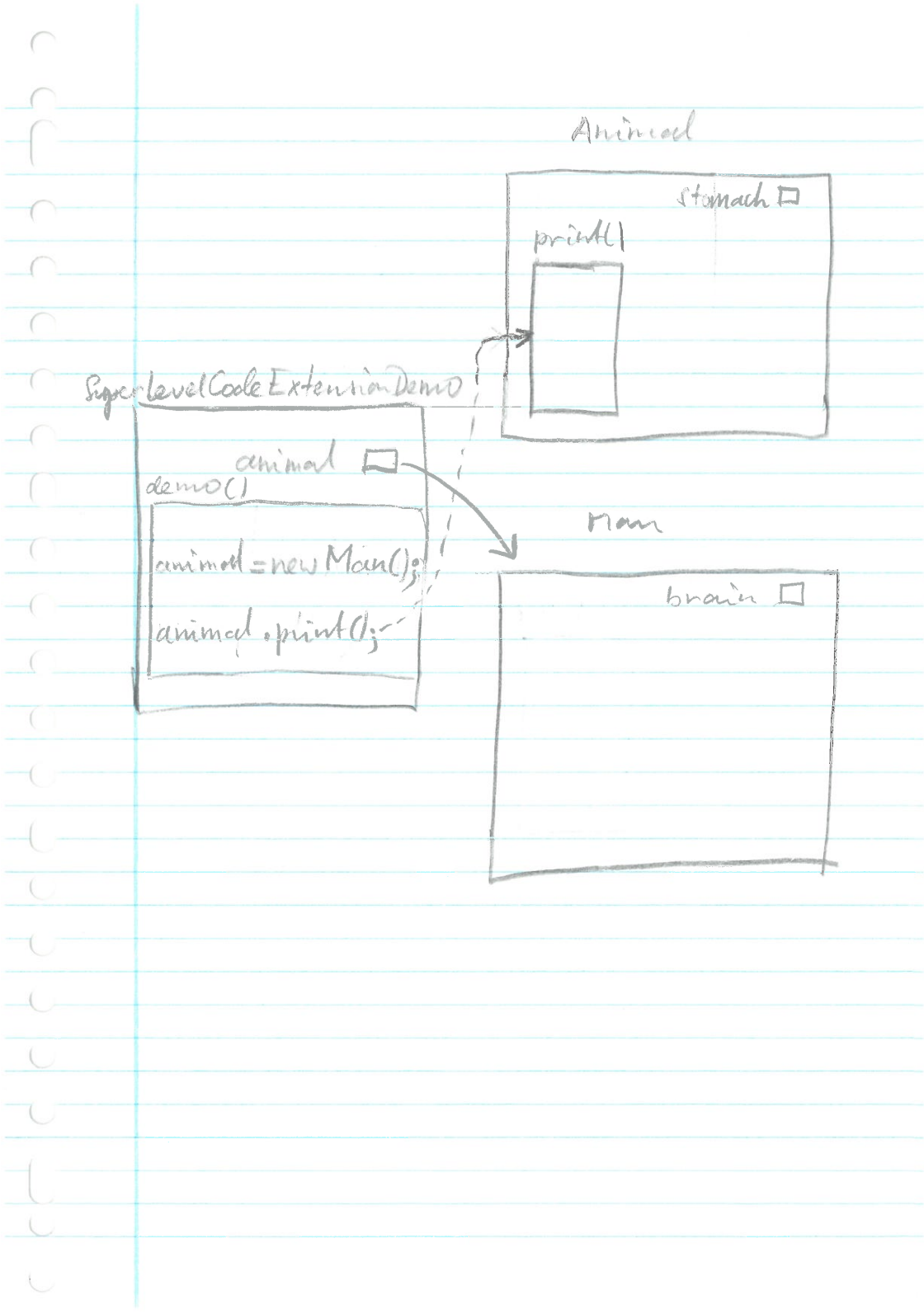
In the example the superlevel code is limited: just the call to `animal.print`.

A more useful application of subtyping is an array of supertype in which inIn the example the superlevel code is limited: just the call to `animal.print`.

A more useful application of subtyping is an array of supertype in which in which also objects of subtype(s) can be stored; superlevel code can than be used to pass through the array and call methods of the supertype. also objects of subtype(s) can be stored; superlevel code can than be used to pass through the array and call methods of the supertype.

This can be pictured as below.

Figure 26.1: SuperlevelCodeExtension



26.3 Execution model

As before.

Chapter 27

Changing functionality – Overriding methods with extends

In chapter 25, adding functionality to an existing class was considered, now changing the functionality is treated.

27.1 Aim

To write a new class that redefines part of the functionality of an existing one without the need to copy code. The extension should result in an is-a relation between the new class and the existing one.

27.2 Means

inheritance with *overriding* to change methods of an existing class. We write the new class as a change from an existing one as follows.

1. We acquire the functionality of the existing class for the new class by means of the keyword `extends`: the first line of the class definition is $\langle NewClass \rangle$ `extends` $\langle OldClass \rangle$. All variables and methods of the old class remain available as names in the new class, but the implementation may be changed.
2. The rest of the code of the new class provides changed functionality for a method. A method of the superclass that we want to change we declare again in the subclass, i.e., with the same name but re-implement it with a different body. This new method *overrides* the method of the superclass. If the method with this name is called on an object of the superclass, the old body is executed; if it is called on an object of the subclass, the new body is executed.

Because the functionality of the superclass still carries over to the subclass, although possibly in adapted form, this is also called inheritance. The keyword `extends` is used for both forms of inheritance; in the second case, extends may also mean a change in functionality rather than just an extension.

Both kinds of extension, adding and changing functionality, may be used together in creating a subclass/.

Redefining functionality is shown on the same example as used for adding functionality. The idea is, that when the subclass extends the superclass, the printing does not require an additional print method, but rather a change, an extension, of the functionality of the method itself: hence that method of the superclass is overridden in the subclass: in `Man` the method `print` of `Animal` is overridden, reflecting the extension of the data with brain.

```
1  class Animal {
2    int stomach;
3
4    Animal() {
5    stomach = 1;
6    }
7
8    void print() {
9      System.out.println("Stomach content is " + stomach);
10   }
11 }
12
13 class Man extends Animal {
14   int brain;
15
16   Man() {
17     brain = 5;
18   }
19
20   @Override // overridden method
21   void print() { // same name as superclass method
22     System.out.println("Stomach content is " + stomach);
23     System.out.println("Brain size is " + brain);
24   }
25 }
26
27
28 public class ExtensionOverrideDemo {
29   Animal animal;
30   Man man;
31
32   void demo(){
33    animal = new Animal();
34     man = new Man();
35
36     animal.print(); // method from Animal
37
38     man.print(); // method from Man
39   }
40
41    // main instantiates and uses the object
42    public static void main(String[] args){
43      new ExtensionOverrideDemo().demo();
44    }
45 }
```

```
Stomach content is 1
Stomach content is 1
Brain size is 5
```

In the example, `print` is completely redefined in `Man`. It would have been nice to be able to re-use the method `print` from `Animal` in implementing `print` of `Man`, as for instance that method from `Animal` could have been of substantial size. However, the method `print` of `Animal` is not available on objects of type `Man`. Therefore, for writing the code of the subclass, methods of the superclass are made available through prefixing the method name with `super.` .... So the code for the method `print` of `Man` can also be given as:

```
@Override // overridden method
void print() {
  super.print(); // call to print of Animal
  System.out.println("Brain size is " + brain);
}
```

NB Overloading of a method name, using the same name with a different parameter list, makes this name for all programming purposes a new name. So using it in a subclass, leads to addition of a method, not to overriding it.

The constructors of superclasses are also available through `super(parameters)`, which invokes the nearest matching superclass constructor.

In the example this allows for the following application.

```
1  class Animal {
2      int stomach;
3
4      Animal() {
5          stomach = 1;
6      }
7
8      Animal(int s) {
9          stomach = s;
10     }
11
12     void print() {
13         System.out.println("Stomach content is " + stomach);
14     }
15 }
16
17 class Man extends Animal {
18     int brain;
19
20     Man() {
21         // explicit default constructor used first
22         // all other cases need to be programmed explicitly
23         brain = 5;
24     }
25
26     Man(int b) {
27         super(); //! to set stomach tpdefault value
28         brain = b;
29     }
30
31     Man(int s, int b){
32         super(s);
33         brain = b;
34     }
35
36     @Override
37     void print() {
38         super.print();
39         System.out.println("Brain size is " + brain);
40     }
41 }
42
```

```
43 public class ExtensionSuperConstructorsDemo {
44     Animal animal1, animal2; //parameter(s)
45     Man man1, man2, man3;
46
47     void demo(){
48         animal1 = new Animal();
49         animal1.print();
50         System.out.println();
51
52         animal2 = new Animal(2);
53         animal2.print();
54         System.out.println();
55
56         man1 = new Man();
57         man1.print();
58         System.out.println();
59         man2 = new Man(7);
60         man2.print();
61         System.out.println();
62         man3 = new Man(2,7);
63         man3.print();
64         System.out.println();
65     }
66
67     // main instantiates and uses the object
68     public static void main(String[] args){
69         new ExtensionSuperConstructorsDemo().demo();
70     }
71 }
```

```
Stomach content is 1

Stomach content is 2

Stomach content is 1
Brain size is 5

Stomach content is 1
Brain size is 7

Stomach content is 2
Brain size is 7
```

27.3 Execution model

Objects of the subclass are modeled as one object, according to the combined description in the super- and subclass. The call to `print()` in both the version with direct implementation of `Man`'s `print()` and the version using `super` are shown.

Figure 27.1: ExtensionOverrideDemo

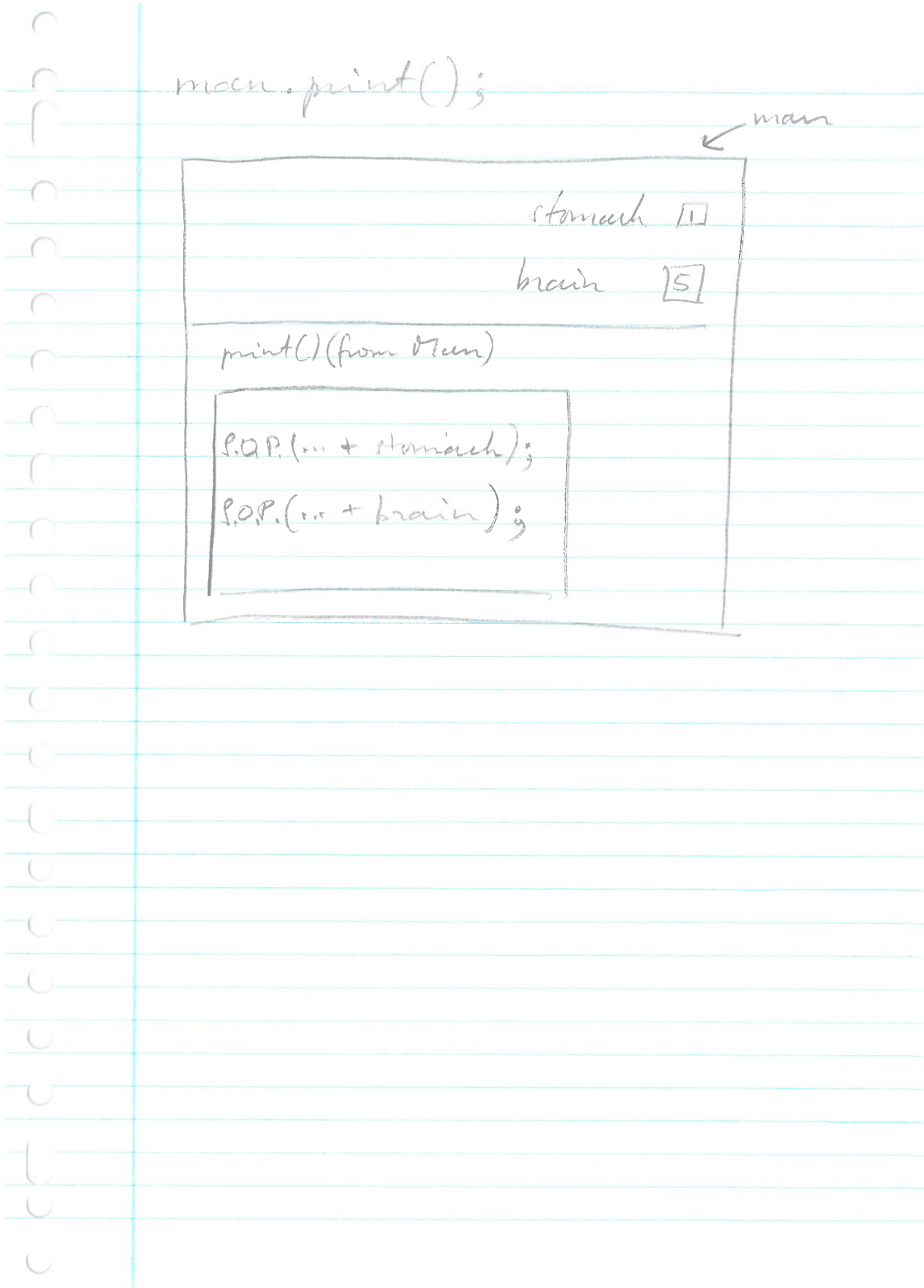
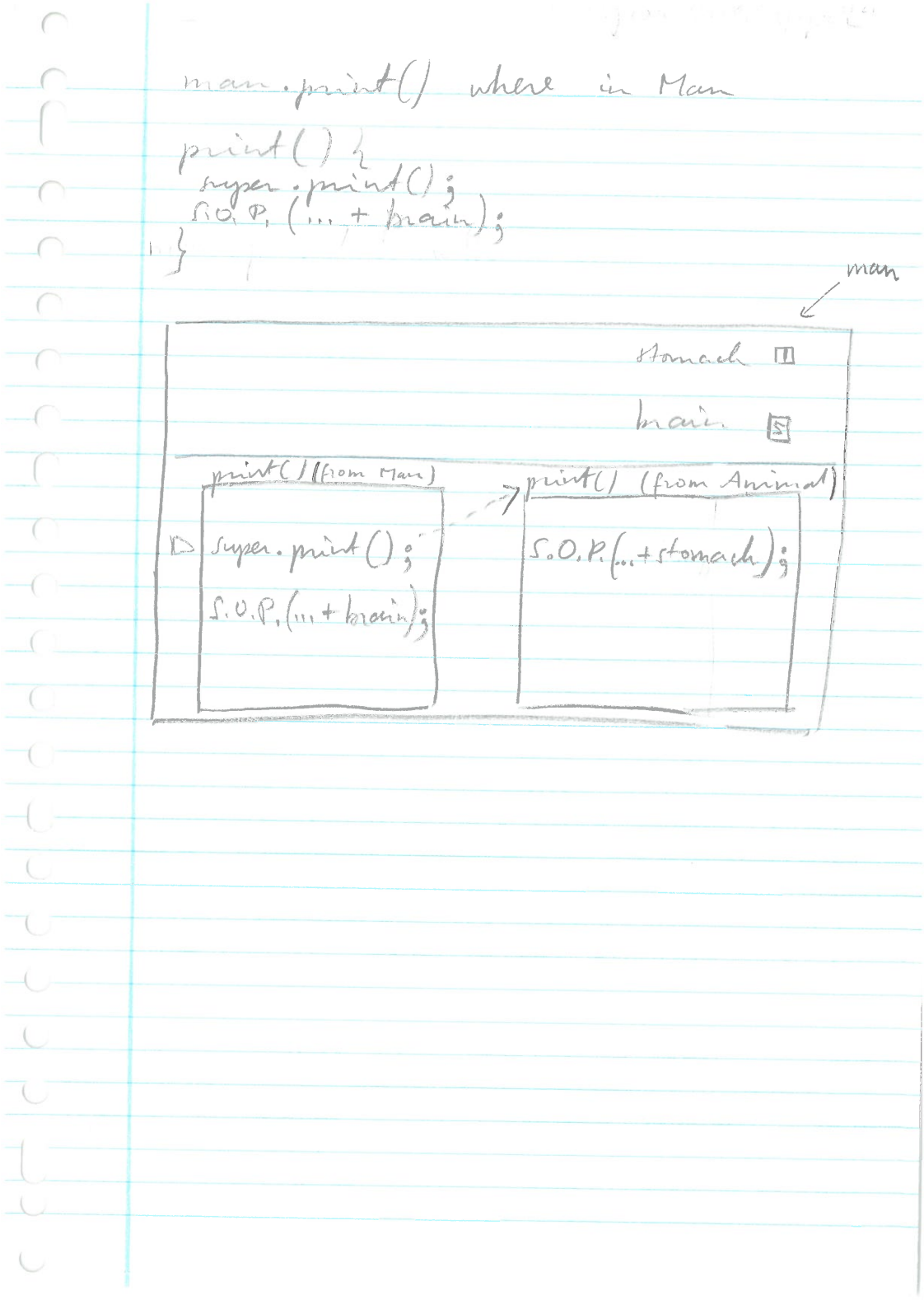
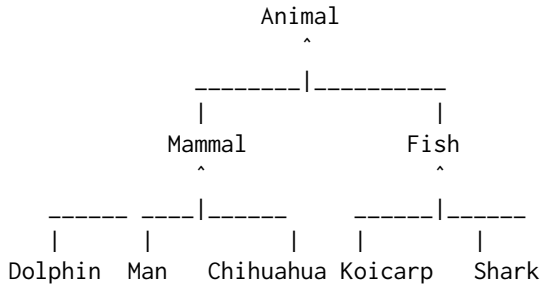


Figure 27.2: ExtensionOverrideSuperDemo



27.4 Remarks

1. Inheritance can be repeatedly applied. This gives rise to structures like upside-down trees, just called trees in computer science.



2. In some languages it is allowed to inherit from more than one class, thus allowing not only trees but also more general directed acyclic graphs. For example, a class `TaxableLuxury` could be a superclass of both `Chihuahua` and `Koicarp` (and `Ferrari`). In Java this is not allowed. An important reason is, that when two different superclasses contain a method with the same name, it is unclear which one the subclass inherits. Thus, in Java the is-a relation provides a single path from sub- to superclasses through the tree: a dolphin is-a mammal is-an animal.

3. Because subclasses inherit functionality, if not overridden, from the superclass, such functionality can be changed uniformly for all subclasses by making the change once in the superclass.

4. The is-a extension is in part a methodological issue, not something that the language (compiler) completely controls: a method can be overridden by one that has illogical functionality, e.g., in the example by a print method that prints the poem the Ancient Mariner. This however would run counter to the aim to understand a subclass as an is-a extension of the superclass.

Chapter 28

Super level code – subtyping, polymorphism, dynamic binding, overriding

Up till now, the inheritance relation between classes was used as a subtype relation for subclasses that were extended with extra methods. The methods that were called in superlevel code on objects of subclasses were superclass methods. Dynamic binding enables to write code that executes the overridden method of subclasses, thus providing the functionality corresponding to subclass.

28.1 Aim

To write superlevel code that for a call to a superclass method maintains the functionality that corresponds to the corresponding overriden method in a subclass.

28.2 Means

The language feature of dynamic binding that if a superclass method is called on an object of a subclass in which that method is overridden, than the corresponding overriding method will be executed.

Example

In the following example the method `demo` contains superlevel code, using a variable `Animal` of type `Animal` in which either a reference to an object of type `Man` or to an object of type `Man` is stored. The method `print()` in superclass `Animal` is overridden in subclass `Man`.

The output is shown for choices a (animal) and m (man).the subclass.

The output is shown for choices a (animal) and m (man).

```
1 import java.util.*;
2
3 class Animal {
4     int stomach;
5
6     Animal() {
7         stomach = 1;
```

```
 8     }
 9
10     void print() {
11         System.out.println("Stomach content is " + stomach);
12     }
13 }
14
15 class Man extends Animal {
16     int brain;
17
18     Man() {
19         stomach = 2;// to show by printing content difference between animal and man
20         brain = 7;
21     }
22
23     @Override
24     void print() {
25         System.out.println("Stomach content is " + stomach);
26         System.out.println("Brain size is " + brain);
27     }
28 }
29
30 public class SuperlevelCodeOverridingDemo {
31     Scanner sc = new Scanner(System.in);
32     Animal animal;
33
34     void demo() {
35         char choice;
36
37         System.out.println("Choose a(nimal) or m(an)");
38         choice = sc.next().charAt(0);
39
40         if (choice == 'a') {
41             animal = new Animal();
42         } else {
43             animal = new Man(); // value of subtype stored in variable of supertype
44         }
45
46         animal.print(); // superlevelcode – one variable for both animal and man objects
47         // same code for printing for both object types
48         // print of dynamic type is used: dynamic binding
49     }
50
51     public static void main(String[] args){
52         new SuperlevelCodeOverridingDemo().demo();
53     }
54 }
```

```
Choose a(nimal) or m(an)
> m
Stomach content is 1

Choose a(nimal) or m(an)
> m
Stomach content is 2
brain size is 5
```

In the example the superlevel code is limited: just the call to `animal.print`.

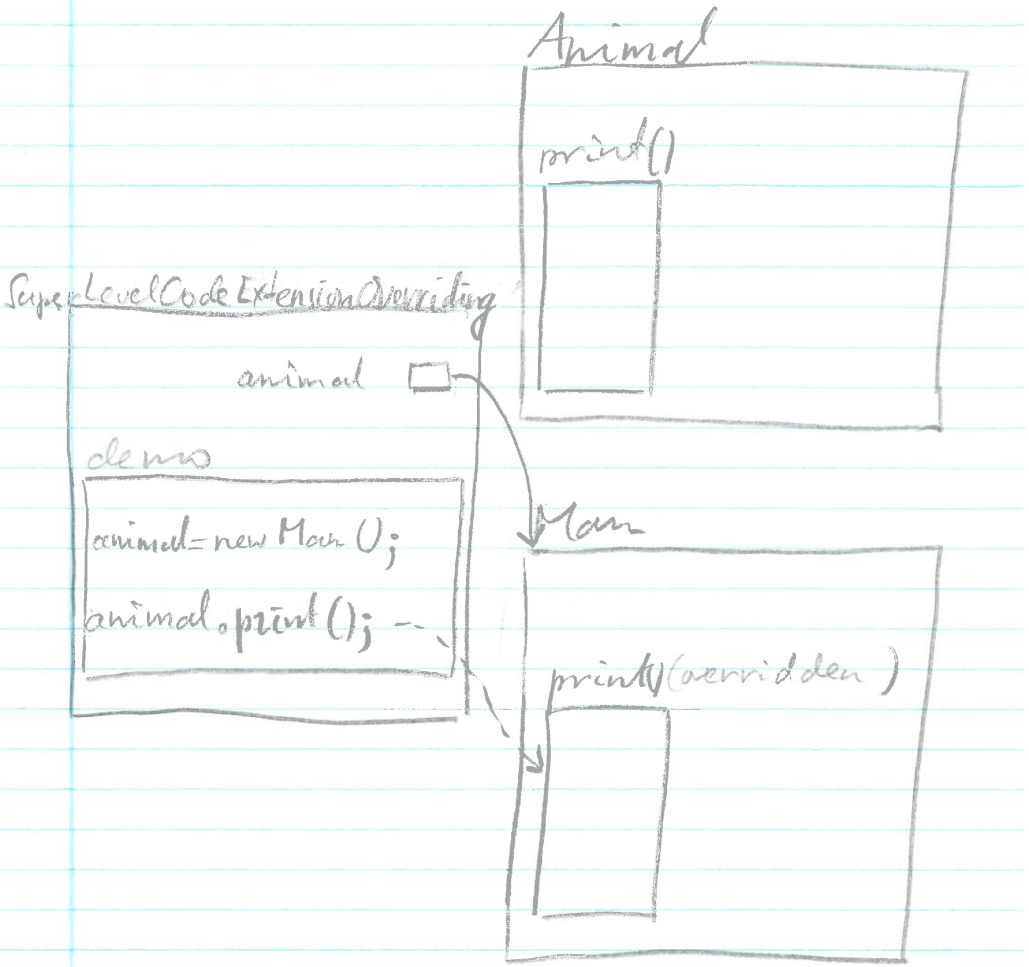
A more useful application of subtyping is an array of supertype in which also objects of subtype(s) can be stored; superlevel code can than be used to pass through the array and call, possibly overridden, methods of the supertype.

This can be pictured as below.

28.3 Execution model

As before.

Figure 28.1: SuperlevelCodeOverriding



Chapter 29

Unimplemented methods for superlevel code – Abstract classes

Up till now superlevel code was written in terms of a superclass. A superclass may be written ”above” two subclasses to enable superlevel code to call a superclass method that was overridden in the subclasses. Sometimes there is no natural implementation for the superclass method. Leaving such a method unimplemented is a solution for which there is language support.

29.1 Aim

For two classes that are not in an is-a relationship and have a different method with the same purpose, to write a superclass with a method to be overridden by these two. If this method has no natural implementation, it should be left unimplemented and be marked as such, e.g., to avoid instantiation to objects that would miss the functionality of that method.

29.2 Means

An `abstract method` in an `abstract class`. Abstract classes cannot be instantiated, as this would create objects that miss the functionality of the unimplemented methods. The compiler will provide an error if an abstract class is instantiated. To obtain an instantiable subclass, all abstract methods need to be implemented.

In the superlevel code also abstract methods named in the super class can be called, on variables of the abstract type that hold references to objects of non-abstract subtype; the corresponding overriding method will be executed.

Abstract classes, apart from having unimplemented methods, have implemented methods that are common to the subclasses. Thus the advantage of shared code is maintained where possible. A consequence is, that the objections against multiple inheritance, namely the possible occurrence of methods with the same name but different implemented behavior remain. Therefore multiple inheritance is also impossible from abstract classes.

Example

In the following example the method `demo` contains superlevel code, using a variable `shape` of abstract type `Shape` in which either a reference to an object of type `Circle` or of type `Square` is stored.

```
1  import java.util.*;
2
3  abstract class Shape {
4      abstract double findArea(); // abstract method
5  }
6
7  class Circle extends Shape {
8      double radius;
9
10     Circle(double r) {
11         radius = r;
12     }
13
14     double findArea() {
15         return Math.PI * radius * radius;
16     }
17 }
18
19 class Square extends Shape {
20     double side;
21
22     Square(double s) {
23         side = s;
24     }
25
26     double findArea() {
27         return side * side;
28     }
29 }
30
31
32 public class AbstractClassDemo {
33     Scanner sc = new Scanner(System.in);
34
35     void demo() {
36         char choice;
37         double size;
38         Shape shape; // variable of abstract type
39         double area;
40
41         System.out.println("Choose c(ircle) or s(quare)");
42         choice = sc.next().charAt(0);
43         System.out.println("Choose size");
44         size = sc.nextInt();
45
46         if (choice == 'c') {
47             shape = new Circle(size); // abstract variable, implemented object
48         } else {
49             shape = new Square(size); ; // abstract variable, implemented object
50         }
51
52         area = shape.findArea(); // abstract method: generic code
53         System.out.println("The area of your shape is " + area);
54     }
55
56     public static void main(String[] args){
57         new AbstractClassDemo().demo();
```

```
58     }
59 }
```

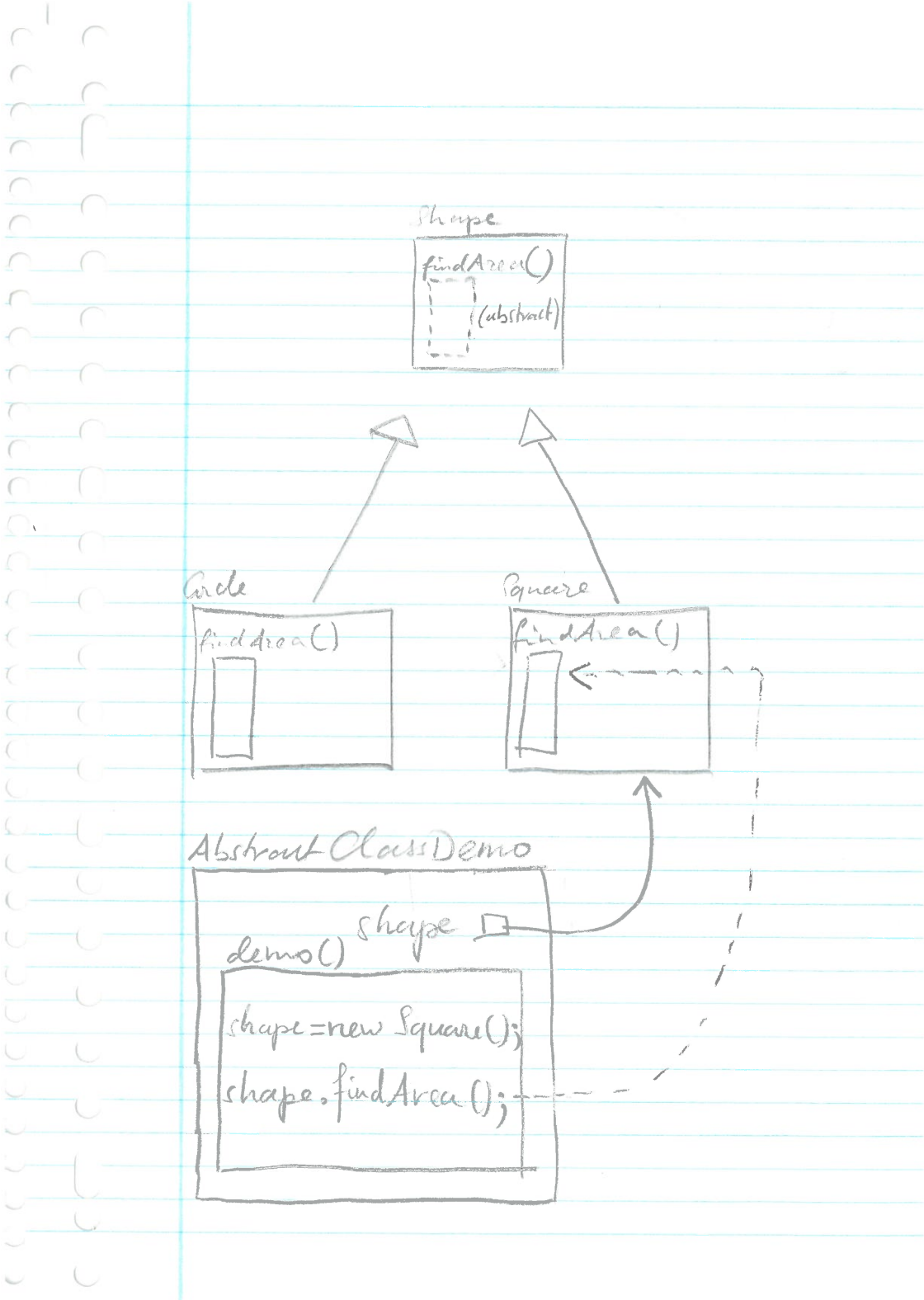
```
Choose c(ircle) or s(quare)
>c
The area of your shape is 78.53981633974483

Choose c(ircle) or s(quare)
>s
The area of your shape is 25.0
```

This can be pictured as below.

29.3 Execution model

As before.

Figure 29.1: AbstractClasses



Chapter 30

Uninplemented class for superlevel code – Interfaces

Up till now superlevel code was written in terms of a, possibly abstract, superclass. Such a superclass was written ”above” a subclass to enable superlevel code to call a superclass method that was overridden in the subclass. Sometimes there is already a class in the inheritance structure directly above that subclass in which a method with the intended functionality does not fit. In such a case, a separate superclass would be required for that method, but this is prohibited by the single inheritance requirement of the language that prevents inheriting methods with the same name. Leaving all methods of the superclass unimplemented is a solution for which there is language support.

30.1 Aim

For two classes that are not in an is-a relationship and have a different method with the same purpose, to write a superclass that has a method to be overridden by these two. If no such superclass would fit in the already present inheritance hierarchy (e.g., because at least one of them has a class above it not having a purpose for that method), a special class of which all methods are abstract (to avoid the confusion that forbade general multiple inheritance) could be placed in a multiple inheritance relation with them.

30.2 Means

An *interface* is an abstract class where all methods are unimplemented, i.e., that consists of abstract methods only. Because in interfaces all methods are abstract, the keyword `abstract` is omitted. Interfaces cannot be instantiated, as this wuold lead to objects that miss the functionality of the unimplemented methods. The compiler will provide an error if an abstract class is instantiated. To obtain an instantiable subclass, all abstract methods need to be implemented. We acquire the names for the interface for the superlevel-code by means of the keyword `implements`: the first line of the class definition is $\langle SubClass \rangle$ `implements` $\langle Interface \rangle$.

In the superlevel code the abstract interface methods named in the interface can be called, the corresponding overriding method will be executed.

A subclass can, possibly additional to extending one class through `extends`, implement multiple interfaces through `implements`.

Example

In the following example we add the capability of being taxable to the classes derived from `Animal`. This is done by requiring that a class `TaxableAnimal`, additionally to inheriting from `Animal`, implements `Taxable`.

`TaxableAnimal` now also has to implement the method `computeTax`. Other classes can also implement `Taxable`, e.g., `TaxableShape` or `TaxableCircle`. Now we can write a class `TaxAdmin` where objects of these classes can be registered. The code of this class is superlevelcode in the sense that it can contain code like

```
Taxable taxItem;
...
int tax = taxItem.computeTax();
```

where `taxItem` can be a `taxableAnimal` or an object of any other class as long as it implements `Taxable`.

Note, that being taxable is a small, partial capability that is quite different and independent of the features of an animal. It therefore is natural that being taxable does not have a place in the primary, tree-structured, hierarchy of animals.

Note, that an interface was motivated as enabling to provide a (fully unimplemented) class to be placed above two subclasses to enble writing superlevel code, in the example just one subclass, taxable animal is povided. This is because for the explanation of the multiple inheritance structure it is enough to only consider the subclass that requires (to avoid multiple inheritance) the interface to be used. If so desired, add anopther subclass, e.g., `TaxableShape`.

An important other application of interfaces, e.g., when using framework code as in graphical applications, is to provide the connection between framework code and the specific graphics application. In this case also just one class implementing an interface occurs.

```
1  public class InterfaceDemo {
2    Taxable taxItem; // abstract variable taxItem
3
4    void demo() {
5    int tax;
6
7      taxItem = new TaxableAnimal(); // abstract variable taxItem, implemented object
8
9       tax = taxItem.computeTax(); // abstract method: superlevel code
10
11     System.out.println("The tax is " + tax);
12   }
13
14   public static void main(String[] args){
15    new InterfaceDemo().demo();
16   }
17 }
18
19 class Animal {
20   int weight;
21
22   Animal() {
23     weight = 5;
24   }
25 }
26
27 interface Taxable {
28   int computeTax();
29 }
30
31 class TaxableAnimal extends Animal implements Taxable{
32
```

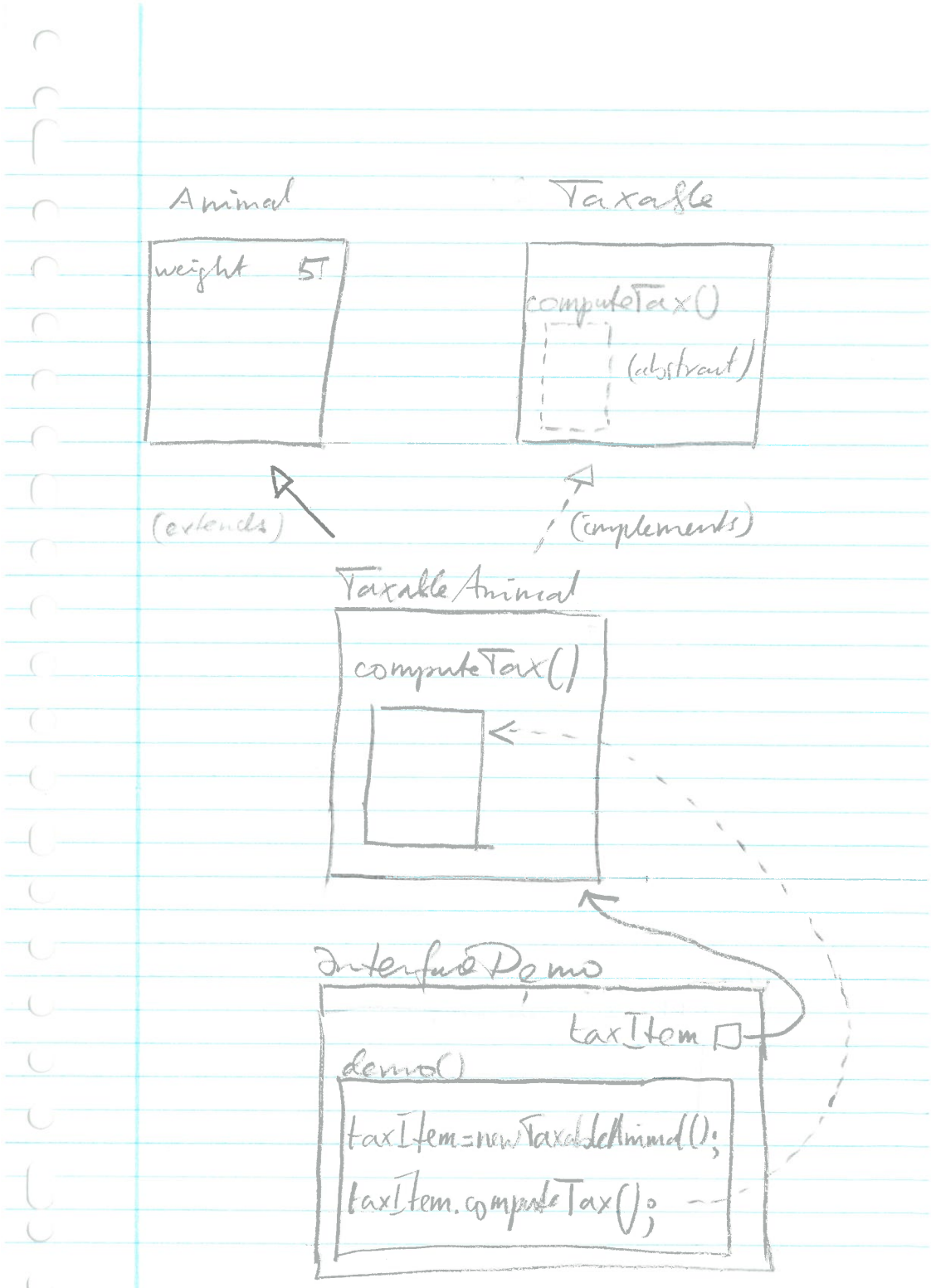
```
33   public int computeTax() {
34    return 3 * weight;
35   }
36 }
```

This can be pictured as below.

30.3 Execution model

As before.

Figure 30.1: Interfaces



Chapter 31

Specialization scoping – modifiers

Up til now visibility modifiers only enforced accessibility for classes that were in composition, has-a, relation. Now this is extended to classes that are in an inheritance relation

31.1 Aim

Decrease dependencies between classes that are in an inheritance relation.

31.2 Means

Restrict what classes can see and use from other classes. Methods and variables can be hidden from other classes. This way, the other classes can not use the hidden methods and variables and hence do not depend on them.

Packages

A *package* is a group of classes that belong together and are labeled as such by the programmer. An example is `java.io`, a set of classes that deals with input and output and files. The class `Scanner` is part of the package `java.io`. You can declare that the classes in a file belong to package `myPackage` by putting the line

```
package myPackage;
```

at the beginning of the file *and* putting the classes together in a folder with this name.

`protected`: visible only in classes of the same package and in all subclasses.

When you leave out the line with `package`, the classes belong to the so-called *anonymous* package for which the name of the folder is irrelevant.

Overview of visibility modifiers (also the ones for the composition relation) from wide to narrow:

- `public`: no limitations, visible in all classes;
- `protected`: visible only in classes of the same package and in all subclasses[1];

[1] subclasses are treated in Part III

- *none* (sometimes called ”package”): visible only in the classes of the same package (this is what we have used mostly so far);
- `private`: visible only in the same class.

Bibliography

[1] The Java Language Specification, 3rd ed., http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html, September 7, 2006.